macromedia®

SHOCKWAVE MULTIUSER SERVER

Using Shockwave Multiuser Server

macromedia®

Acknowledgments

Project Management: Joe Schmitz

Documentation: Jay Armstrong, Christopher Basmajian, Rosana Francescato, Mark Castle

Special thanks: Sarah Allen, Christopher Campbell, Margaret Dumas, Mike Edmunds, Thomas Higgins, Alan Isaacs, William Jones, Allison Kelsey, Steve Kury, Dave Simmons

Second Edition: September 1999

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

CONTENTS

CHAPTER 1
Using the Shockwave Multiuser Server and Xtra

Overview

The Shockwave Multiuser Xtra and Server allow Director movies to exchange information with other Director movies over the Internet or smaller networks. The Xtra and a 50-user version of the server are included with the Director Shockwave Internet Studio. The Xtra is also included with Shockwave 7.

You can use the Multiuser Server and Xtra to do the following:

- Create a custom chat movie that allows real-time conversation.
- Hold an online meeting with a shared "whiteboard" that each participant can write on.
- Provide a shared presentation, with a presenter and an audience who all watch the presentation at the same time.
- Run a multiplayer interactive game.

Movies that use the Multiuser Xtra can exchange information in three basic ways:

- By sending it to the Shockwave Multiuser Server, which then sends it on to the movie or movies it is intended for.
- By establishing peer-to-peer connections directly with other movies.
- By connecting to a text-based server such as a standard mail or Internet Relay Chat server. In order to communicate with a text-based server, you must be familiar with the commands the server understands. You can send these commands to the text-based server in the same way you send other messages.

The Shockwave Multiuser Server and Xtra are two separate components that work together to enable multiuser movies. The server is a separate application that runs on a separate computer. The Director Xtra checks messages for errors, prepares them for passage over the network, and then sends them to the server. The server then determines whom the message was intended for and sends it to the appropriate recipient(s). The recipient's Xtra gets the message from the network so that it can be used by the movie.

When using the server, movies can communicate with other instances of the same movie connected to the server (such as a chess movie exchanging player-move information with other instances of the chess movie), or with different movies (such as a chess movie exchanging chat messages with a checkers movie in a virtual game room).

The types of messages your movie sends depends on what you want the movie to do. Movies can share all the types of data that Lingo supports, including strings, integers, floating point numbers, colors, dates, points, rects, and lists. In addition, cast members may be exchanged by using the `media` or `picture of member` cast member properties. This enables chat movies to share pictures of the participants, for example.

As a Lingo Xtra, the Multiuser Xtra extends Lingo by adding new commands and other elements to the Lingo vocabulary. The Xtra is used by writing Lingo scripts that include these commands. The multiuser behaviors included in Director's Behavior Library provide all of the functionality that a basic chat application requires.

In addition to simply passing information from movie to movie, the Shockwave Multiuser Server also provides functions that make it easier to create rich and complex multiuser experiences:

- You can store and retrieve information such as user names or profiles in databases.
- You can create groups in order to organize users in logical ways, such as opposing teams in an adventure game.
- You can assign attributes to those groups, such as a team's score.
- You can send messages directly to the server to get information about the server and the other movies connected to it.

What’s new in Version 2.0

Version 2.0 of the Shockwave Multiuser Server offers important new features for creating rich, flexible multiuser experiences on the Internet. Delivering multiuser capabilities in browsers and in projectors is now easier and more powerful. The improved Lingo interface and increased messaging and database flexibility allow for greater creativity with your multiuser projects. This version of the server still uses Version 1 of the Multiuser Xtra.

Intermovie messaging

Movies can now send messages to any other Director movie connected to the server, allowing you to page other users on the server and giving you greater flexibility with movie communication.

Relational databases

A powerful new relational database model allows for easy storage and retrieval of persistent data, such as high scores or game status.

Group attributes

In addition to organizing connected users into the groups of your choice, you can now assign attributes to those groups. You can store as an attribute anything the group members have in common, such as a group’s score. This enables movies to share data efficiently.

Improved Lingo syntax

The syntax for sending administrative commands to the server is now more consistent and allows for greater flexibility when setting up message callback handlers. See “Sending messages”.

Server extensibility

The server can be extended with Xtras. You can implement your own server functions by writing code that accesses the MOA interfaces of the server.

Creating multiuser movies

To create a multiuser experience, you must design a movie that connects to the server or directly to another movie and then sends and receives messages that contain the information needed for the movie to function. For example, you might design a chat movie that connects to the server and then allows each user to send text messages and pictures to all users of the chat movie or to specific users. Or you might design a Ping-Pong game that has two people connect to the server and then sends messages describing the location of each user's paddle and score.

Connecting to the server

To connect to the Shockwave Multiuser Server, you must write Lingo scripts that execute each step of setting up the Multiuser Xtra and establishing the Xtra's communication with the server. See "Multiuser Scripting Elements" for detailed explanations of each Multiuser Xtra Lingo command. You may also use the multiuser behaviors included in Director's Behavior Library to connect to the server without writing Lingo of your own.

To set up a server connection for sending messages, follow these steps:

1. Make sure you have a working server running on a computer on your network and that you know its network address.
2. Create an instance of the Multiuser Xtra.

   For example, this statement places the Xtra instance into the global variable gMultiuserInstance:

   ```
   gMultiuserInstance = new(xtra "Multiuser")
   ```

   A single Xtra instance always corresponds to a single server connection. To make more than one connection at a time, use multiple Xtra instances. When you are ready to disconnect from the server, set the variable containing your Xtra instance to zero.

3 Set up one or more callbacks for handling incoming messages using `setNetMessageHandler`.

When your movie initially connects to the server, the server responds with a message confirming the connection. In order to handle this and subsequent messages, you must create callbacks. These are Lingo handlers that are run when messages arrive at a movie from the server or from peer-to-peer users. You must create at least one callback before your movie connects to the server. The arrival of a message is treated as a Lingo event in the same way that a mouse click is treated as a `mouseUp` event. Once the callback is declared in Lingo it will be triggered each time a message arrives from the network. A callback handler should be written to perform tasks based on the contents of the incoming message that triggers it.

For example, this statement declares that the handler `defaultMessageHandler` in the script cast member Connection Script is the handler to run when any incoming message is received from the server:

```
errCode = gMultiuserInstance.setNetMessageHandler(#defaultMessageHandler, script
"Connection Script")
```

The message handler would look like this:

```
on defaultMessageHandler
    newMessage = gMultiuserInstance.getNetMessage
    member("messageOutput").text = newMessage
    if newMessage.errorCode <> 0 then
        alert "Incoming message contained an error."
    end if
end
```

The first parameter you specify with `setNetMessageHandler()` is a symbol for the handler you are declaring. You make it a symbol by adding a pound sign (#) to the beginning of the handler name. The second parameter is the name of the script object that contains the handler. This could be a script cast member or the name of a variable containing a behavior instance, a parent script, or a simple Lingo value to be passed to global scripts.

You can specify individual handlers to be run based on a particular message subject, a sender, or both by adding optional parameters to `SetNetMessageHandler()`. These individual handlers are run instead of the default message handler when the specified type of message arrives.

This statement declares a handler that runs when a message containing the subject Chat Text arrives from sender Guest Speaker:

```
errCode = gMultiuserInstance.setNetMessageHandler(#guestMessageHandler, script
"Connection Script", "Chat Text", "Guest Speaker")
```

See "Sending messages" for information about message subjects, senders, and contents.

4 Establish the server connection with connectToNetServer.

When the message handlers have been declared, you are ready to make your server connection. The Xtra connects to the server with a user name and password, along with other parameters you supply. You must know the server's address and port number as well as the name you want your movie to use to identify itself.

This Lingo connects the movie Tech Chat to a Shockwave Multiuser Server with a user ID of Bob and a password of MySecret. The example server name is chatserver.mycompany.com and the communications port number, 1626, is the default.

```
errCode = gMultiuserInstance.connectToNetServer("Bob", "MySecret",
"chatserver.mycompany.com", 1626, "Tech Chat")
```

The server uses the name you provide for the movie to associate other instances of the same movie with each other. By default, all messages sent by the movie Tech Chat will be sent only to other users of the same movie on the network.

When the server accepts the connection, it will respond with a message whose subject is ConnectToNetServer, which will trigger the defaultMessageHandler you declared earlier.

The entire Lingo script for connecting to the server looks like this:

```
on makeAServerConnection
    -- declare a global variable to hold the Xtra instance
    global gMultiuserInstance

    -- create the Xtra instance
    gMultiuserInstance = new(xtra "Multiuser")

    -- declare message handler callback(s) to handle incoming messages,
    -- including the server's initial connection response
    errCode = gMultiuserInstance.setNetMessageHandler(#defaultMessageHandler, script
    "Connection Script")
    if errCode <> 0 then
        alert "Problem with setNetMessageHandler"
    end if

    -- connect to the server
    errCode = gMultiuserInstance.connectToNetServer("Bob", "MySecret",
    "chatserver.mycompany.com", 1626, "Tech Chat")
    if errCode <> 0 then
        alert "Problem with connectToNetServer"
    end if

end

-- message handler for incoming messages
on defaultMessageHandler
    newMessage = gMultiuserInstance.getNetMessage
    member("messageOutput").text = newMessage
    if newMessage.errorCode <> 0 then
        alert "Incoming message contained an error."
    end if
end
```

The following code contains two message callback handlers. One will handle generic messages and one will handle messages with a #subject property of Chat Text.

```
on makeAServerConnection
    -- declare a global variable to hold the Xtra instance
    global gMultiuserInstance

    -- create the Xtra instance
    gMultiuserInstance = new(xtra "Multiuser")

    -- declare message callback handler to handle incoming messages that don't meet
    -- the specific criteria of the second message callback handler declared below
    errCode = gMultiuserInstance.setNetMessageHandler(#defaultMessageHandler, script
    "Connection Script")
    if errCode <> 0 then
        alert "Problem with setNetMessageHandler"
    end if

    -- this is the second message callback declaration
    -- declare a specific message callback handler for messages with #subject Chat Text
    errCode = gMultiuserInstance.setNetMessageHandler(#chatMessageHandler, script
    "Connection Script", "Chat Text")
    if errCode <> 0 then
        alert "Problem with setNetMessageHandler"
    end if

    -- connect to the server
    errCode = gMultiuserInstance.connectToNetServer("Bob", "MySecret",
    "chatserver.mycompany.com", 1626, "Tech Chat")
    if errCode <> 0 then
        alert "Problem with connectToNetServer"
    end if
end

-- message handler for generic incoming messages
on defaultMessageHandler
    newMessage = gMultiuserInstance.getNetMessage
    member("messageOutput").text = newMessage
    if newMessage.errorCode <> 0 then
        alert "Incoming message contained an error."
    end if
end

-- message handler for incoming messages with #subject Chat Text
on chatMessageHandler
    newMessage = gMultiuserInstance.getNetMessage
    put newMessage.content after member "chatField"
    if newMessage.errorCode <> 0 then
        alert "Incoming message contained an error."
    end if
end
```

Sending messages

When you have successfully connected to the server, you are ready to send messages. Your messages may contain any kind of data that Lingo supports. The messages you receive will take the form of Lingo property lists, and you may choose to send your outgoing messages as property lists as well.

To send any kind of data as a message to another movie, use sendNetMessage. This function can take two forms: either a property list or the same parameters, separated by commas.

```
errCode = gMultiuserInstance.sendNetMessage([#recipients: whichUsersOrGroups,
#subject: “Example Subject”, #content: whatMessage])
```

or

```
errCode = gMultiuserInstance.sendNetMessage(“whichUsersOrGroups”, “Example Subject”,
whatMessage)
```

You can send messages to specific users and to named groups of users in the same movie. For example, user Guest Speaker in the movie Tech Chat could be in New York and send a message to user Bob in the same movie in San Francisco.

```
errCode = sendNetMessage(gMultiuserInstance, “Bob”, “Chat Text”, “Hello. Welcome to our
conversation.”)
```

To send a message to all the members of a group, use the group name, in this case @MultimediaAuthors, as the recipient parameter. You must begin all group names with the @ sign. See the “Using groups” section for information on creating groups.

```
errCode = gMultiuserInstance.sendNetMessage(“@MultimediaAuthors”, “Chat Text”, “How
is everyone doing?”)
```

You can also send messages to users of other movies connected to the server. A player of a Chess game might send a message to a player of a Checkers game:

```
errCode = gMultiuserInstance.sendNetMessage(“Chris@Checkers”, “Question”, “Who’s
winning?”)
```

To send a message to a user in any other movie on the server, use @System after the user name. You can use this to page a user to see if he or she is connected to the server at all.

```
errCode = gMultiuserInstance.sendNetMessage(“Chris@System”, “Page”, “Are you there?”)
```

Retrieving messages

To retrieve messages from the server, use getNetMessage in your message callback handlers.

```
newMessage = gMultiuserInstance.getNetMessage
```

The contents of newMessage will be a property list that looks like this:

```
[#errorCode: 0, #recipients: ["Bob"], #senderID: "Guest Speaker", #subject: "Chat Text",
#content: "Hello. Welcome to our conversation.", #timeStamp: 66114697]
```

You can set up individual message handlers to be triggered by specific values in the #subject or #sender properties and to perform different actions based on the values of each of the properties in the incoming message.

#errorCode contains an integer that can be translated to a usable error message with getNetErrorString.

#recipients contains a list of strings that are the names of the users the message was sent to.

#senderID contains a string that is the user name of the sender.

#subject contains a string chosen by the movie author. You can use this string to indicate what kind of data is in the message and to have the message processed by a specific message callback handler.

#content can contain whatever type of Lingo data your movie requires, including integers, floating point numbers, strings, lists, rects, points, colors, dates, and the media of member and picture of member properties.

#timeStamp contains an integer that is the number of milliseconds since the server was launched. You can use this number to synchronize events in several instances of your movie or to determine how much time the server is using to process your messages.

Once you have retrieved a message with getNetMessage() inside a callback handler, you can write additional Lingo to use the contents of the message to do specific things in your movie. For instance, you can use the data to set the location of sprites, add new images to the stage, display chat text, update catalog information, or keep a running score in a game.

The Multiuser Xtra checks the network for incoming messages during idle time. Some complex movies may not allow enough idle time to retrieve all the incoming messages that the movie is receiving. You can determine how many messages have arrived but have not yet been processed by using the getNumberWaitingNetMessages command, which will give you the number of unprocessed messages. You can then use this number to call checkNetMessages(), which will process the number of waiting messages you specify.

Error checking

Checking for errors in your multiuser movies is essential. Latency and stalled connections are common network problems. Servers may be temporarily unavailable because of power failures or other physical malfunctions. Error checking will let you detect and fix problems so that the end-user experience is not affected.

Because most of the commands used with the Shockwave Multiuser Server and Xtra are dependent on one another, it is important to verify that each command used is successful before executing the next one. Many multiuser commands can produce both synchronous and asynchronous results. A synchronous result is one returned immediately by the Xtra, which indicates whether the Xtra was able to correctly execute the command with the parameters you provided. An error code of 0 indicates success.

An asynchronous result is one that may be returned after some operation takes place on the server, such as when your connectToNetServer request is accepted or declined. This error code is provided as part of the contents of an incoming message and should be checked in each of your message callback handlers.

The following Lingo script sets the variable errCode to the number returned by the Xtra as the immediate result of the connectToNetServer command. If one or more of the parameters you provide is unacceptable to the Xtra, it will return a nonzero result and trigger an alert.

```
errCode = gMultiuserInstance.connectToNetServer("Bob", "MySecret",
"chatserver.mycompany.com", 1626, "Tech Chat")
if errCode <> 0 then
    alert "Problem with connectToNetServer" & RETURN & ¬
        gMultiuserInstance.getNetErrorString(errCode)
end if
```

To check the asynchronous results returned from commands as the #errorCode property of incoming messages, use a Lingo script such as the following:

```
on defaultMessageHandler
    newMessage = gMultiuserInstance.getNetMessage
    member("messageOutput").text = newMessage
    if newMessage.errorCode <> 0 then
        alert "Incoming message contained an error."
    end if
end
```

You should keep checking the error codes in the messages you receive so that you are aware when conditions change in ways that will affect your movie. For instance, other movies on the network may disconnect or other events may prevent them from communicating efficiently with your movie. When you receive a message containing an error, use getNetErrorString to convert the numeric error code into a useful description of the error that occurred.

For example, these statements place the string returned by getNetErrorString into a field for the user to see:

```
newMessage = gMultiuserInstance.getNetMessage
errCode = newMessage.errorcode
member("errorText").text = gMultiuserInstance.getNetErrorString(errCode)
```

Setting up peer-to-peer connections

In some situations, you might prefer to design your movies to communicate directly with one another. By using the peer-to-peer functions of the Multiuser Xtra, you can set up one instance of a movie as a virtual server, or peer host, and connect up to 16 other movies to it. This method lets you set up private chats, create games for limited numbers of users, or make presentations to small groups, without using a server. The disadvantage of using peer connections is that you do not have access to the server's group, database, or administrative functions.

One of the movies in a peer-to-peer connection must be the host. You set up a movie to be a peer host by using waitForNetConnection. Once the movie is in peer-host mode, other movies can connect to it in the same way they connect to the server using connectToNetServer.

Movies that wish to connect to a peer host must know the Internet address of the computer the host movie is running on. This will be either a number, such as 192.98.1689.1, or a name such as myServer.myCompany.com. You can get the numeric Internet address of your computer with getNetAddressCookie. By default, this function returns an encrypted version of your IP address. In addition getNetAddressCookie can return an unencrypted result, but only in projectors. One way to make the address available to other users is to meet them on the server and then send them your Internet address cookie before making a peer connection.

You send messages in peer-to-peer mode in the same way as when connected to the server. You can send messages directly to other peer users or to the peer host. Messages sent to other peers are routed to the recipient by the Xtra in the host movie, but their contents are not made available to the host movie. Only messages sent to the host user or to System are actually received by the host as messages that can be accessed in Lingo. In peer-host mode, the Xtra does not provide any of the server features such as group management or database access.

Peer hosts can control who is connected to their movie by keeping track of the list of connected users and breaking connections if necessary. Using getPeerConnectionList, you can get a list of the current peer connections on a peer host. You can sever the connection of a specific user in the list by using breakConnection.

When implementing a peer-to-peer movie design, you may choose to make one Director movie that always acts as a host and another Director movie that is used for the clients. You may also choose to author your movie so that it can function as either host or client, allowing anyone who uses the movie to initiate the peer-to-peer session.

Keep in mind that because all peer messages are routed through the peer host, there will be an extra CPU burden and network throughput required on that machine. The degree of this burden will depend on the size and number of messages your movie sends and the number of peers connected to the host. See the "Optimizing multiuser movies" section for ways to minimize this load.

Creating text connections

Another way the Multiuser Xtra can communicate is through text-based servers such as Internet Relay Chat, Internet mail servers, or even proprietary text-based servers. These servers are called text-based because they respond to simple text commands that are used to control them. You make a text-based server connection by adding a 1 to the end of the `connectToNetServer()` command, which tells the Xtra to enter text mode:

```
errCode = gMultiuserInstance.connectToNetServer(“Bob”, “MySecret”,
“mailserver.mycompany.com”, 110, “Tech Chat”, 1)
```

The port number must also change to reflect the port your text-based server is using for communication. In this example the port is 110, which is commonly used with Internet mail servers. Consult the documentation for your text-based server to determine which port it uses.

After the connection is made, you issue commands to your server with `sendNetMessage()`. Use System as the recipient and place your command into the content parameter. The subject parameter will be ignored. The following example sends a command to a mail server to retrieve the first piece of mail in the mailbox.

```
errCode = gMultiuserInstance.sendNetMessage(“System”, “anySubject”, “RETR 1”)
```

The server will respond with whatever data is appropriate. The Xtra will retrieve the data as the `#content` property of a typical incoming message property list.

Using server functions

You can retrieve many kinds of information from the Shockwave Multiuser Server and perform administrative functions by sending special commands to the server in the contents of sendNetMessage(). You can get a list of movies connected to the server, control users' and movies' access to the server, and get connected users' IP addresses. Using these functions, you can design a movie that lets you remotely monitor and control activity on the server.

The server commands themselves are sent in the recipient parameter of a sendNetMessage(), and any additional parameters required by the command are sent as the content. To determine the version of the server you are using and the platform it is running on, use getVersion:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.getVersion", "anySubject")
```

When using server functions, the recipient parameter uses a simple dot syntax that contains a reference to the server (since this is a server command), the object the command will be performed on, and the command itself. In this case the object is the server; in other cases, it could be a user or a movie.

The message returned from the server containing the version and platform information will look like this:

```
[#errorCode: 0, #recipients: ["Bob"], #senderID: "system.server.getVersion", #subject:
"anySubject", #content: [#vendor: "Macromedia", #version: "2.0", #platform:
"Macintosh"]]
```

To get a list of all the movies connected to the server, use getMovies:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.getMovies", "anySubject")
```

To prevent all instances of a particular movie from connecting to the server, use movie as the object and disable as the command. Here the movie being disabled is Tech Chat:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.disable", "anySubject",
"Tech Chat")
```

To disconnect a specific user from the server, the object is user and the command is delete:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.delete", "anySubject",
"RudePerson")
```

Server commands, including getMovies, disable, and delete, require the sender to have a specific user level. The required user levels for each command are set in the server's Multiuser.cfg file. See "Configuring the server" for more information.

Some server commands can be used to return information about a movie other than the one that sends the message. If you want to get the list of groups in a movie called Chess but you are using an administrative movie called GameAdmin, you can send the command getGroups to the movie Chess:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.getGroups@Chess",
"anySubject")
```

See "Multiuser Scripting Elements" for a complete list of the server functions that are available.

Using groups

Another function that the Shockwave Multiuser Server provides is the ability to define groups of connected users. By creating groups, you can organize users in logical ways, such as into opposing teams, collaborating companies, and so on.

Using groups allows you to do the following:

- Send a single message to all the members of a group.
- Prevent users from sending messages to users outside their own group.
- Assign attributes to the groups, such as scores, team colors, or group leader.

Defining groups

When users connect to the server, they are automatically added to the group @AllUsers for the movie they are using. This lets you easily send messages to everyone connected to the movie without having to create a new group and have everyone join it individually.

You create a new group by having one or more connected users join a group name that does not yet exist. If the group does not exist when the first user joins, the server creates the group for you. Once the group has some members, you can ask the server for the number of members in the group, for a list of who the members are, or for any attributes that have been assigned to the group.

Once a movie has connected a user to the server, the user can join a new group by sending a message to the server specifying `group` as the object and `join` as the command in the recipient parameter of a `sendNetMessage()` and sending the group name in the `#content` parameter.

```
errCode = gMultiuserInstance.sendNetMessage("system.group.join", "anySubject",
"@RedTeam")
```

This message above joins the group called @RedTeam. It is important that all your group names begin with the @ symbol. The server uses this symbol to distinguish group names from user names when routing standard messages like this one:

```
errCode = gMultiuserInstance.sendNetMessage("@RedTeam", "Status Update", "Your team
is winning.")
```

If the recipient were simply RedTeam, the server would attempt to find the user named RedTeam instead of the group. No error message is generated when a user is not found by the server.

You can leave a group by specifying `system.group.leave` in the recipient parameter of a `sendNetMessage()` and indicating which group you want to leave in the `#content`:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.leave", "anySubject",
"@RedTeam")
```

You can determine how many users are in a group by using getUserCount. For example, you can use this to keep team sizes even as more users join a game by adding users to whichever team has a smaller number of players. Or you could create a new group when an existing one exceeds a certain number of users.

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getUserCount",
"anySubject", "@RedTeam")
```

If you want to get a list of the users in a group, use getUsers:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getUsers", "anySubject",
"@RedTeam")
```

Working with group attributes

You may wish to keep track of the traits of a group or set of groups. For example, for groups of people working for two different companies that are collaborating, you might want to store the name of the leader of each group. If the groups are negotiating the price of a contract, you might store the current amount of each group's bid. In a gaming scenario, each team might choose a mascot, a theme song, or a starting location in a virtual world. Each of these can be stored as an attribute of a group as long as the group exists. Attributes can be set to any of the Lingo values that can be sent using sendNetMessage. Group attributes persist as long as the group persists on the server. To store information that must persist indefinitely on the server, use databases instead; see Using databases for more information.

You use setAttribute to add an attribute or update its value and getAttribute to determine the current value of the attribute. These commands are specified in the #recipient parameter of a sendNetMessage(), and the attributes you wish to access are specified in the #content parameter of the message.

To define the current location and mascot of the group @RedTeam, your Lingo would look like this:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.setAttribute", "anySubject",
[#group: "@RedTeam", #attribute: [#currentLocation: "New York City", #mascot:
"dalmatian", #lastUpdateTime: "1999/07/27 15:52:11.123456"]])
```

When you use setAttribute, the #content parameter of your message contains a #group property and a #attribute property. The #group property indicates which group you are setting attributes for. The #attribute property indicates which attributes you are setting. The names of the attributes you set should always be symbols beginning with the # character. In the example above, the #group is @RedTeam and the #attribute is a list of two attributes, including #currentLocation and #mascot.

The #lastUpdateTime property is optional and lets you determine whether some other user has updated the attributes of the group since you last checked them with getAttribute. When you use getAttribute, the server responds with the values of the attributes you requested plus a #lastUpdateTime property, which indicates the moment in time when the server read the values of those attributes for the group you requested. The #lastUpdateTime property is a string containing the year, month, day, hour, minutes, seconds, and microseconds on the server. By sending this same string with your setAttribute command, you allow the server to check whether the attributes for the group have been updated since you last checked them.

If the server determines that any attribute for the group has been updated by someone else since you checked them, it will respond with a #errorCode indicating a concurrency error. If no one else has updated the attributes since you checked them, the server will respond with a new #lastUpdateTime for the group, indicating that you have just updated the attributes.

This statement gets the values of the attributes #currentLocation and #mascot for the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getAttribute", "anySubject",
[#group: "@RedTeam", #attribute: [#currentLocation, #mascot]])
```

The server's response will look like this:

```
[#errorCode: 0, #recipients: ["Bob"], #senderID: "system.group.getAttribute", #subject:
"anySubject", #content: ["@RedTeam: [#currentLocation: "New York City", #mascot:
"dalmatian", #lastUpdateTime: "1999/07/27 15:53:32.123456"]]]
```

You can get attributes for more than one group at a time by including multiple group names in your request:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getAttribute", "anySubject",
[#group: ["@RedTeam", "@BlueTeam"], #attribute: [#currentLocation, #mascot]])
```

When you make a request for multiple groups, it is possible that part of the request will succeed while part of it produces errors. In the example above, attributes are requested for the group @BlueTeam. If this group has not yet been created, the server response will include errors in both the #errorCode property of the response and the attributes list for @BlueTeam. If a requested attribute has not been set, no error is generated and the attribute is simply omitted from the response.

The server's response will look something like this:

```
[#errorCode: -2147216173, #recipients: ["Bob"], #senderID: "system.group.getAttribute",
#subject: "anySubject", #content: ["@RedTeam: [#currentLocation: "New York City",
#mascot: "dalmatian", #lastUpdateTime: "1999/07/27 15:53:32.123456"], "@BlueTeam:
[#errorCode:[-2147216194]]]]
```

The first of these error codes indicates that the #content property of the message contains errors that you should check. The second error code indicates that the group @BlueTeam does not exist.

If you want to determine exactly what attributes have been declared for a particular group, use getAttributeNames. It will return a list of the attribute names for the group:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getAttributeNames",
"anySubject", [#group: "@RedTeam"])
```

To remove an attribute from the attribute list for a group, use deleteAttribute. The following example deletes the #mascot property from the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.deleteAttribute",
"anySubject", [#group: "@RedTeam", #attribute: #mascot])
```

Keep in mind that the attributes you define for groups will persist only while the group exists. If you want this kind of information to be stored indefinitely so that it can be recalled from one multiuser session to the next, use the database commands.

Using databases

The Multiuser Server provides extensive database functionality that allows your movies to store a wide variety of information. When you use databases, the information you store on the server will be available each time your movies connect to the server. You can use databases to do the following:

- Store information about individual users, such as their e-mail address or the type of computer they are using.
- Keep a record of a user's high score in a particular game.
- Keep track of the status of a game or other multiuser application.
- Store environment information for an application, such as map data for an adventure game.

Using data objects

The Shockwave Multiuser Server can store information for you in four different types of data objects. A data object is simply a container that holds data that you put into it. You decide what kinds of objects to use and what data to put into them. Which kind of data object you use will depend on the kind of information you want to store and what that information will be used for. You can define as many objects of each type as you need to.

The following are the four types of data objects you can use to store information:

DBUser is the type of object you should use if you are storing information about a user that is specific to the user but not to any particular movie they might use, such as their e-mail address.

DBPlayer is used to store information that is specific to both the user and a particular game they are playing, such as their high score. Since a user may use many different multiuser movies with your Multiuser Server, they might have many DBPlayer objects but just one DBUser object.

DBApplication objects are used to store information that is specific to a particular movie, such as the highest score ever achieved in a particular game.

DBApplicationData objects are appropriate for information that will be read-only, such as map data for an adventure game. Typically there will be more than one DBApplicationData object for each DBApplication. For example, in a trivia game you might use DBApplicationData objects for configuration information such as lists of trivia questions, buzzer and bell sounds, or suits the host might wear. Each of these would be stored in its own DBApplicationData object.

Storing information

Once you have decided what types of objects to use, it is time to store your data. You start by creating a new object, and then you add attributes to the object. The attributes will be whatever items of information you want to store, such as an e-mail address or a high score. As with other server functions, these operations are performed by sending commands to the server in the #recipient parameter of a sendNetMessage() and indicating which type of object you want to create or edit.

Keep in mind that you can set different combinations of attributes for different objects. You could assign user Bob a #email and a #favoriteFood but assign user Mary a #email and a #favoritePlace. What attributes you define and the information you put into them is up to you and what you want your movie to do. Object attributes can contain any Lingo value you wish to store.

You might want to start by storing user-specific information in a DBUser object. Since the DBUser object for the new user does not yet exist, you must tell the server to create it for you. This is one of several administrative functions you can perform on the server by specifying DBAdmin as the object of your command. To create a new DBUser object for a new user, use createUser in conjunction with DBAdmin:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.createUser",
"anySubject", [#userID: "Bob", #password: "MySecret", #userlevel: 20])
```

This will create a new DBUser object with a #userID property of Bob.

To add new information to the new user's DBUser object, you must name the information, which will be an attribute of the user. If you are storing e-mail addresses, you could call the attribute #email. The attribute name should be a symbol preceded by the # sign. You declare a new attribute by using declareAttribute with the DBAdmin object. Once an attribute has been declared, its name may be used with any of the database object types.

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.declareAttribute",
"anySubject", [#attribute: #email])
```

Now the attribute named #email has been declared and may be set for any of the objects you create. To set the e-mail address in the database for the user Bob that was created earlier, use setAttribute with the DBUser object:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBUser.setAttribute",
"anySubject", [#userID: "Bob", #attribute: [#email: "bobsmith@companyname.com"]])
```

The #content parameter of this sendNetMessage() is a nested property list containing the parameters for the setAttribute command. The #userID property tells the server which user's information is being edited. The #attribute property indicates which attributes are being set and is followed by a list of attributes and values. In this case, only one attribute is being set.

To create a DBApplication object and set its attributes, you start by creating a new object, using createApplication, and then declare the attributes you want to use for the object. You then place information into those attributes with setAttribute. You add attributes to a DBPlayer object by supplying both a #userID and a #application with setAttribute. See "Multiuser Scripting Elements" for detailed examples of these.

To construct DBApplicationData objects, you put all the data you want them to hold in the form of lists of attributes and then write them to the server as an administrator-level user with createApplicationData.

This Lingo statement creates a DBApplicationData object containing values that describe a room in an online casino:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.createApplicationData",
"anySubject", [#application: "Casino", #attribute: [#roomName: "BlackJack",
#dealerName: "Larry", #wallArt: "Mona Lisa", #minimumBet: 50, #music: "Classical"]])
```

Subsequent Lingo statements might create additional DBApplicationData objects with the attributes for additional rooms in the casino.

Once you have placed the objects on the server, you retrieve them for use in the movie by using getApplicationData. This command returns the list of attributes and values for the object that matches the criteria you specify for the current application. To retrieve the DBApplicationData object you previously created, you would specify the attribute #roomName and the string BlackJack as the search criteria. You supply the string in the #text property.

```
errCode = gMultiuserInstance.sendNetMessage
( "system.DBApplication.getApplicationData", "anySubject", [#application: "Casino",
#attribute: "#roomName", #text: "BlackJack"] )
```

In addition to the attributes you define, DBUser, DBPlayer, and DBApplication objects each have their own default attributes that are assigned by the server. There are no default attributes for DBApplicationData objects.

DBUser objects include these attributes:

#userID must be unique.

#password cannot be obtained with getAttribute, or set with setAttribute. To change a password, delete and recreate the DBUser object.

#lastLoginTime is changed by the server each time a user logs in and can be edited by an administrator-level user. See "Configuring the server" for more information.

#status can be written only by administrators and can be used for whatever purposes you see fit. You might use this to keep track of whether a customer's balance is due or paid in full for a site membership subscription.

#userlevel determines privileges based on the levels you define in the server's configuration file. See "Configuring the server". This attribute may be changed only by administrators.

#lastUpdateTime allows you to check for whether other users have changed an attribute since you last checked it. This is identical to how it is used with group attributes. See Working with group attributes.

DBPlayer objects include default attributes of #creationTime and #lastUpdateTime. The #creationTime attribute indicates when the object was created on the server and will usually correspond to when the user began participating in a particular movie.

DBApplication objects include default attributes of #userID, #description, and #lastUpdateTime. The #userID attribute defaults to the name of the movie that created the object, which you supplied in the connectToNetServer() command. The #description attribute contains whatever description you choose to supply when you create the DBApplication object.

A database scenario

Suppose you want to create an online casino in which users will play various games, such as poker, blackjack, and roulette. Each game could be a separate movie you create and could contain several different rooms to play in. You could use each of the types of database objects to store information about users, players, games, and the casino environment.

You could create DBUser objects for each new person who logs in and wants to enter the casino. The attributes you store for each user might include their e-mail address, preferred games, and so on. Your DBPlayer objects might include a player's current winnings for a particular game.

You could use DBApplication objects to store the name of the user who has won the most money in each game. You might also have an attribute in each of these DBApplication objects that is a list of the DBApplicationData objects you've created for different rooms in each game. You could then use this list to retrieve the data for a particular room in a particular game. The DBApplication object for the blackjack movie might contain an attribute called #roomList with room names in a linear list like this one:

```
["Art Deco", "Western Saloon", "Contemporary", "Egyptian"]
```

You could then use getApplicationData to retrieve the list of attributes for each room. The DBApplicationData object for one room might look like this:

```
[#roomName: "Art Deco", #dealerName: "Larry", #wallArt: "Mona Lisa", #minimumBet:
50, #music: "Classical"]
```

By creating DBApplicationData objects for each room, you can design your movies to display certain graphics and sounds based on which room object is chosen by the user.

Controlling user access to the server

You can control who connects to the server by using DBUser objects. You can choose to allow anyone to connect to the server, or you can limit access to users who already have a DBUser object. To limit user access you would use an administrative movie to create new users' DBUser objects before allowing them to connect on their own. You can also choose to let all users connect but restrict certain privileges to only those users with DBUser objects. You do this by giving users with DBUser objects a higher value for their #userlevel attribute and letting other users default to a lower value. See Configuring the server for more information.

Optimizing multiuser movies

When designing a multiuser experience, you should ensure that your movie responds to the information coming in from other movies in a prompt manner. You will also want your movie to send its own outgoing messages as soon as they are needed. The following guidelines will help you to create movies that are as fast and responsive as possible.

Minimize message frequency

Design your movies to send only information that is absolutely necessary. For example, if you are tracking the position of a player's sprite, design you movie to send position information only when the sprite's position changes, rather than sending it at some regular interval, regardless of whether the sprite has moved or not.

For some applications, it will make sense to collect a set of very small messages and send them together as one larger message. For example, you might collect several points of a whiteboard brush stroke and send them together in a list. For chat movies, send the entire chat message after the user presses return instead of sending every character individually.

Prioritize receiving over sending

In most cases, you will want your movies to receive incoming messages before sending messages, since the contents of incoming messages may affect the information you want to send.

One way to accomplish this is to increase the framerate of your movie. The Multiuser Xtra checks for incoming messages during idle periods between frames, so higher framerates will cause the Xtra to check for messages more often. If you don't want to increase the framerate, you can set the `idleHandlerPeriod` to zero in Lingo. This will maximize the frequency of idle events and allow the Xtra to check for messages during frames as well as between them.

You can tell the Xtra to check for messages at any moment you wish by using the `checkNetMessages()` function. Or you can find out how many messages are waiting by using `getNumberWaitingNetMessages()`. This allows you to specify a number of messages to retrieve as a parameter of `checkNetMessages()` so any backlog of messages can be handled all at once.

Minimize stage updates

In general, drawing sprites on stage is much more time-consuming than sending, receiving, or processing messages. Therefore, you won't necessarily want to update the screen every time you receive a message. For example, if every player in a game sends their status or position, you may want to collect that information and then update the sprites on stage all at once.

Send targeted messages

If only one user needs the information in a particular message, send it to the individual user rather than to a whole group.

Use setNetMessageHandler()

By assigning different subjects to different types of messages you are sending, you can use setNetMessageHandler() to process incoming messages depending on the subject, the sender, or both. The Xtra's message-dispatching routines are faster than similar code written in Lingo.

To change which message callback handler is triggered by a certain type of message, use setNetMessageHandler() to remove the reference to the first handler by using a zero in place of the handler symbol:

```
errCode = gMultiuserInstance.setNetMessageHandler(0, script "Connection Script", "Chat
Text", "Guest Speaker")
```

You can then declare a new handler for the same message criteria by calling setNetMessageHandler() again and specifying the new handler symbol:

```
errCode = gMultiuserInstance.setNetMessageHandler(#newMessageHandler, script
"Connection Script", "Chat Text", "Guest Speaker")
```

2

CHAPTER 2 The Server Application

System requirements

The following are the minimum system configurations for running the Shockwave Multiuser Server:

- Windows NT (recommended), or Windows 98
- Macintosh System 8.1 or later, PowerPC
- 6 MB available RAM
- An active TCP (Transmission Control Protocol) network connection, configured to allow incoming connections

Installing the server

To install the server, double-click the installer icon. You can install the server into any location you wish. The installer will create a folder that contains the following items:

- The server application
- A Multiuser.cfg file, which contains parameters you can set in order to control certain aspects of the server's behavior
- A MultiuserLicense.cfg file, containing your Multiuser Server serial number
- A ReadMe file containing last-minute information about the server application
- An Xtras folder containing server Xtras that extend the server's functionality
- Two DLLs, C4dll.dll and Msvcrt.dll (Windows only)

Running the server

Once you have installed the server, launch it by double-clicking its application icon. At startup, the server will read through the Multiuser.cfg file, load the two default server Xtras, and then allocate the proper number of connections depending on which server license you have purchased. When the connections have been allocated, the server is ready to accept connections and process messages from Director movies.

Viewing server information

You can view details about the server's activity in two ways. The View menu lets you see messages in the Server window each time certain events happen, such as when users log in or out of the server. This is useful during testing of movies or any time you want to see a complete log of activity. The Status menu lets you see information on demand about the total number of users on the server, movies connected, databases being used, groups created, and more.

You can see how much time the server is taking to process each incoming message by choosing the Server Response Time menu item from the View menu. A check mark next to the menu item indicates that it is active. It will display a message every ten seconds that shows the number of messages processed and the average time in milliseconds used for each message. You can adjust the time interval from ten seconds to another number by editing the Multiuser.cfg file, which will be discussed later.

To see a message each time a user logs in or out of the server, choose the Users Log On and Off menu item. To see a message each time the first instance of a new movie such as Poker connects to the server, choose the Movie Creation and Deletion menu item. This menu item also displays a message when the last instance of a particular movie disconnects from the server. By using the other menu items, you can choose to see messages each time a connected user joins or leaves a group or when new groups are created or deleted.

To view a display of general status information, choose the Server item from the Status menu. It will display the following information:

- Server IP address: 123.456.78.90. This is the address you specify in the Multiuser.cfg file. See "Configuring the server".
- Server port: 1626. This is the communications port number assigned to the Shockwave Multiuser Server.
- Number of connections: 47 available, 3 in use. In this example, three users are connected to the server. The number of connections is based on the number permitted by the server license key.

- Connections waiting to be recycled: 0. These are connections that have logged out or timed out but have not yet been reset by the server to be able to accept new connections. This number will usually be zero except on very active servers.
- Number of connections awaiting logon: 0. These are connection requests that the server has received but not yet processed.
- Number of movies: 3. This is the number of different movies connected to the server. These might be Chess, Pinball, and TechChat, for example, for a total of three movies.

To display a list of movies currently connected to the server as well as the number of users, groups, and databases used by each movie, choose Movies from the Status menu.

To see the total number of users currently connected to each movie on the server, choose Users from the Status menu. This will also display each user's name, user level, and IP address.

To see a list of the groups for each movie on the server and the members in each group, choose Groups from the Status menu.

To see a list of the databases used by each movie, choose Databases from the Status menu.

Configuring the server

You can change many aspects of how the server operates by editing the Multiuser.cfg file. You can specify separate settings for different movies by creating additional CFG files. By editing parameters in any of these CFG files, you can do the following:

- Change the amount of memory available for handling messages.
- Limit the number of users that can connect to the server, or to a particular movie.
- Specify which types of users are allowed to connect to the server.
- Specify which movies may connect to the server.
- Set the user levels required to execute each server command.

Using the Multiuser.cfg file

There are many other settings that can also be changed by editing the Multiuser.cfg file. Examples of each of these can be found in the Multiuser.cfg file that came with your server. Comments can be added to any of the configuration files by preceding each comment line with a # sign. You can also add a # sign to the beginning of an optional parameter line to prevent it from being read by the server. For parameters that can take more than one value, place a space and a \ after the first value and place the next value on the next line. For changes to the file to take effect, the server must be restarted.

When you edit the Multiuser.cfg, keep a copy of the original file provided with the server installation.

The following server settings can be edited:

Echo allows you to add text that will be displayed in the Server window when the server reads the Multiuser.cfg file at startup.

ServerPort specifies the communications port used by the server. The Shockwave Multiuser Server is assigned port 1626 so that it will not conflict with other server applications running on the same computer. You may change this if you find it necessary to do so.

ServerIPAddress is an optional parameter that allows you to specify an IP address if your server computer uses more than one. Most computers use only one IP address. You can also use this parameter to make the server listen for messages on multiple port numbers on a single IP address.

MaxMessageSize sets the amount of memory to use for the incoming and outgoing message buffers. The total memory used will be twice the number you specify. The value must be in bytes and should be a multiple of 1024 (1 kilobyte).

ConnectionLimit is an optional parameter that specifies the maximum number of users that can connect to the server. It will default to the number indicated by the license you purchased. You can use this parameter to limit the connections to a smaller number.

EncryptionKey is an optional parameter that lets you specify a string to use for encrypting user name and password information when users connect to the server. If you use this parameter, you must include the same exact string with the `connectToNetServer` command.

UserLevel lets you specify required user levels for each of the database commands and server commands. You can choose any number between 0 and 100 for each command you want to control access to. By using different numbers for different commands, you can create custom privileges for users of different levels. See the original Multiuser.cfg file that came with your server for the default values for each command.

MovieCFGPath is an optional parameter that lets you specify the directory where movie-specific configuration files are located.

LogFileName is an optional parameter that lets you specify the name and location of the text file the server generates with its status messages. This file contains all the information displayed in the server's application window and is useful for looking back at server activity and debugging movies.

AllowMovies is an optional parameter that lets you specify movies that should be allowed to connect to the server. If this parameter is not used, any movie may connect.

IdleTimeOut is an optional parameter that lets you specify how many seconds to let movies stay connected to the server if they are not sending any messages. Movies that remain idle for longer than the time you specify will be disconnected by the server. If omitted, this parameter defaults to 1 hour.

ScanTimeReportInterval lets you set the number of seconds between each Server Response Time message in the server's application window. This takes effect only when the Server Response Time item in the server's View menu is turned on.

You can change the default settings for which types of messages are displayed in the server's application window by editing the following parameters. These parameters change which items in the server's View menu are turned on and off at startup. A value of 1 turns the item on, and a value of 0 turns the item off.

ShowLogonMessages corresponds to the Users Log On and Off item in the View menu.

ShowCreateMovieMessages corresponds to the Movie Creation and Deletion item in the View menu.

ShowScantimeMessages corresponds to the Server Response Time item in the View menu.

ShowCreateGroupMessages corresponds to the Group Creation and Deletion item in the View menu.

ShowJoinGroupMessages corresponds to the Users Join and Leave Groups item in the View menu.

The following parameters indicate which server Xtras you want the server to load at startup and which commands each Xtra provides:

ServerExtensionXtras lets you specify the names of Server Xtras to load at startup. These Xtras need to be located in the Xtras folder at the same directory level as the server.

XtraConfigCommands lets you specify a list of commands for a specific Xtra. This parameter is followed by a list of commands.

XtraCommand lets you specify each command that a particular Xtra uses. This makes the server aware of what strings may be sent as commands for the Xtra. The default Multiuser.cfg file that comes with the server has a series of commands declared for the ObjectDB Xtra, which controls the use of database objects with the server and determines whether users without a preexisting `DBUser` object can log on to the server. See the Multiuser.cfg file that was installed with your server for details on these logon controls.

Using the MultiuserCommon.cfg file

If you have multiple servers running on multiple computers, you can use a MultiuserCommon.cfg file to make it easy to apply the same settings to all your servers. This file will be read by each of the servers when they start up, before they read the Multiuser.cfg file. By using both of these files, you can give your servers the same values for some settings and different values for others.

Follow these steps to configure multiple servers using both a MultiuserCommon.cfg and a Multiuser.cfg file:

1. Install the server on each computer.
2. Create a single MultiuserCommon.cfg file with the settings you want to be common to all the servers.
3. Place the MultiuserCommon.cfg file in a location of your choice.
4. Create an alias or shortcut to the file and place a copy of it next to each copy of the server.
5. Edit the Multiuser.cfg file for each server with the individual settings you choose. If a parameter appears in both the MultiuserCommon.cfg file and a specific Multiuser.cfg file, the setting in the Multiuser.cfg file will have precedence.

Using movie configuration files

Each different movie that connects to the server may also have its own configuration file. You can use the same parameters in this configuration file that you use in the server's Multiuser.cfg file to give the movie its own specific settings for those parameters. The server reads the movie's configuration file when the first instance of the movie connects to the server. Give the movie's configuration file the same name that the movie uses to connect to the server, such as Whiteboard.cfg, and place it in the location you specify in the Multiuser.cfg file.

In addition to the parameters used in the server's Multiuser.cfg file, you can use the following parameters to control specific movie actions:

NotifyDisconnect lets you specify a group name for the server to send a message to each time a user of the movie disconnects from the server.

GroupSizeLimits lets you specify a maximum number of users to allow in each group you specify for the movie. You can specify a different limit for each group. When this parameter is used, the groups will exist on the server even when they have no members.

Administering the server

In order to run the server successfully in a network environment, it is important that you or your network administrator understand how the server interacts with the rest of the network and what features of your network the server depends on.

If your server is installed behind a firewall and you want to allow users outside the firewall to connect, you must make sure your firewall is set to allow incoming traffic on the port number that the server uses. By default, this port is 1626. You can choose another port number and configure the server's Multiuser.cfg file to use it, but you must be sure no other application on your server computer will try to use the same port number as the server. If your network uses a proxy server, it must also be configured to allow incoming traffic on the port that the Multiuser Server is using.

To ensure that your server will always be available to users in the event of a hardware or software failure, you can implement a redundant server installation by using two servers on two computers and connecting them with a third-party load-balancing solution. These include products such as Central Dispatch™ from Resonate and Local Director™, a hardware tool from Cisco Systems.

You can set up the server to start automatically after a hardware restart by adding a server alias to the Startup Items folder in the Macintosh System Folder or a server shortcut to the Startup folder in Windows. This will cause the server to restart any time there is an unplanned hardware restart, such as those caused by power failures or other accidents.

Extensibility

Version 2.0 of the Multiuser Server uses Xtras, separate files that contain software code for specific server functions. You can choose not to load certain parts of the server's functionality by removing Xtras from the server's Xtras folder. For example, you can remove the Database Xtra if you are not using databases. You can also write your own Xtras to add custom functions to the server if you are a competent C or C++ programmer. A separate Xtra Developers Kit is available from Macromedia's Web site for those who wish to learn more about writing Xtras for the server.

Troubleshooting

Here are three common issues you might encounter when configuring and using the server:

Connection attempts are blocked. This usually means that the user is behind a firewall that is not configured to allow outgoing connections on the port that Shockwave uses. Contact your network administrator to adjust the firewall settings to allow outgoing TCP/IP connections on port 1626. Client movies running behind firewalls need to be able to make outgoing connections through the firewall in order to communicate with the server. Clients movies acting as peer hosts will need the firewall to allow incoming connections.

Large messages are not getting through the server. This usually indicates that the client or server's message buffers have not been set large enough to handle the size of the messages you are sending. You can change this setting by editing the `MaxMessageSize` parameter in the Multiuser.cfg file. You may also need to use `setNetBufferLimits` to adjust the Xtra's buffer sizes in your movie's Lingo scripts.

Proxy servers can interfere with the Shockwave Multiuser Server or multiuser movies that act as peer hosts, since the host uses the clients' IP addresses for message routing purposes. Director movies or servers running behind a proxy server may have difficulty making connections and sending messages because the proxy server may mask the real IP addresses of the movies.

CHAPTER 3
Multiuser Scripting Elements

There are several categories of scripting elements for multiuser applications:

- Multiuser Lingo functions are provided by the Multiuser Xtra. They control the Director movie itself as it functions in a multiuser application, connecting to remote hosts, sending and receiving messages, checking error codes, monitoring the state of the movie, and so on.
- Server commands are instructions to the Shockwave Multiuser Server. Send them as messages with the `sendNetMessage()` Lingo function.
- Group commands are server commands used to manage groups of users and group attributes.
- Server database commands are server commands for interacting with database files on a remote host.

Lingo commands

These commands are used for establishing and managing server connections:

```
setNetMessageHandler
setNetBufferLimits
connectToNetServer
sendNetMessage
getNetMessage
getNumberWaitingNetMessages
checkNetMessages
getNetErrorString
getAddress
getNetOutgoingBytes
```

Peer-to-peer connections

These commands are used to establish and manage peer-to-peer connections:

setNetMessageHandler
waitForNetConnection
connectToNetServer
getPeerConnectionList
breakConnection
getNetAddressCookie

Server commands

The following commands control the Shockwave Multiuser Server. Send the commands as messages with sendNetMessage(), using a recipient containing the syntax System.object.command, a subject of your choosing, and any required parameters in the message contents.

getVersion
getTime
getMovieCount
getMovies
getUserCount
delete
disable
enable

Group commands

These server commands control groups and group attributes:

join
leave
getGroupCount
getGroups
getUserCount
getUsers
setAttribute
getAttribute
getAttributeNames
delete
deleteAttribute
disable
enable
createUniqueName

New database commands

Version 2.0 of the server supports new database types that are managed entirely by the server and can be manipulated with Lingo. The following commands control these databases and use a "server.*databaseObjectType*.*commandName*" syntax.

The commands below are listed according to the types of objects they can be applied to:

DBAdmin
- createUser
- deleteUser
- createApplication
- deleteApplication
- createApplicationData
- deleteApplicationData
- declareAttribute
- deleteAttribute

DBUser
- setAttribute
- getAttribute
- getAttributeNames
- deleteAttribute

DBPlayer
- setAttribute
- getAttribute
- getAttributeNames
- deleteAttribute

DBApplication
- setAttribute
- getAttribute
- deleteAttribute
- getAttributeNames
- getApplicationData

Group
- getAttribute
- setAttribute
- getAttributeNames

Multiuser Lingo functions

The Multiuser Xtra provides the following functions for controlling movies in multiuser applications.

breakConnection

Syntax gMultiuserInstance.breakConnection(*userIDString*)

Description Lingo command; breaks a peer connection accepted with waitForNetconnection().

Example This statement has the host movie break a connection with the user logged in as Spencer:

```
errCode = gMultiuserInstance.breakConnection("Spencer")
```

See also getNetErrorString, waitForNetConnection

checkNetMessages

Syntax gMultiuserInstance.checkNetMessages({*numberOfMessages*})

Description Lingo command; forces a check for any incoming messages and calls the callback handlers specified by setNetMessageHandler(). It is necessary only when the movie itself does not allow the normal idle time in which the Xtra would check for incoming messages automatically. The optional parameter defaults to 1 and specifies the maximum number of messages to process. Setting this number to a high value can cause performance problems since this function does not return until the waiting messages have been processed.

This function is useful in movies that sit in one frame with a go to the frame loop, for example. It should not be called from a message handler specified by setNetMessageHandler(), or by any handler called by a message handler (including beginSprite, enterFrame, etc.) that can be executed as the result of a go to frame n statement.

Example This statement tells the Multiuser Xtra to check for incoming messages, with a maximum of four messages:

```
errCode = gMultiuserInstance.checkNetMessages(4)
```

See also setNetMessageHandler, idleHandlerPeriod

connectToNetServer

Syntax gMultiuserInstance.connectToNetServer(*userNameString*, *passwordString*, *serverIDString*, *portNumber*, *movieIDString* {, *mode*, *encryptionKey*})

Description Lingo command; starts a connection to the server and returns an error code indicating whether the connection was initiated.

userNameString and **passwordString** provide account information for logging on to the host system. User names must not contain any control characters, or the symbols @ or #.

serverIDString represents the Internet address of the server, such as gameServer.macromedia.com, or 123.45.67.1.

portNumber is an integer specifying the connection port to be used. (By default, the Shockwave Multiuser Server uses port 1626.) This must be the same port specified in the server configuration file.

movieIDString represents the multiuser application name for this connection, such as GameChat or PingPong.

mode is an optional parameter. This parameter must be an integer; the default value is zero. If set to 1, it opens the connection as a text-based connection for communicating with IRC and SMTP servers. A movie running in a browser or in ShockMachine cannot make a text server connection unless the text server is in the same domain name as the Web server the movie is served from. Projectors running with the safePlayer = TRUE cannot make text connections, unless the movie being played is a remote DCR file on the same server as the text server. If this command is used to connect to POP, SMTP, or IRC chat servers, none of the server command functionality is available. The command supports only simple text messaging.

encryptionKey is another optional parameter. This string is used to encrypt logon information sent to the server and must match the encryption key specified in the Multiuser.cfg file for the server.

Use this command to connect to multiuser servers. Finding and connecting to the server can take some time. This function first returns an error message immediately from the Xtra. If the function's parameters are valid, the returned error code is 0. The server sends a subsequent message and calls the message handler when the connection is actually made.

Before using the connectToNetServer command, you should first set up a handler to receive the message that connectToNetServer returns. To set up the connection, create an instance of the Xtra, define the message handler using a setNetMessageHandler statement, and then connect to the server.

The message returned from getNetMessage is a property list with the following information:

#errorCode	Resulting error code: 0 if there is no error
#senderID	System
#subject	ConnectToNetServer
#content	Empty string
#timeStamp	Time in milliseconds on the server

You disconnect from the server by setting the variable containing the Multiuser Xtra instance to zero.

Example This statement is a typical call that initiates a connection that is not encrypted:

```
errCode = gMultiuserInstance.connectToNetServer( "Fred", "secret",
"serverName.company.com", 1626, "ExcitingGame")
```

See also waitForNetConnection, setNetMessageHandler, getNetMessage

getNetAddressCookie

Syntax gMultiuserInstance.getNetAddressCookie({*encryptFlag*})

Description Lingo command; returns a network address cookie for the current machine. By default, the returned string is an encrypted string containing the local computer's IP address.

The getNetAddressCookie() function lets Lingo work with the local IP address without knowing the actual address. This is a security precaution to prevent movies being run by an end user behind a firewall from determining the address of the end user's computer.

The cookie's contents are "MacromediaSecretIPAddressCookie <encrypted IP address>". This value can be passed over the network to another computer, which can then use it as the server address for connectToNetServer.

This is a typical scenario when you meet another user or users on a server and want to create a peer-to-peer connection without revealing your IP address. You can send an encrypted address to the other users, and then wait for the other users by using waitForNetConnection.

If the optional *encryptFlag* is set to zero, the returned string looks like a regular IP address, such as 123.45.67.1. This unencrypted address information is not available when movies are playing in Shockwave, or if the safePlayer movie property is set to True.

See also connectToNetServer, waitForNetConnection, getUserIPAddress

getNetErrorString

Syntax gMultiuserInstance.getNetErrorString(*errorCodeNumber*)

Description Lingo command; returns a string explaining the error code that is provided in place of *errorCodeNumber*. If the error code is invalid, this command returns a string representing an unknown error. Error codes are negative integers; 0 indicates that no error occurred.

Possible error codes and their strings are as follows:

Error code	Translated error string
0	No error
-2147216223	Unknown error
-2147216222	Invalid movie ID
-2147216221	Invalid user ID
-2147216220	Invalid password
-2147216219	Incoming data has been lost
-2147216218	Invalid server name
-2147216217	Server or movie is full; no connections are available
-2147216216	Bad parameter
-2147216215	No socket manager present
-2147216214	There is no current connection
-2147216213	No waiting message
-2147216212	Bad connection ID
-2147216211	Wrong number of parameters
-2147216210	Unknown internal error
-2147216209	Connection was refused
-2147216208	Message buffer is full
-2147216207	Invalid message format
-2147216206	Invalid message length
-2147216205	Message is missing
-2147216204	Server initialization failed
-2147216203	Server send failed

Error code	Translated error string
-2147216202	Server close failed
-2147216201	Connection is a duplicate
-2147216200	Invalid number of message recipients
-2147216199	Invalid message recipient
-2147216198	Invalid message
-2147216197	Server internal error
-2147216196	Error joining group
-2147216195	Error leaving group
-2147216194	Invalid group name
-2147216193	Invalid server command
-2147216192	Not permitted with this user level
-2147216191	Error with database
-2147216190	Invalid server initialization file
-2147216189	Error writing database
-2147216188	Error reading database
-2147216187	User ID not found in database
-2147216186	Error adding new user
-2147216185	Database is locked
-2147216184	Data record is not unique
-2147216183	No current record
-2147216182	Record does not exist
-2147216181	Moved past beginning or end of database
-2147216180	Data not found
-2147216179	No current tag selected
-2147216178	No current database
-2147216177	Can't find configuration file
-2147216176	Current database record is not locked
-2147216175	Operation not allowed at current security level

Error code	Translated error string
-2147216174	The requested data or object was not found
-2147216173	Message content contains error information
-2147216172	Data concurrency error

getNetMessage

Syntax gMultiuserInstance.getNetMessage

Description Lingo command; returns the oldest waiting network message in the Xtra's message queue. The message is then deleted from the Xtra's memory space. These messages queue up internally for each connection, waiting for this function to be called. If no messages are waiting, this function returns an empty message. This function does not check the connection to the server; it only returns messages that are stored in the Xtra's message queue. This function should be called from a network message handler.

If the connection has been opened as a text connection, the sender is set to System and the subject is String. The content is a string containing the data from the server. The Xtra returns all the data available, which might not terminate with the end of a text line. The Xtra includes and does not alter end-of-line characters sent by the server (CR or CRLF). If the server sends a zero byte, the Xtra drops the byte and does not pass it to Lingo.

This function's results are in the form of a property list with the following information. Symbols are used to access the list, retrieving an error code, the sender's ID, the subject, and the message contents.

#errorCode	Resulting error code: 0 if there is no error
#recipients	Users or groups the message was sent to
#senderID	String such as "Fred"
#subject	String subject of the message
#content	List of values, depending on the subject
#timeStamp	Server's time code stamp at the point of handling the message

Example These statements retrieve a message from the queue:

```
netMsg = gMultiuserInstance.getNetMessage
errCode = netMsg.errorCode
if (errCode = 0) then
    senderID = netMsg.senderID
    subject = netMsg.subject
    messageList = netMsg.content
    --handle the message
else
    --do error processing
    alert gMultiuserInstance.getNetErrorString(errCode)
end if
```

The contents of the retrieved message would look like this:

```
[#errorCode: 0, #recipients: ["@AllUsers"], #senderID: "Fred", #subject: "ExampleSubject",
#content: "ExampleContent", #timeStamp: 36437632]
```

See also getNetErrorString

getNetOutgoingBytes

Syntax gMultiuserInstance.getNetOutgoingBytes({*userIDString*})

Description Lingo command; returns the number of bytes currently in the outgoing message buffer, which is data waiting to be sent. This command is useful for determining if all data has been sent, or if the client's outgoing buffer has room for large data chunks the user may want to send.

The total number of outgoing bytes possible is set by the maxMessageSize parameter of the setNetBufferLimits command. The default value of maxMessageSize is 16K.

The optional *userIDString* parameter is used to specify a particular user's buffer when hosting peer connections.

Example The following code determines how much data is currently in the buffer for the current movie, and sends a new message containing a large chunk of data (such as a cast member image) only if there is less than 500K in the buffer. The statements allow for sending pictures of up to 100K:

```
gMultiuserInstance.setNetBufferLimits(16 * 1024, 600 * 1024, 100)
totalWaiting = gMultiuserInstance.getNetOutgoingBytes
if totalWaiting < (500 * 1024) then
    gMultiuserInstance.sendNetMessage("@AllUsers", "New image", member("Latest
snapshot").picture)
end if
```

See also sendNetMessage, setNetBufferLimits

getNumberWaitingNetMessages

Syntax gMultiuserInstance.getNumberWaitingNetMessages

Description Lingo command; returns an integer representing the number of messages that have arrived in the Xtras queue and have not been read yet. This can be useful for determining whether you should call checkNetMessages.

Example This statement gets the number of messages waiting to be processed:

```
numMessages = gMultiuserInstance.getNumberWaitingNetMessages
```

See also checkNetMessages

getPeerConnectionList

Syntax gMultiuserInstance.getPeerConnectionList

Description Lingo command; returns a list of all the peer users connected to the host movie. An outgoing connection made with the connectToNetServer() command is not included in this list; only peer connections accepted with waitforNetConnection() are included.

The return value is a list containing the userIDs, or an empty list in the case of an error or no connections.

Example This statement obtains the list of users connected to the host movie:

```
userList = gMultiuserInstance.getPeerConnectionList
```

See also connectToNetServer, waitForNetConnection

sendNetMessage

Syntax gMultiuserInstance.sendNetMessage(*string/ListRecipient*, *stringSubject*, *string/ListMessage*)

gMultiuserInstance.sendNetMessage(*propertyListMessage*)

Description Lingo command; sends a message to one or more recipients in the current movie or a specified movie on the same server. The recipient may be a specific user, such as Sarah, a group, such as @TicTacToePlayers, or a list of strings containing individual user names or groups. To send a message to all users connected to the same movie, set the recipient to the default group @AllUsers. To send a message to a user in a different movie on the server, add the movie name to the end of the user name with the @ symbol, such as Jane@TechChat.

The message recipient can be a single string, or a list of strings in the case of multiple recipients. The subject must be a valid string. The contents may be a single Lingo value, such as a string, integer, float, point, rect, property, list, property list, color, or date, or data such as the media of member or the picture of member.

If you are sending large data, such as a picture or the media of a cast member, make sure the buffer limits are large enough for both the client and the server. See setNetBufferLimits and "Configuring the server".

The second possible syntax accepts a property list containing all the information for the message. The property list contains values for the symbols *#errorCode*, *#recipient*, *#subject*, and *#content.* (The *#errorCode* and *#content* values are optional.) This format is similar to the format for messages received by getNetMessage().

If you send a message to a group of which you are a member, you will also receive the message. If the server has problems sending the message to any recipients in a group or list, no error message is returned.

To send requests to the server, a special syntax is used for the recipient, containing System, an object name, and the server command name. The server responds with a message that has the same subject with an error code of zero if the operation is successful.

If the connection is opened as a text connection, the recipient and subject are ignored. The message contents should all be simple Lingo values that will be converted to strings. Any carriage-return-line-feed (CRLF) combinations, such as those required by a POP server, must be explicitly added to the text sent by the Director movie.

Example These statements are some possible variations that can be used to send messages:

```
errCode = gMultiuserInstance.sendNetMessage("@AllUsers", "ChatMsg", gChatText)
errCode = gMultiuserInstance.sendNetMessage("BlackBeard", "MoveShipTo", point(shipX,
shipY))
errCode = gMultiuserInstance.sendNetMessage("@groupRedTeam", "Help", "I am lost")
errCode = gMultiuserInstance.sendNetMessage(["Bill", "Joe", "Sue"] "JoinMe", "")
```

This statement provides its parameters in the form of a property list:

```
errCode = gMultiuserInstance.sendNetMessage([#recipients: "Bob", #subject: "Hello",
#content: "How are you?"])
```

This statement sends a message to user Jane in a separate movie called TechChat:

```
errCode = gMultiuserInstance.sendNetMessage("Jane@TechChat", "ChatMsg", "What are you
talking about?")
```

This statement sends the command getUsers to the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getUsers", "anySubject",
"@RedTeam")
```

This statement sends the command getGroupCount to the server as a request for information about a separate movie on the server called AdventureGame:

```
errCode = gMultiuserInstance.sendNetMessage
("system.movie.getGroupCount@AdventureGame", "anySubject")
```

This statement sends the RETR command to a mail server to retrieve one piece of mail:

```
errCode = gMultiuserInstance.sendNetMessage("system", "anySubject", "RETR 1")
```

setNetBufferLimits

Syntax gMultiuserInstance.setNetBufferLimits(*tcpipReadSize*, *maxMessageSize*, *maxIncomingUnreadMessages*)

Description Lingo command; sets the size of internal buffers and sets limits on the number of messages that the Xtra can queue in memory. The command is not normally needed, but it can be used to fine-tune memory management. If a movie sends or receives particularly large messages, such as picture data, these values should be set to accommodate them. Specify all the values in bytes.

In order to take effect, this command must be issued immediately after a Multiuser Xtra instance is created. It will not take effect once a server connection has been established.

tcpipReadSize controls the maximum amount of data read each time the Xtra takes data from the low-level TCP/IP data stream. This may be altered to tweak performance, particularly when receiving large messages. The default is 16K bytes.

maxMessageSize controls the buffers used to store parts of messages sent and received from another system. The default value is 16K bytes. This must be larger than the largest message sent or received.

maxIncomingUnreadMsgs sets a limit on the number of unread incoming messages that can be accumulated. The default is 100.

Example This statement sets the maxMessageSize to 350K bytes to accommodate sending image data back and forth:

```
errCode = gMultiuserInstance.setNetBufferLimits(16 * 1024, 350 * 1024, 100)
```

setNetMessageHandler

Syntax gMultiuserInstance.setNetMessageHandler(*#handlerSymbol*, *handlerObject*, {*subject*, {*sender*}})

Description Lingo command; sets the callback handler that is invoked when messages arrive. This should be set before connecting via connectToNetServer() in order to capture the response message.

You can set as many callbacks as you need. It is much more efficient to have multiple callbacks that react to specific types of incoming messages than to have one callback that filters messages and then calls another handler.

This ability is key for creating flexible and effective experiences. You might have a scenario that uses avatar objects to represent other users, and set a callback to each specific object. In that way, any message related to that user is routed automatically when dealing with the object. The avatar object itself may even have multiple callbacks for messages with different subjects.

#handlerSymbol is a symbol that represents the handler.

handlerObject represents the object where the handler is located. This may be a behavior instance, a parent script, or any Lingo value. If it is a script or behavior instance, the handler associated with it will be called. If it is a Lingo value, the handler must be in a global script and the Lingo value will be passed as the first parameter to that script.

subject and **sender** are optional parameters that route messages with particular subjects, senders, or both to a handler. This lets avatar scripts receive all the messages from a sender. To route all messages from one sender to a handler, set the subject to zero and specify the sender's ID.

To disable message notification, set the handler information to zero.

Example This statement sets the handler `on MyNetMessageHandler` located in the script cast member CallBackScript as the generic message callback handler (no specific subject or sender is specified):

```
errCode = gMultiuserInstance.setNetMessageHandler(#MyNetMessageHandler, script
"CallBackScript")
```

This statement sets the handler `on BobHandler` in the script cast member Callbacks as the message callback handler for messages with any subject received from user Bob:

```
errCode = gMultiuserInstance.setNetMessageHandler(#BobHandler, script "Callbacks", 0,
"Bob")
```

This statement sets the handler `on RedCarMsgHandler` in the script object `me` as the message callback handler for messages with a subject of setCarPosMsg and a sender of Fred:

```
errCode = gMultiuserInstance.setNetMessageHandler(#RedCarMsgHandler, me,
"setCarPosMsg", "Fred")
```

This statement disables the handler declared above by specifying zero in place of the handler symbol:

```
errCode = gMultiuserInstance.setNetMessageHandler(0, me, "setCarPosMsg", "Fred")
```

See also connectToNetServer

waitForNetConnection

Syntax gMultiuserInstance.waitForNetConnection(*userIDString*, *portNumber*{, *maxNumberOfConnections*", {*encryptionKeyString*}})

Description Lingo command; starts listening for incoming peer-to-peer connections from other computers.

userIDString represents the logon name of the user acting as the host and waiting for connections.

portNumber represents the Internet port the connecting system will contact. Multiuser servers should use port 1626 by default. Generic inbound TCP connection requests are not supported.

maxNumberOfConnections represents an optional parameter for the maximum number of possible connections to the host. Up to 16 peer connections are allowed.

encryptionKeyString is an optional parameter that supplies a key string to decode logon information from other systems. If this parameter is used, the other systems must use the exact same encryption key string when they connect using connectToNetServer.

The connecting computer will use connectToNetServer and appear to be connecting to a normal multiuser server, but it will actually be connecting to a peer computer that has issued the waitForNetConnection command. In connections of this type, no server commands are available. The computer that calls waitForNetConnection must do so before a peer calls connectToNetServer.

Because waiting for an incoming connection can take a long time, this function returns an error code immediately. An error code of zero indicates that the Xtra has begun listening successfully. A message is sent back from the Xtra (and the message handler called) when the incoming connection is actually established. You must set a handler with setNetMessageHandler for this before calling waitForNetConnection. The handler called for waitForNetConnection should return TRUE if the host movie wants to accept the connection, or FALSE if it is to be rejected. The returned message is a list that contains the following items:

#errorCode	Resulting error code: 0 if there is no error
#senderID	System
#subject	WaitForNetConnection
#content	Remote user information in a property list containing #userID, #password, and #movieID

Up to 16 peer-to-peer connections can be established with each Xtra instance using waitForNetConnection. Multiple calls to waitForNetConnection cannot be made using the same port number, because the Xtra instances cannot all wait on the same port number.

After the function is called, you cannot turn off waiting for additional connections, except by deleting the Xtra instance. You can specify a limit to the number of connections in the call to waitForNetConnection, or use the handler to return TRUE or FALSE to allow or reject the incoming connection attempts.

Do not make outgoing server connections using an Xtra instance that has called waitForNetConnection.

Example These statements show a range of typical calls to set the movie to receive connections:

```
errCode = gMultiuserInstance.waitForNetConnection("Fred", 1626)
errCode = gMultiuserInstance.waitForNetConnection("Mark", 1626, 16, "Queen3ToRook2")
errCode = gMultiuserInstance.waitForNetConnection("Joe", 1626, 2, "RogerWilco")
```

See also setNetMessageHandler

Server, group, and database commands

createApplication

Syntax system.DBAdmin.createApplication [#application: "*ApplicationName*", #description: "*DescriptionString*"]

Description Adds a new DBApplication database object to the server. Application names may contain up to 100 characters. The description string may contain up to 255 characters.

Example This statement creates a new DBApplication object for the movie called Checkers with a description of Two-player Checkers game:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.createApplication",
"anySubject", [#application: "Checkers", #description: "Two-player Checkers game"])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBAdmin.createApplication",
#subject: "anySubject", #content: [#application: "Checkers", #description: "Two-player
Checkers game"], #timeStamp: 186034087]
```

See also connectToNetServer, sendNetMessage

createApplicationData

Syntax system.DBAdmin.createApplicationData [#application: "*ApplicationName*", #attribute: [*#Attribute1*: *value1* {, *#Attribute2*: *value2*, *#Attribute3*: *value3, #Attribute4: value4* }]

Description Adds a new DBApplicationData object to the server. Once created, DBApplicationData objects contain read-only data associated with a particular multiuser application.

Example This statement creates a new DBApplicationData object for the movie called Poker with the attributes #dealerName, #tableColor, and #wallArt:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.createApplicationData",
"anySubject", [#application: "Poker", #attribute: [#dealerName: "Larry", #tableColor:
color(#rgb, 155, 0, 75), #wallArt: member(3).media]])
```

It is important that at least one of the attributes contain a string or an integer so that the object can be identified with getApplicationData or deleteApplicationData.

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID:
"system.DBAdmin.createApplicationData", #subject: "anySubject", #content: [#application:
"Poker"], #timeStamp: 189123520]
```

See also sendNetMessage, getApplicationData, deleteApplicationData

createUniqueName

Syntax system.group.createUniqueName

Description Server command; obtains the name of an unused group from the server. This group name is unique and not previously used in the movie.

Example The following statement has the server respond with a message that has the same subject. An error code of 0 means that the operation was successful. The contents contain a string that can be used as a group name.

```
errCode = gMultiuserInstance.sendNetMessage("system.group.createUniqueName",
"anySubject")
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.group.createUniqueName",
#subject: "anySubject", #content: "@RndGroup0", #timeStamp: 34653020]
```

See also sendNetMessage

createUser

Syntax system.DBAdmin.createUser [#userID: *userName*, #password: *passwordString,* #userlevel: *integer*]

Description Adds a new DBUser database object to the server. The userID and password must be limited to forty characters each and may not contain # or @ symbols.

Example This statement creates a new DBUser object for the user Bob with the password MySecret and a user level of 40:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.createUser", "anySubject",
[#userID: "Bob", #password: "MySecret", #userlevel: 40])
```

The #userlevel attribute is optional. If omitted it will default to the level specified in the server's Multiuser.cfg file.

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBAdmin.createUser",
#subject: "anySubject", #content: [#userID: "Bob"], #timeStamp: 180885670]
```

declareAttribute

Syntax system.DBAdmin.declareAttribute [#attribute: *#attributeName*]

Description Declares a new attribute name that can be used by any database object. Attributes may not be set until they have been declared. Attribute names must always be symbols. Group attributes do not need to be declared with declareAttribute.

Example This statement declares a new attribute called #email:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.declareAttribute",
"anySubject", [#attribute: #email])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBAdmin.declareAttribute",
#subject: "anySubject", #content: [#attribute: #email], #timeStamp: 184545570]
```

See also sendNetMessage, setAttribute

delete

Syntax

system.movie.delete ["*movieName*"]

system.movie.delete ["*movieName1*", "*movieName2*", "*movieName3*"]

system.group.delete ["*groupName*"]

system.group.delete ["*groupName1*", "*groupName2*", "*groupName3*"]

system.user.delete ["*userID*"]

system.user.delete ["*userID1*", "*userID2*", "*userID3*"]

Description

Server command; when movie is specified, immediately deletes all instances of the given movie or movies from the server. New connections to that movie are still possible if disableMovie has not been called.

If group is specified, the given group(s) is deleted from the server. This does not delete the users who occupied the group.

If user is specified, the given user(s) is immediately disconnected from the server. To disconnect several users, send a list of users.

The server responds with a message that has the same object and command in the #sender and the same #subject and #contents.

Note: There is also a delete command in the general Lingo Dictionary that is applied to chunk expressions. It should not be confused with the Multiuser Lingo delete command, which is always used in the context of a sendNetMessage() command.

Example

This statement deletes all instances of the movie TankWars on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.delete", "anySubject"
"TankWars")
```

To delete more than one movie, use a list containing the movie names:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.delete", "anySubject"
["TankWars", "TicTacToe", "Checkers"])
```

This statement deletes the group @RedTeam from the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.delete", "anySubject"
"@RedTeam")
```

This statement disconnects the user BillyJoe from the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.delete", "anySubject", "BillyJoe")
```

See also

sendNetMessage, disableMovie

deleteApplication

Syntax system.DBAdmin.deleteApplication[#application: *applicationName*]

Description Deletes one or more DBApplication objects from the server. Also deletes all attributes of the DBApplication object, and all DBApplicationData and DBPlayer objects associated with it.

Example This statement deletes the DBApplication object for the movie Poker.

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.deleteApplication",
"anySubject", [#application: "Poker"])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBAdmin.deleteApplication",
#subject: "anySubject", #content: [#application: "Poker"], #timeStamp: 186056753]
```

See also sendNetMessage, createApplication

deleteApplicationData

Syntax system.DBAdmin.deleteApplicationData [#application: "*applicationName*", #attribute: *#attributeName*, #text: "*String*"]

Description Deletes one or more DBApplicationData objects from the server.

Example This statement deletes the DBApplicationData object for the movie Poker with an attribute called #dealerName containing the string Larry.

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.deleteApplicationData",
"anySubject", [#application: "Poker", #attribute: #dealerName, #text: "Larry"])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID:
"system.DBAdmin.deleteApplicationData", #subject: "anySubject", #content: [#application:
"Poker", #attribute: #dealerName, #text: "Larry"], #timeStamp: 193099437]
```

See also sendNetMessage, createApplicationData

deleteAttribute

Syntax system.group.deleteAttribute [#group: "@*groupName*", #attribute: #*attributeName*]

system.group.deleteAttribute [#group: ["@*groupName1*", "@*groupName2*", "@*groupName3*"], #attribute: [#*attributeName1*, #*attributeName2*]]

system.DBUser.deleteAttribute [#userID: "*userName*", #attribute: #*attributeName*]

system.DBUser.deleteAttribute [#userID: ["*userName1*", "*userName2*"], #attribute: [#*attributeName1*, #*attributeName2*, #*attributeName3*]]

system.DBPlayer.deleteAttribute [#userID: "*userName*", #application: "*appName*", #attribute: #*attributeName*]

system.DBApplication.deleteAttribute [#application: "*appName*", #attribute: #*attributeName*]

Description Deletes an attribute with the given name from the given group or database object. Either a #userID or a #application may be supplied. If both are supplied, the attribute is deleted from the DBPlayer object.

Example This statement deletes the attribute #accountBalance from the DBPlayer object for the user Bob in the movie Poker.

```
errCode = gMultiuserInstance.sendNetMessage ( "system.DBPlayer.deleteAttribute",
"anySubject", [#userID: "Bob", #application: "Poker", #attribute: #accountBalance] )
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBPlayer.deleteAttribute",
#subject: "anySubject", #content: [:], #timeStamp: 7430457]
```

See also sendNetMessage

deleteMovie

This Lingo is obsolete. Use delete instead.

deleteUser

Syntax system.DBAdmin.deleteUser [#userID: *userName*]

Description Deletes a DBUser database object from the server.

Example This statement deletes the DBUser object for the user Bob from the server.

```
errCode = gMultiuserInstance.sendNetMessage ( "system.DBAdmin.deleteUser",
"anySubject", [#userID: "Bob"] )
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBAdmin.deleteUser",
#subject: "anySubject", #content: [#userID: "Bob"], #timeStamp: 183543403]
```

See also sendNetMessage, createUser

disable

Syntax

system.movie.disable ["*movieName*"]

system.group.disable ["*groupName*"]

Description

Server command; when movie is specified, disables the given movie from making any future connections. Current connections are not broken. When group is specified, the given group is disabled from accepting new members. Current members are not removed from the group.

If the server is not set up with the allowMovie command in the configuration file, the server will by default allow connections from all movies. The disable command blocks logons from specific movies, while the enable command reenables a movie disabled by disable. Never disable all movies; this prevents even an administrator movie from logging in to reenable other movies.

The server responds with a message that has the same object and command in the #sender and the same #subject and #contents.

Example

This statement disables all future connections by the movie TankWars on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.disable", "anySubject"
"TankWars")
```

This statement disables more than one movie by using a list containing the movie names:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.disable", "anySubject"
["TankWars", "TicTacToe", "Checkers"])
```

This statement disables the group @RedTeam from accepting new members:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.disable", "anySubject"
"@RedTeam")
```

See also

sendNetMessage, delete, enable

disableMovie

This Lingo is obsolete. Use disable instead.

disconnectUser

This Lingo is obsolete. Use delete instead.

enable

Syntax

system.movie.enable ["*movieName*"]

system.movie.enable ["*movieName1*", "*movieName2*", "*movieName3*"]

system.group.enable ["*groupName*"]

system.group.enable ["*groupName1*", "*groupName2*", "*groupName3*"]

Description

Server command; when movie is specified, enables the given movie on the server so future connections can be made. Current connections are not affected. When group is specified, the given group is enabled to accept new members.

The server will respond with a message that has the same object and command in the #sender and the same #subject and #contents.

If the server is not set up with the allowMovie command in the configuration file, it will by default allow connections from all movies. The disable command will block out logons from specific movies, while the enable command will reenable a movie disabled by disable. Never disable all movies; this would prevent even an administrator movie from logging in to reenable other movies.

Example

This statement enables the movie TankWars on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.enable", "anySubject",
"TankWars")
```

To enable more than one movie, use a list containing the movie names:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.enable", "anySubject",
["TankWars", "TicTacToe", "Checkers"])
```

This statement enables the group @RedTeam to allow new members to join:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.enable", "anySubject",
"@RedTeam")
```

See also

sendNetMessage, disable

enableMovie

This Lingo is obsolete. Use enable instead.

getAddress

Syntax system.user.getAddress ["*userID*"]

Description Server command; when sent to the server, returns a specific user's IP address.

Example This statement obtains the IP address for the user logged in as Jane:

```
errCode = gMultiuserInstance.sendNetMessage( "system.user.getAddress", "anySubject",
"Jane")
```

The server responds with a message that has the same subject. The content of the message is a property list containing the requested information.

Note: The IP address may not be correct if the user is behind a firewall.

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.user.getAddress", #subject:
"anySubject", #content: [#userID: "Jane", #ipAddress: "123.45.67.1"], #timeStamp:
763283481]
```

See also getNetAddressCookie

getApplicationData

Syntax

```
system.DBApplication.getApplicationData [#application: "appName", #attribute:
#attributeName, #text: "searchString"]
```

```
system.DBApplication.getApplicationData [#application: "appName", #attribute:
#attributeName, #number: integer]
```

```
system.DBApplication.getApplicationData [#application: "appName", #attribute:
#attributeName, #lowNum: integer, #highNum: integer]
```

Description Returns the list of attributes and values from all DBApplicationData objects that correspond to the given application and contain the given attribute with the given value. The given value may be a string, an integer, or a range of integers. The result is a list of lists, each of which is the list of attributes and values for a single DBApplicationData object.

If the #application parameter is omitted, it defaults to the movie ID the current movie used to connect to the server.

Up to one hundred DBApplicationData objects may be returned per request.

Example This statement returns the lists of attributes from the DBApplicationData objects for the movie Poker that contain the attribute #dealerName with a value of Larry:

```
errCode = gMultiuserInstance.sendNetMessage ("system.DBApplication.getApplicationData",
"anySubject", [#application: "Poker", #attribute: #dealerName, #text: "Larry"])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID:
"system.DBApplication.getApplicationData", #subject: "anySubject", #content:
[[#dealerName: "Larry", #tableColor: color(#rgb, 155, 0, 75), #wallArt: (media 7afa4d0)]],
#timeStamp: 189027987]
```

See also createApplicationData, deleteApplicationData, sendNetMessage

getAttribute

Syntax

system.group.getAttribute [#group: "@groupName", #attribute: [#attributeName1, *#attributeName2*]]

system.DBPlayer.getAttribute [#userID: "userName", #application: "appName", #attribute: [#attributeName1, *#attributeName2*]]

system.DBUser.getAttribute [#userID: "userName", #attribute: [#attributeName1, *#attributeName2*]]

system.DBApplication.getAttribute [#application: "appName", #attribute: [#attributeName1, *#attributeName2*]]

Description

Obtains from the server the values of the given attributes for the given group or object. Attributes may contain any Lingo value.

Example

This statement gets the values of the attributes #accountBalance and #cardHand for the user Bob in the movie Poker:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBPlayer.getAttribute",
"anySubject", [#userID: "Bob", #application: "Poker", #attribute: [#accountBalance,
#cardHand]])
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBPlayer.getAttribute",
#subject: "anySubject", #content: ["Bob": [#accountBalance: 3500, #cardHand: "Royal
Flush", #lastUpdateTime: "1999/08/26 12:43:33.070364"]], #timeStamp: 7025771]
```

See also

setAttribute, sendNetMessage

getAttributeNames

Syntax

system.group.getAttributeNames [#group: "@groupName"]

system.DBUser.getAttributeNames [#userID: "userName"]

system.DBApplication.getAttributeNames [#application: "appName"]

system.DBPlayer.getAttributeNames [#userID: "userName", #application: "appName"]

Description

Gets the list of attribute names that have been set for the given group or database object. If #userID is supplied, the attribute list is returned for the user's DBUser object. If #application is supplied, the attribute list is returned for the movie's DBApplication object. If both are supplied, the attribute list is returned for the user's DBPlayer object for the given movie.

Example

This statement gets the list of attributes that have been set for the DBPlayer object of the user Bob in the movie Poker:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBPlayer.getAttributeNames",
"anySubject", [#userID: "Bob", #application: "Poker"])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBPlayer.getAttributeNames",
#subject: "anySubject", #content: ["Bob": [#accountBalance, #cardHand,
#lastUpdateTime]], #timeStamp: 7326833]
```

See also

declareAttribute, setAttribute, sendNetMessage

getGroupCount

Syntax

```
system.movie.getGroupCount ["movieName"]
system.user.getGroupCount ["userName"]
```

Description Server command. When movie is specified, returns the number of groups that exist in the specified movie. If no movie is given, the result is for the current movie.

When user is specified, returns the number of groups the given user is a member of. If no user is given, the number returned is for the current user.

Example This statement gets the number of groups in the current movie:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.getGroupCount",
"anySubject")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.movie.getGroupCount",
#subject: "anySubject", #content: ["currentMovieName": 2], #timeStamp: 763283481]
```

This statement gets the number of groups the current user is a member of:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.getGroupCount", "anySubject")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.user.getGroupCount",
#subject: "anySubject", #content: ["userName": 2], #timeStamp: 763283987]
```

This statement gets the number of groups for users Bob and Mary:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.getGroupCount", "anySubject",
["Bob", "Mary"])
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.user.getGroupCount",
#subject: "anySubject", #content: ["Bob": 3, "Mary": 4], #timeStamp: 763284273]
```

getGroupList

This Lingo is obsolete. Use getGroups instead.

getGroupMembers

This Lingo is obsolete. Use getUsers instead.

getGroups

Syntax

```
system.movie.getGroups
system.user.getGroups
```

Description

Server command; when movie is specified, gets the list of groups for the application in the current connection, including the predefined group @AllUsers. When user is specified, returns a list of current groups the user is a member of.

The server responds with a message that has the same object and command in the #sender and the same #subject, and #contents comprising a list of strings naming the groups.

Example

This statement retrieves the current list of groups in this movie connection:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.getGroups", "anySubject")
```

The server's response would look like the following:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.movie.getGroups", #subject:
"anySubject", #content: [#movieID: "theMovieName", #groups: ["@AllUsers", "@RedTeam",
"@BlueTeam"]], #timeStamp: 79349843]
```

This statement returns a list of the current groups that the sender is a member of:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.getGroups", "anySubject")
```

The returned list would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.user.getGroups", #subject:
"anySubject", #content: [#userID: "userName", #groups: ["@AllUsers", "@Photographers",
"@Designers"]], #timeStamp: 79349843]
```

See also

sendNetMessage

getListOfAllMovies

This Lingo is obsolete. Use getMovies instead.

getMovieCount

Syntax

```
system.server.getMovieCount
```

Description

Server command; returns the number of different applications on the server. This is the number of different movies on the server, such as Checkers and Chess, not different instances of the same movie.

Example

This statement gets the number of different applications on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.getMovieCount",
"anySubject")
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.server.getMovieCount",
#subject: "anySubject", #content: 3, #timeStamp: 30214905]
```

getMovies

Syntax `system.server.getMovies`

Description Server command; returns a list of the current movies connected to the server.

Example This statement returns a list of all the movies currently connected to the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.getMovies", "anySubject")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.server.getMovies", #subject:
"anySubject", #content: ["TankWars", "TicTacToe", "TechChat"], #timeStamp: 61726385]
```

See also sendNetMessage

getNewGroupName

This Lingo is obsolete. Use createUniqueName instead.

getNumberOfMembers

This Lingo is obsolete. Use getUserCount instead.

getServerTime

This Lingo is obsolete. Use getTime instead.

getServerVersion

This Lingo is obsolete. Use getVersion instead.

getTime

Syntax `system.server.getTime`

Description Server command; returns the current time from the server. The server responds with a message containing a string representing the time.

For synchronization between movies, it is better to examine the #timeStamp property returned in a message from the server. To get the current server #timeStamp value, just send a message to yourself.

Example This statement shows a request for the server to send back the current time:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.getTime", "anySubject")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.server.getTime", #subject:
"anySubject", #content: "1999/08/25 18:22:27", #timeStamp: 30203034]
```

See also sendNetMessage

getUserCount

Syntax system.movie.getUserCount [*movieName*]

system.group.getUserCount [*groupName*]

Description Server command; returns the number of users logged in to the given movie or group. When movie is specified, the result is the same as calling getUsers for the group @AllUsers. If no movie is specified, the result is for the current movie. When group is specified, the result is for the given group. The reply message's contents is a property list.

Example This statement returns the number of users logged into the current movie ID on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.getUserCount", "anySubject")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.movie.getUserCount",
#subject: "anySubject", #content: 17, #timeStamp: 30231031]
```

This statement has the server report the number of members in the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getUserCount", "anySubject",
"@RedTeam")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.group.getUserCount",
#subject: "anySubject", #content: [#groupName: "@RedTeam", #numberMembers: 6],
#timeStamp: 30234705]
```

To find the number of members in more than one group at a time, put the group names in a list in a statement similar to this:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getUserCount", "anySubject",
["@RedTeam", "@BlueTeam", "@GreenTeam"])
```

The server will respond with a separate message for each group.

See also sendNetMessage

getUserGroups

This Lingo is obsolete. Use getGroups instead.

getUserIPAddress

This Lingo is obsolete. Use getAddress instead.

getUsers

Syntax system.group.getUsers ["@*groupName*"]

Description Server command; returns the list of members for a particular group or groups.

If there is more than one group, the server responds with a separate message for each of the groups requested. Each response will have the same subject as the original request. The reply message's contents will be in the form of a property list with two properties, the first for the group name and the second for the members.

For large groups, the return message may be fairly long. Be certain that the server and Xtra message buffers are large enough to contain the list of all group members. See setNetBufferLimits and "Configuring the server".

Example This statement has the server return a list of members in the @RedTeam group:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.getUsers", "anySubject",
"@RedTeam")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.group.getUsers", #subject:
"anySubject", #content: [#groupName: "@RedTeam", #groupMembers: ["Mark", "Jane",
"Chris"]], #timeStamp: 98324982]
```

See also sendNetMessage

getVersion

Syntax system.server.getVersion

Description Server command; this function requires no content. The server will return a message with the same subject and content containing information regarding the server application itself.

Possible values for #platform are currently Macintosh and Windows.

Example The following statement shows the request for server information:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.getVersion", "anySubject")
```

The reply message is similar to this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.server.getVersion", #subject:
"anySubject", #content: [#vendor: "Macromedia", #version: "2.0", #platform: "Macintosh"],
#timeStamp: 30196205]
```

See also sendNetMessage

join

Syntax system.group.join ["@*groupName*"]

Description Server command; adds the sender to a group. If the group doesn't exist it is automatically created. If the user already belongs to the group, no error occurs.

The server responds with a message for each group joined, matching the subject of the request message, and the content containing the name of the group successfully joined.

Example This statement adds the sender to the group @BeatleLovers:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.join", "anySubject",
"@BeatleLovers")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.group.join", #subject:
"anySubject", #content: "@BeatleLovers", #timeStamp: 21765127]
```

This statement adds the sender to more than one group at the same time by putting the names of the groups in a list:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.join", "anySubject",
["@BeatleLovers", "@Photographers", "@Designers"])
```

The server will send a separate response for each group.

See also sendNetMessage

joinGroup

This Lingo is obsolete. Use join instead.

leave

Syntax system.group.leave ["@*groupName*"]

Description Server command; removes the current user from the given group. If the user is not a member of the group, the command is ignored.

Calling leave for the default group @AllUsers returns an error message.

The server responds with a message for each group that is left, matching the subject of the request message, and the content containing the name of the group successfully left.

Example This statement removes the sender from the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.leave", "anySubject",
"@RedTeam")
```

The server response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.group.leave", #subject:
"anySubject", #content: "@RedTeam", #timeStamp: 762551131]
```

See also sendNetMessage

leaveGroup

This Lingo is obsolete. Use leave instead.

setAttribute

Syntax

system.group.setAttribute [#group: "@groupName", #attribute: [#attribute1: value1, *#attribute2: value2*, *#lastUpdateTime: "timeString"*]]

system.DBUser.setAttribute [#userID: "userName", #attribute: [#attribute1: value1, *#attribute2: value2*, *#lastUpdateTime: "timeString"*]]

system.DBPlayer.setAttribute [#userID: "userName", #application: "appName", #attribute: [#attribute1: value1, *#attribute2: value2*, *#lastUpdateTime: "timeString"*]]

system.DBApplication.setAttribute [#application: "appName", #attribute: [#attribute1: value1, *#attribute2: value2*, *#lastUpdateTime: "timeString"*]]

Description

Sets the value of an attribute for a group or a database object. If a group attribute is being set, the group name is supplied. If a database object attribute is being set, the #userID, the #application, or both are supplied. If both are supplied the attribute is set for the DBPlayer object for the given user in the given application.

The #lastUpdateTime property is optional and lets you determine whether some other user has updated the attributes of the group since you last checked them with getAttribute. When you use getAttribute, the server responds with the values of the attributes you requested plus a #lastUpdateTime property, which indicates the moment in time when the server read the values of those attributes for the group you requested. The #lastUpdateTime property is a string containing the year, month, day, hour, minutes, seconds, and microseconds on the server. By sending this same string with your setAttribute command, you allow the server to check whether the attributes for the group have been updated since you last checked them.

If the server determines that the attributes for the group have been updated by someone else since you checked them, it will respond with an #errorCode indicating a concurrency error. If no one else has updated the attributes since you checked them, the server will respond with a new #lastUpdateTime for the group indicating that you have just updated the attributes.

Example

This statement sets the attributes #teamLeader and #location for the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.setAttribute", "anySubject",
[#group: "@RedTeam", #attribute: [#teamLeader: "Mary", #location: "New York",
#lastUpdateTime: "1999/07/26 15:26:33:123456"]])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.group.setAttribute",
#subject: "anySubject", #content: ["@RedTeam": [#lastUpdateTime: "1999/08/25
18:55:33.132456"]], #timeStamp: 32189685]
```

This statement sets the attribute #favoriteColor for the DBUser object Bob:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBUser.setAttribute", "anySubject",
[#userID: "Bob", #attribute: [#favoriteColor: "Blue", #lastUpdateTime: "1999/07/26
15:26:33:123456"]])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBUser.setAttribute",
#subject: "anySubject", #content: ["Bob": [#lastUpdateTime: "1999/08/25
13:28:22.123456"]], #timeStamp: 184472070]
```

This statement sets the attributes #accountBalance and #cardHand for the DBPlayer object of the user Bob in the movie Poker:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBPlayer.setAttribute", "anySubject",
[#userID: "Bob", #application: "Poker", #attribute: [#accountBalance: 3500, #cardHand:
"Royal Flush", #lastUpdateTime: "1999/07/26 15:26:33:123456"]])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBPlayer.setAttribute",
#subject: "anySubject", #content: ["Bob": [#lastUpdateTime: "1999/08/26
12:43:33.647483"]], #timeStamp: 6461476]
```

This statement sets the attribute #highScore for the DBApplication object Basketball:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBApplication.setAttribute",
"anySubject", [#application: "Basketball", #attribute: [#highScore: 1352, #lastUpdateTime:
"1999/07/26 15:26:33:123456"]])
```

The server's response would look like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.DBApplication.setAttribute",
#subject: "anySubject", #content: ["Basketball": [#lastUpdateTime: "1999/08/25
14:16:14.852673"]], #timeStamp: 187351603]
```

See also getAttribute, sendNetMessage

Flat file database commands

The following commands control the standard dbaseIV DBF files supported by earlier versions of the server as well as Version 2.0. These commands may not be supported in future versions of the server, however. Send these commands as messages with `sendNetMessage()` using a "System.DBF.*commandName*" syntax.

appendRecord

Syntax system.DBF.appendRecord

Description Server database command; creates a new, empty record at the end of the current database.

If the operation is successful, the newly created record becomes the current record.

The server will respond with a message that has the same subject and message.

Example This statement tells the server to create a new, empty record at the end of the current database, with the result code from the Xtra being placed in the variable errCode:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.appendRecord", "anySubject")
```

See also sendNetMessage

deleteRecord

Syntax system.DBF.deleteRecord

Description Server database command; marks the current database record for deletion and issues an error if the record is locked by another user. The record is not removed from the file until the `pack` command is issued. The record must be locked by the sender before being deleted.

Example This statement has the server mark the current record for deletion:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.deleteRecord", "anySubject")
```

See also sendNetMessage, pack

getFields

Syntax system.DBF.getFields

Description Server database command; returns the given field values from the current record. The contents indicate which fields to read.

The server will respond with a message that has the same subject and content. The return message will contain a property list with the field symbols and their values.

To check the contents of more than one field, format the field names in a linear list.

Example The following statements have the server report the values in specific fields of a database record. The result is a property list of field symbols and their values.

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.getFields", "anySubject",
[#HISCORE])
```

This statement checks the contents of more than one field:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.getFields", "anySubject",
[#HISCORE, #DATE, #CARTYPE])
```

See also sendNetMessage

getReadableFieldList

Syntax system.DBF.getReadableFieldList

Description Server database command; returns a list of the fields the user can read from the database.

The server responds with a message that has the same subject and an error code indicating success. The response contains a linear list with field names of the database as symbols.

The result is a list of symbols representing the fields that can currently be read from the database. This field list depends on the database configuration and the user levels.

Example This statement returns the list of readable fields from the currently selected database:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.getReadableFieldList",
"anySubject")
```

The #content value of the return message is similar to this:

```
[#FirstName, #LastName, #Score]
```

getRecordCount

Syntax system.DBF.getRecordCount

Description Server database command; returns an integer representing the number of records in the database, including those currently marked for deletion.

Example This statement has the server report the number of records in the current database:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.getRecordCount", "anySubject")
```

See also sendNetMessage

getRecords

Syntax system.DBF.getRecords

Description Server database command; returns a linear list containing data from multiple records. After this function is called, the current record pointer is left at the last record read.

The server responds with a message that has the same subject, and the content is the property list containing the data. Each element of the linear list is a property list with data from all readable files.

This function fails if one of the fields is locked by another user.

Example This statement has the server report the data from two records (the current record and one successive record):

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.getRecords", "anySubject", 2)
```

The result would be something such as this:

```
[#errorCode: 0, #recipients: ["Jay"], #senderID: "System", #subject: "getRecords", #content:
[[#FIRSTNAME: "Joe", #LASTNAME: "Smith", #CLOTHING: "pants", #COLOR: "green",
#WEIGHT: 157.0000, #SIZE: 42.0000], [#FIRSTNAME: "Mary", #LASTNAME: "Jones",
#CLOTHING: "blouse", #COLOR: "white", #WEIGHT: 132.0000, #SIZE: 6.0000]],
#timeStamp: 425722419]
```

See also sendNetMessage

getWriteableFieldList

Syntax system.DBF.getWriteableFieldList

Description Server database command; lists the fields the user can write to in the database.

The return message's contents will be in the form of a list of symbols representing the fields that may currently be written to in the database. This field list depends on the database configuration and the user levels.

Example This statement has the server report the database fields that the sender can write to:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.getWriteableFieldList",
"anySubject")
```

The response #content can be a list such as this:

```
[#FirstName, #ItemID, #NumberWanted]
```

See also sendNetMessage

goToRecord

Syntax system.DBF.goToRecord

Description Server database command; moves the current record pointer to the requested record of the current database. The return message contains the same subject and an error code.

Example This statement has the server make record 125 the current record of the database:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.goToRecord", "anySubject",
125)
```

See also sendNetMessage

isRecordDeleted

Syntax system.DBF.isRecordDeleted

Description Server database command; queries the server to determine whether the current record is marked for deletion (TRUE) or not (FALSE).

The server responds with a message that has the same subject and the content of the message containing the True or False value.

Example This statement queries the server to determine whether the current record is marked for deletion:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.isRecordDeleted", "anySubject")
```

See also sendNetMessage

lockRecord

Syntax system.DBF.lockRecord

Description Server database command; locks the current record and prevents other users from modifying the record until the unlockRecord command is issued. An error occurs if the record is already locked by another user.

The server responds with a message that has the same subject and an error code of zero if the operation is successful.

Example This statement locks the current record:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.lockRecord", "anySubject")
```

See also sendNetMessage

pack

Syntax system.DBF.pack

Description Server database command; packs the database, compresses any open memo files, and permanently removes all deleted records. An error occurs if any record is locked by another user.

The server responds with a message that has the same subject and an error code of zero if the operation is successful.

Example This statement issues the pack command on the current open database:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.pack", "anySubject")
```

See also sendNetMessage

recallRecord

Syntax system.DBF.recallRecord

Description Server database command; marks the current record as not deleted. An error occurs if the record is locked by another user. The record must be locked by the sender prior to being deleted.

The server responds with a message that has the same subject and an error code of zero if the operation is successful.

Example This statement marks the current record as not deleted:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.recallRecord", "anySubject")
```

See also sendNetMessage

reIndex

Syntax system.DBF.reIndex

Description Server database command; rebuilds the index file for the currently open database. Reindexing sorts all tags defined in the index file. No packing or deleting of records occurs. If any records are locked, an error is generated and the command fails.

Because the indices are updated whenever field data changes, reIndex is useful only as an occasional administration function and can require a significant amount of time.

If the operation is successful, the server responds with a message that has the same subject and an error code of zero.

Example This statement reindexes the current database:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.reIndex", "anySubject")
```

See also sendNetMessage

seek

Syntax system.DBF.seek

Description Server database command; moves the record pointer to the first record in the current sort order that matches the search criteria following seek. Use selectTag first to establish the database sort order.

The server responds with a message containing the same subject and content. The error code indicates whether the command encountered problems. After seeking data, the current record points to the first record that matches search criteria. To confirm the results, examine the current record with getFields.

Example

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.seek", "anySubject",
"Enlightenment")
```

See also sendNetMessage, selectTag

selectDatabase

Syntax system.DBF.selectDatabase

Description Server database command; selects the current database to be used. All further actions, such as getFields, operate on the database that this command selects. This is the first command sent when accessing a database. When database operations are completed, the selectDatabase command should be sent with an empty string content. This releases the database access on the server and improves server performance.

If the operation is successful, the server responds with a message that has the same subject and an error code of zero.

The database must be specified in either the Multiuser.cfg file or a particular movie's configuration file in order to be accessed.

Example This statement makes the database HighScore the current database:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.selectDatabase", "anySubject", 
"HighScore.dbf")
```

This statement terminates the same session by sending an empty string as the database to select:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.selectDatabase","anySubject", "")
```

See also sendNetMessage

selectTag

Syntax system.DBF.selectTag

Description Server database command; selects the current index tag to order the database. The index tags are defined in the database file when the database is created.

The server responds with a message that has the same subject and an error code of zero if the operation is successful.

After a tag is selected, the current record pointer is set to the beginning of the file.

Example This statement has the server select the LastName tag to reorder the current database:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.selectTag", "anySubject", 
"LastName")
```

setFields

Syntax system.DBF.setFields

Description Server database command; sets the value of the specified fields in the current database record. The current user must use the lockRecord command to lock the record before using the setFields command.

The server responds with a message that has the same subject and an error code of zero if the operation is successful. The contents are omitted from the response to limit network traffic.

If you use a variable as the contents of the setFields command, the variable must be a property list.

Example This statement sets the value in the Age field of the current record:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.setFields", "anySubject", [#AGE:
24])
```

This statement sets more than one field at once:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.setFields", "anySubject",
[#NAME: "Dave", #AGE: 29] )
```

See also sendNetMessage

skip

Syntax system.DBF.skip

Description Server database command; moves the current record pointer. The content of the message indicates how far (in number of records) to move the pointer, and the direction in which to move it. A positive number moves the pointer forward in the database; a negative number moves the pointer backward.

The server responds with a message that has the same subject and content.

Example This statement moves the pointer forward one record in the database:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.skip", "anySubject", 1)
```

unlockRecord

Syntax system.DBF.unlockRecord

Description Server database command; unlocks the current record, allowing other users to modify it.

The server responds with a message that has the same subject and an error code of zero if the operation is successful.

Example This statement unlocks the current record:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBF.unlockRecord", "anySubject")
```