

SCE-RT SDK Medius API Release 2.10

Overview

March 2005

© 2005 Sony Computer Entertainment Inc.

Publication date: March 2005

Sony Computer Entertainment Inc.
2-6-21, Minami-Aoyama, Minato-ku
Tokyo 107-0062, Japan

Sony Computer Entertainment America
919 E. Hillsdale Blvd.
Foster City, CA 94404, U.S.A.

Sony Computer Entertainment Europe
30 Golden Square
London W1F 9LD, U.K.

The *SCE-RT SDK Medius API Release 2.10 – Overview* is supplied pursuant to and subject to the terms of the Sony Computer Entertainment PlayStation® license agreements.

The *SCE-RT SDK Medius API Release 2.10 – Overview* is intended for distribution to and use by only Sony Computer Entertainment licensed Developers and Publishers in accordance with the PlayStation® license agreements.

Unauthorized reproduction, distribution, lending, rental or disclosure of this book to any third party, in whole or in part, is expressly prohibited by law and by the terms of the Sony Computer Entertainment PlayStation® license agreements.

Ownership of the physical property of the book is retained by and reserved by Sony Computer Entertainment.
Alteration to or deletion of the book, in whole or in part, its presentation, or its contents is prohibited.

The information in the *SCE-RT SDK Medius API Release 2.10 – Overview* is subject to change without notice. The content of this book is Confidential Information of Sony Computer Entertainment.


“” and “PlayStation” are registered trademarks of Sony Computer Entertainment Inc. All other trademarks are property of their respective owners and/or their licensors.

Table of Contents

About This Manual	vii
Introduction	1
General Steps in Executing Medius API Function Calls	3
Connecting to the Medius Platform	6
Initializing the Medius Client Library	7
MediusInitialize() vs. MediusInitializeBare()	7
MediusInitializeInParams	7
MediusInitializeOutParams	7
MediusClose() vs. MediusCloseBare()	8
Sample – Initializing the Medius API	8
Connecting to the MAS	8
Making the Initial Connection to the MAS	9
Sample – Connecting to the MAS	10
Starting a Session	12
Sample – Starting a Session	12
Version Information and Build Timestamps	13
Sample – Version Information and Build Timestamps	14
Establishing an Authenticated State	15
Registering a New Account	15
Account Login	16
Sample – Account Login and Registering a New Account	17
Anonymous Logins	19
Reconnecting to the MLS	20
Sample – Connecting to the MLS	21
Disconnecting From the Medius Platform	23
Graceful Disconnects	23
Ungraceful Disconnects	23
Server Time	25
What is a ‘World’?	26
Reporting (Maintaining Medius Platform Connections and Objects)	27
Understanding MUM Game World Objects and DME Game Worlds	28
If a Client/Server Game	28
If a Peer-to-Peer Game	29
Report Intervals	29
Special Notes about Peer-to-Peer Host Migration and MediusWorldID	30
How and When a MediusWorldID Can Change	30
Maintaining the Medius Connection	30
Additional Notes About MediusUpdate()	30
Sending World Reports (Maintaining Game World Objects)	31
Additional Notes About MediusSendWorldReport()	31
The Sequence for Sending a ‘World Report’	32
Sample – Sending a World Report	33
Sending an End Game Report (Closing Game World Objects)	34
Coordinating the Sending of End Game Reports	34
Additional Notes About MediusSendEndGameReport()	35
The Sequence for Sending an ‘End Game Report’	36
Sample – Sending an End Game Report	36
Sending Player Reports (Maintaining Player Objects)	37

Additional Notes About MediusSendPlayerReport()	37
Special Note About a Player's Stats Field	38
Sequence for Sending a 'Player Report'	38
Sample – Sending a Player Report	38
Maintaining the MGCL Connection	39
Additional Notes about MGCLUpdate()	39
Sending Server Reports (Maintaining Game Server Objects)	39
Additional Notes About MGCLSendServerReport()	39
Sequence for Sending a 'Server Report'	40
Sample – Sending a Server Report	40
Leaving a Game World	41
Chat Channel Management	43
Preventing a Player from Joining a Chat Channel	43
Disabling or Enabling Medius Chat	44
Creating a Chat Channel	44
Searching for a Player	44
Searching for a Chat Channel	44
Getting Chat Channel Information	44
Retrieving a List of Chat Channels	44
Controlling the Number of Chat Channels Returned	45
Retrieving a List of Players in a Chat Channel	45
Getting Player Object Information	45
Getting the Total Chat Channel Count	45
Getting the Total Player Count	46
Chat Channel Security	46
Joining a Chat Channel	46
Sending a Chat Message (Broadcast or Instant Messaging)	46
Receiving a Chat Message	47
Chat Message History	47
Filtering Chat Channels	47
Game Management	49
Preventing a Player from Joining a Game	49
Searching for a Player	49
Searching for a Game	49
Getting Game World Object Information	50
Retrieving a List of Players in a Game World Object	50
Retrieving a List of Game World Objects	50
Controlling the Number of 'Game World Objects' Returned	51
Getting Player Object Information	51
Getting the Total Game World Object Count	51
Getting the Total Player Count	51
Game World Object Security	51
Creating a Game	52
If Creating a Client/Server Game	52
If Creating a Peer-to-Peer Game	52
Joining a Game	53
Joining Either a Client/Server Game or a Peer-to-Peer Game	53
Vote Player Out of a Game	53
Design Considerations	53
Account Management	55
Deleting an Account	55

Getting an Account ID	55
Getting an Account Profile	55
Updating an Account Password	55
Updating an Account Profile	55
Updating Account Stats	55
Buddy Lists	56
Retrieving a Player's Buddy List	56
Adding a User to a Player's Buddy List	56
Removing a User From a Player's Buddy List	56
Buddy List Permissions	56
Requesting Permission	57
Responding to a Request for Permission	57
Ignore Lists	58
Retrieving a Player's Ignore List	58
Adding a User to a Player's Ignore List	58
Removing a User From a Player's Ignore List	58
Medius Dynamic List System	59
Introduction	59
Interface	59
Type System	59
Callbacks	60
Tailoring the Subscription	62
Subscribing	62
Setting an Interest Callback	62
Associating an Interest Callback	63
Usage	63
Service Levels	63
Game List Row Object Declaration	63
Field Specification Definition	64
Field Map Definition	64
DType Declaration	64
DType Creation	65
DType Destruction	65
Platform Requirement Functionality	66
Retrieving the Usage Policy and the Privacy Policy	66
Retrieving Announcements	66
Co-location	67
Co-location Example	67
Co-location Process	68
Ladder Management	69
Getting a Player's Current Ladder Stats and Updating	69
Total Number of Players in a Ladder Ranking	69
Getting a List of Player's in a Ladder Ranking	69
Controlling the Number of 'Ladder Ranks' Returned	70
Getting a Player's Current Ladder Ranking	70
Token Management	71
Clan Management	72
Creating a Clan	73
Disbanding a Clan	73

Getting a List of Clan Memberships	73
Getting Clan Information	74
Transferring Clan Leadership	74
Adding Members to a Clan	74
Removing Members from a Clan	75
Getting a List of Members in a Clan	75
Clan Invitations Management	75
Clan Messages Management	76
Clan Team Challenges Management	76
Clan Ladder Management	77
Medius File Services (MFS)	79
The Client Perspective	79
MFS Back End Overview	80
Medius File Services API	80
Initialization	80
Uploading a File	80
Downloading a File	81
Canceling a Upload or Download	81
Listing Files	81
Deleting a File	81
MediusFileServices Sample	81
Vulgarity Filtering	82
Vulgarity Filtering Process	82
AntiCheat	83
Overview	83
Trigger Events	83
Cheat Challenges	84
Cheat Policies	84
Cheat Messages	84
Examples of MsgText in a Cheat Message	84
What AntiCheat Can and Cannot Do	85
Internationalization (Multi-Byte Language Support)	86
Example of Internationalization	86
The Process for Setting Internationalization Parameters:	87
Medius Universe Information Server (MUIS)	88
Connecting to the MUIS	88
The Process for Requesting Universe Information:	89
Running the Medius Servers	90
Operating the Medius Server Executables	90

About This Manual

The *SCE-RT SDK Medius API Release 2.10 – Overview* manual provides a description of the various functionalities of the Medius API. This API is part of the SCE-RT SDK, provided by Sony Computer Entertainment America (SCEA).

Please forward any questions about this document to scert-support@scea.com.

Changes Since Last Release

Several changes/additions were made to various sections throughout the document, including:

- Introduction
- Connecting to the MAS
- Registering a New Account
- Reconnecting to the MLS
- Sample – Connecting to the MLS
- Graceful Disconnects
- Additional Notes About MediusSendEndGameReport()
- Medius Universe Information Server (MUIS)
- Running the Medius Servers

In addition, the following new sections were added:

- Medius File Services (MFS)
- Medius Dynamic List System
- AntiCheat

Typographic Conventions

Certain typographic conventions are used throughout this manual to clarify the meaning of the text:

Convention	Meaning
<code>courier</code>	Indicates literal program code.
<i>italic</i>	Indicates names of arguments and structure members (in structure/function definitions only).
medium bold	Indicates data types and structure/function names (in structure/function definitions only).
blue	Indicates a hyperlink.

Developer Support

Sony Computer Entertainment America (SCEA)

SCEA developer support is available to licensees in North America only. You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
Attn: Developer Tools Coordinator Sony Computer Entertainment America 919 East Hillsdale Blvd. Foster City, CA 94404, U.S.A. Tel: (650) 655-8000	E-mail: scert-support@scea.com scea_support@ps2-pro.com Web: https://www.ps2-pro.com/ https://psp.scedev.net Developer Support Hotline: (650) 655-5566 (Call Monday through Friday, 8 a.m. to 5 p.m., PST/PDT)

Sony Computer Entertainment Europe (SCEE)

SCEE developer support is available to licensees only in the PAL television territories (including Europe and Australasia). You may obtain developer support or additional copies of this documentation by contacting the following addresses:

Order Information	Developer Support
Attn: Production Coordinator Sony Computer Entertainment Europe 13 Great Marlborough Street London W1F 7HP, U.K. Tel: +44 (0) 20 7859-5000	E-mail: scee_support@ps2-pro.com Web: https://www.ps2-pro.com/ https://psp.scedev.net Developer Support Hotline: +44 (0) 20 7911-7711 (Call Monday through Friday, 9 a.m. to 6 p.m., GMT/BST)

Introduction

The SCE-RT SDK includes the Medius API, DME (Distributed Memory Engine) API, and MGCL (Medius Game Communication Library) API. The Medius API, through corresponding client protocols, targets “lobby” technology and data issues, while the DME API is specific to in-game, real-time networking technology. The DME facilitates communication between game clients, game servers, and the Medius platform. In addition, included in the SCE-RT SDK is the SCE-RT++ Wrapper. If you are developing in C++ it is highly recommended that you use the SCE-RT++ Wrapper instead of making the raw Medius Calls listed in this document.

Note: The SCE-RT 2.10 Client SDK will require SCE-RT 2.10 or higher servers, any other use will result in undetermined behavior.

Use of the MGCL API is optional. The MGCL API provides peer-to-peer application developers the ability to post information about Internet-based ‘hosted’ peer-to-peer games to the Medius platform. A Medius client then uses this information to join existing games that are either in staging or in progress (late joiners).

Medius “lobby” technology encompasses communication and logic for client match-up, player registration and profiling, chat, clans/teams, ladder ranking lists, instant messaging and other lobby-specific needs.

The Medius platform consists of the following elements:

- Medius Universe Manager (MUM)
- Medius Authentication Server (MAS)
- Medius Lobby Server (MLS)
- Medius Proxy Server (MPS)
- Medius Universe Information Server (MUIS)

The Medius elements of the SCE-RT SDK include the following:

- Linux/Win32 Medius Universe Manager Server executable
- Linux/Win32 Medius Authentication Server executable
- Linux/Win32 Medius Lobby Server executable
- Linux/Win32 Medius Proxy Server executable
- PlayStation®2-compatible header files:
 - MediusClient.h
 - mediustypes.h
 - MediusClans.h
 - mgcl.h
 - MediusFileServices.h
 - MediusFileServicesTypes.h
- PlayStation®2-compatible library files:
 - librtmclPS2.a (Medius Client Library)
 - librtmgclPS2.a (Medius Game Communication Library)
 - MediusPublicKey_scei.a
 - General_Okey_scei.a (and/or a special title specific security key)
- Sample code for the PlayStation®2 (which also requires the DME API in order to run)

- Medius documentation:
 - medius_overview.pdf
 - medius_libref.pdf
 - mgcl_overview.pdf
 - mgcl_libref.pdf
- The “medius_readme.txt” document, which provides a brief explanation of how to deploy the various Medius servers.
- The “medius_changes.txt” document, which lists behavior and API changes since the last release. For example, if upgrading from version 2.7 to version 2.9, you can refer to this file for a quick listing of any changes between the two versions.

Although the Medius Lobby Server executables can be enabled to work with a database versus a “simulated” mode, you must request database setup instructions directly from scert-support@scea.com. This database installation requires a licensed Oracle 9i Database Server (which is currently not offered through SCEA), preferably on a dedicated machine.

General Steps in Executing Medius API Function Calls

The Medius API uses a generalized 'request/response' message protocol when communicating with the Medius platform. The following steps provide an example of how to make a generic Medius function call:

1. Create an instance of a 'request' message data structure and populate the fields (it is recommended that you use **memset()** to set the structure to '0' and/or initialize each field to some value). All request message data structures are defined in `mediustypes.h`. Note that every request structure has a *MessageID* field. This can be populated by calling **MediusCreateMessageID()** or by using your own numbering scheme.
- Note:** The *MessageID* you assign allows you to match a response message from the Medius platform to a given request message. If your application utilizes a custom internal message queue that may or may not represent 'state' in your application, using your own numbering scheme will help identify and categorize the various messages. The SCE-RT samples usually just set this field to "1". It is up to each application to determine if they need any intrinsic value defined in the *MessageIDs*.
2. Call the appropriate Medius API function. Be sure to pass in a populated request message data structure and specify a callback function (if a response message is expected from the Medius platform). If the response is not necessary, NULL can be supplied for the callback function pointer.
3. Every Medius API function returns a *MediusErrorCode*. If *MediusErrorNone* is not returned, your application should be prepared to handle the current error condition. 'Common error' results are as follows:

- a. **MediusErrorNotInitialized (-15)** – The Medius client library has not been initialized. Call **MediusInitialize()** before making any other Medius API call.
- b. **MediusErrorNotConnected (-16)** – Either you have not called **MediusConnect()** or your connection to the MLS or MAS has been closed. Your application should go back to a reconnection state.
- c. **MediusErrorSendingMessage (-5)** – Most likely, you have an SCE-RT client library mismatch. Contact SCE-RT developer support (scert-support@scea.com) to get this resolved.

Note: If you do receive an error code, the functions described below may help interpret the code's meaning:

- a. **MediusGetErrorCodeString()** – Returns a text string description of the *MediusErrorCode* received. This function can help with the application's UI when relaying error conditions back to the online player.
- b. **MediusGetLastNetUpdateError()** – Retrieves the most recent DME **NetUpdate()** error. This may provide more low-level error condition information if **MediusUpdate()** failed (because the Medius layer is built upon the DME layer).
- c. **MediusGetCallbackStatusString()** – Returns a text string description of a *MediusCallbackStatus* error.
4. **NetGetNetUpdateErrors()** – This function fills out a **NetUpdateErrors** structure with extended error information on a per-connection index basis. When the client receives a 'response' message from the Medius platform, the callback function specified in step #2 above will be invoked. Access to the response message is provided as an input parameter to the callback function. You can extract fields from this response message data structure and process as needed.
5. Every response message data structure has a field named *StatusCode*. This field will always be populated with one of the values declared in the **MediusCallbackStatus** enumeration type defined in `mediustypes.h`. The *StatusCode* will notify the application of the success or failure of a request.

Common *StatusCode* results that any response message may generate are:

- a. **MediusSuccess (0)** – The request was successful.
- b. **MediusPlayerNotPrivileged (-989)** – The application attempted to call a Medius API function before the player's session has been fully authenticated (see "Establishing an Authenticated State").
- c. **MediusWMError (-991)** – This is generally a Medius platform server-side error condition. Contact SCE-RT developer support (scert-support@scea.com) if you experience these (they are usually caused by a server configuration issue).
- d. **MediusDBError (-988)** – The Medius platform you are communicating with has the Medius Database connection disabled, or it is having problems communicating with the Medius database. Contact SCE-RT developer support (scert-support@scea.com) for help resolving this error.

Example: Log in using an existing account with the Medius Authentication Server

```
MediusTypeAccountLoginCallback MyAccountLoginHandler(
    MediusAccountLoginResponse*pThisPacket, // Response Message
    void* pUserData) // Pointer to UserData available during callback
{
    printf("Response to MessageID %s", pThisPacket->MessageID);
    if(pThisPacket->StatusCode != MediusSuccess)
    {
        printf("Account Login Failed\n");
    }
    else
    {
        printf("Account Login Succeeded\n");
    }
    return;
}

void MyAccountLogin()
{
    MediusErrorCode retVal;

    // Populate request message data structure
    MediusAccountLoginRequest stAccountLoginRequest;

    memset(&stAccountLoginRequest, 0, sizeof(MediusAccountLoginRequest));

    // We can use either MediusCreateMessageID() or our own
    // numbering scheme.
    strcpy(stAccountLoginRequest.MessageID, "1");

    // Set the Session Key as retrieved from the session begin response
    strcpy(stAccountLoginRequest.SessionKey, g_szSessionKey);

    // Set AccountName and AccountPassword
    strcpy(stAccountLoginRequest.AccountName, g_szAccountName);
    strcpy(stAccountLoginRequest.Password, g_szPassword)

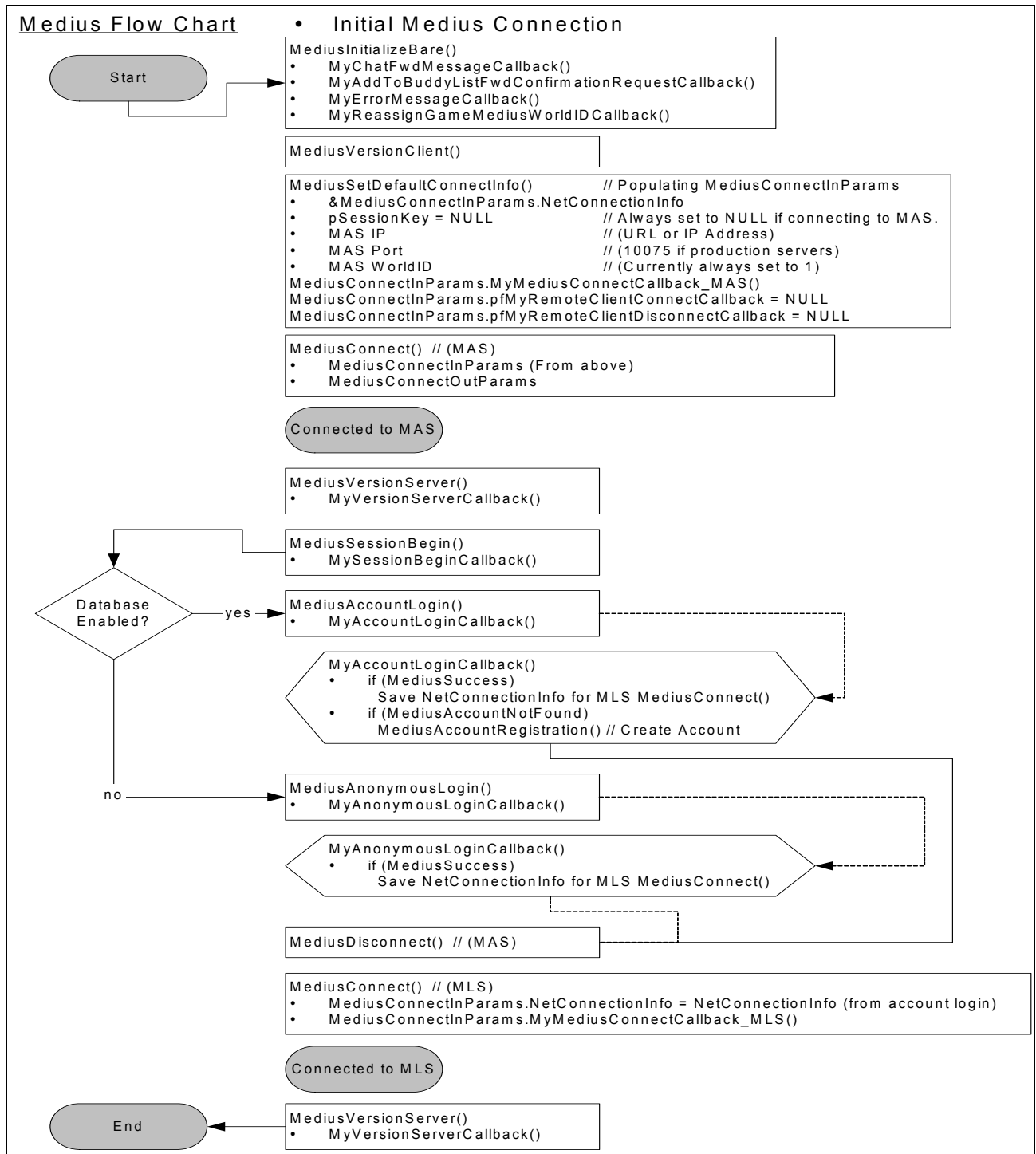
    // Execute and send the Account Login Request
    retVal = MediusAccountLogin(&stAccountLoginRequest,
                                MyAccountLoginHandler,
                                NULL); // No UserData being set
    if (retVal != MediusErrorNone)
    {
        // Return Error Message
    }
}
```

```
}  
  
// Call MediusUpdate() until a response message is received via the  
// callback handler. Call MediusUpdate() every frame if possible,  
// and definitely ASSERT() if it takes longer than 14 seconds  
// between MediusUpdate() calls (more information discussed later in  
// the Medius Overview documentation).  
  
// If you call MediusUpdate() in a while loop in this function, this will  
// block the application from doing other activities (like animating a  
// UI); therefore, it is recommended that you build a state machine so  
// that the application can do other activities while waiting for a  
// pending response message to a given request message.  
  
return;  
}
```

Connecting to the Medius Platform

Figure 1 is a high-level view of the sequence of operations that must be performed to initiate a Medius connection. Subsequent sections will refer to this diagram.

Figure 1: Initial Medius Connection Process



Initializing the Medius Client Library

You must initialize the Medius client library before proceeding with any other Medius API calls. The Medius functions associated with the initialization of Medius resources and the cleaning up/freeing of Medius resources are as follows:

- **MediusInitialize()** – Initializes Medius client library
- **MediusInitializeBare()** – Initializes Medius client library (bare version)
- **MediusClose()** – Uninitializes Medius client library
- **MediusCloseBare()** – Uninitializes Medius client library (bare version)

If you attempt to execute a Medius function before initializing the Medius client library (or after a **MediusClose()**), a **MediusErrorNotInitialized (-15)** *MediusErrorCode* value will be received. .

MediusInitialize() vs. MediusInitializeBare()

The Medius client library utilizes the messaging services provided by the DME library. Therefore it is necessary to initialize the DME by calling **NetInitialize()** before connecting to the Medius platform. Perform initialization in one of the following two ways:

- Call **MediusInitialize()** to initialize the Medius client library. An internal call to **NetInitialize()** initializes the DME library as well.
- Call **NetInitialize()** to initialize the DME library first, then call **MediusInitializeBare()** to separately initialize the Medius client library. SCE-RT recommends this approach because it provides greater flexibility.

MediusInitializeInParams

Most Medius API calls require the registration of a callback function to handle response messages sent from the Medius platform. Using callback functions gives you total control over how to process the data that is returned. The callback functions you assign as part of the **MediusInitializeInParams** (or **MediusInitializeBareInParams**) structure when calling **MediusInitialize()** (or **MediusInitializeBare()**) can easily be changed later. Simply calling the appropriate **MediusReassign*()** function will change the callback. If your application does not require a particular callback, set the callback function to NULL.

Notes about **MediusInitializeInParams**:

- **MediusTypeGenericChatFwdMessageCallback** – Callback triggered when another user calls **MediusSendGenericChatMessage()**.
- **MediusTypeBinaryFwdMessageCallback** – Callback triggered when another user calls **MediusSendBinaryMessage()**.
- **MediusTypeAddToBuddyListFwdConfirmationRequestCallback** – Callback triggered when another user calls **MediusBuddyAddConfirmation()**.
- **MediusTypeErrorMessageCallback** – Callback triggered when the Medius platform issues an Error Message. Error messages are rarely issued.
- **MediusTypeReassignGameMediusWorldIDCallback** – Callback triggered when the current game's *MediusWorldID* has changed (A host migration, for example, will trigger this callback).

MediusInitializeOutParams

This is the output response structure for **MediusInitialize()** (or **MediusInitializeBare()**). The *ErrorCode* field indicates whether initializing the Medius client library succeeded or failed.

MediusClose() vs. MediusCloseBare()

To clean up/free all Medius resources after initialization, call **MediusClose()**. Typically, you call this before exiting the application and/or when the user has exited an online game and no longer needs Medius functionality.

Match the calls as follows:

- Call **MediusClose()** if you previously called **MediusInitialize()**.
- Call **MediusCloseBare()** if you previously called **MediusInitializeBare()**.

Sample – Initializing the Medius API

```
void MyGame_MediusInitialize()
{
    // -----
    // Initialize the MEDIUS API
    // -----
    MediusErrorCode result;
    MediusInitializeBareInParams    inParams;
    MediusInitializeBareOutParams  outParams;

    memset(&inParams, 0, sizeof(MediusInitializeBareInParams));
    memset(&outParams, 0, sizeof(MediusInitializeBareOutParams));

    inParams.MyChatFwdMessageCallback = MyChatFwdMessageCallback;
    inParams.MyBinaryFwdMessageCallback = NULL;
    inParams.MyAddToBuddyListFwdConfirmationRequestCallback = NULL;
    inParams.MyErrorMessageCallback = MyErrorMsgCallback;
    inParams.MyReassignGameMediusWorldIDCallback =
        MyReassignGameMediusWorldIDCallback;

    // -----
    // Medius Initialize Bare
    // -----
    result = MediusInitializeBare(&inParams, &outParams);
    if (result)
    {
        printf("App:[%s][%d]MediusErrorCode:'%d'", __FILE__, __LINE__, result);
    }
    return;
}
```

Connecting to the MAS

A client must register and authenticate with the Medius platform before it can fully utilize “lobby” technology. To be fully logged onto the Medius platform, the following four steps must be completed:

1. Make initial connection to the Medius Authentication Server (MAS).
2. Begin a “session” with the Medius platform.
3. Establish an authenticated state (account login).
4. Disconnect from MAS, and reconnect to the Medius Lobby Server (MLS).

Medius functions used in this section:

- | | |
|--|---|
| • MediusSetDefaultConnectInfo() | – MediusConnect() helper function |
| • MediusConnect() | – Connects to the Medius server |
| • MediusDisconnect() | – Disconnects from the Medius server |
| • MediusSessionBegin() | – Starts a Medius session |
| • MediusSessionEnd() | – Ends a Medius session |
| • MediusVersionClient() | – Retrieves Medius client version information |
| • MediusGetBuildTimeStamp() | – Retrieves Medius client build timestamp information |
| • MediusVersionServer() | – Requests Medius server version information |

Making the Initial Connection to the MAS

The sequence for initiating a MAS connection is as follows:

1. Call **MediusInitialize()** or **MediusInitializeBare()**, as described in the previous section.
2. Create an instance of the **MediusConnectInParams** structure:
 - a. **memset()** the structure to '0'.
 - b. Initialize **MediusConnectInParams**'s *ConnectInfo* field with **MediusSetDefaultConnectInfo()**. Note the following about **MediusSetDefaultConnectInfo()**'s parameters:
 - The first parameter is a returned populated **NetConnectionInfo** needed by the initial **MediusConnect()** call to the MAS.
 - Always set *pSessionKey* to NULL for initial MAS connections.
 - Set *pServerIP* to the IP address of the MAS (perform a DNS lookup on the URL to obtain the IP address).
 - Set *ServerPort* to the port number on the MAS server (10075 is default for production MAS servers).
 - Always set *WorldID* to 1.
 - c. Set *MaxClientsPerConnection* to a number between 0 and 256 (This must be set to *MaxPlayersPerChannel* in *medius.txt* (256 default). *Medius.txt* is used by the MLS server to define the max number of clients per chat-channel).
 - d. Set **MyConnectCallback** to a callback function of type **MediusTypeConnectCallback**. Doing so ensures that **MediusConnect()** will operate as a non-blocking function call. Using **MediusConnect()** in a blocking fashion is no longer supported as of the SCE-RT 2.9 Release.
 - e. Set **pfRemoteClientConnectCallback** to NULL.
 - f. Set **pfRemoteClientDisconnectCallback** to NULL.

Note 1: Anytime you execute non-blocking function calls (which all Medius API calls are), you will need to define a state-machine to represent your current state (in this case 'lobby state'). This will allow the program to process other application-level tasks while waiting for a response message from the Medius platform indicating the success or failure of your request. Be sure that your state-machine manages calls to **MediusUpdate()** each frame until the callback is received. Based upon the success or failure of the request call, transition through your application's state-machine to determine what to do next. The SCE-RT sample 'DmePoolBalls' attempts to build up one such state-machine, but you probably will define your own based on your application needs. The SCE-RT++ Wrapper SDK, new to the SCE-RT v2.9 official release, is a fully qualified Medius state-machine. The SCE-RT++ Wrapper is included in the samples directory.

3. Call **MediusConnect()**, referencing the appropriate Medius Authentication Server (MAS) *IP/Port/WorldID* address information as described in the **MediusConnectInParams** set above. The client's Medius connection index (*ConnectionIndex*) will be set to a value between (0..n) as seen in **MediusConnectOutParams**.

4. Verify the return code for **MediusConnect()** was set to *MediusErrorNone*. Note that the application's state-machine is responsible for 'timing out' **MediusConnect()**'s pending response callback (30 seconds is a common threshold). Be sure to call **MediusUpdate()** while your application is waiting for the response callback. If a timeout occurs, the application's state-machine should transition to a reconnect state.

Note 2: If **MediusConnect()** failed to create a socket and establish a connection with a MAS socket, then **MediusConnect()** will return **MediusErrorConnecting (-2)** immediately upon returning. If *MediusErrorNone* was returned, the application will be in a state where it is waiting for the connect callback to be triggered. The connect callback will occur after a number of synchronizing 'handshakes' have taken place. As stated above, if your connect timeout threshold was exceeded, present the user an error message such as, "Connection timed out and/or failed to connect to this address", then transition to a state where you can attempt to reconnect.

5. Once the connect callback is triggered, verify that the *Connection* pointer (*HDME handle to the DME*) is != NULL. If the *Connection* pointer is == NULL, then there was an error connecting to the MAS (that is, the synchronizing handshakes with the MAS did not resolve into a connected state). Set "success" or "failure" to your state-machine here. If the result is "success", there is now a connection to the MAS and you are ready to start a Medius session.

Note 3: Currently, there is no descriptive information (response message data structure) associated with **MediusTypeConnectCallback**. Nonetheless, you can still determine why a connection failed by calling **MediusGetLastNetUpdateError()** and/or by looking at MAS log files. If **MediusConnect()** returns *MediusErrorNone*, you know that at least a socket connection was made. If the *Connection* pointer is == NULL in the connect callback, it may be associated with a condition such as "trying to connect to a secure server while having no client security key setup". If you receive errors such as this, call SCE-RT developer support.

Sample – Connecting to the MAS

```
// -----
// Connecting to the MAS
// -----
// (Warning) - This code sample is not production quality. It does not manage
// an application-level lobby 'State-Machine'. See SCE-RT samples for a more
// complete sample

void MyConnectCallback_MAS(HDME Connection, void* pUserData)
{
    if (Connection != NULL)
    {
        // Success
        // As a design note, pUserData could set state in your state-machine that
        // flagged success or failure
    }

    g_RtLobby.m_bConnectCallbackComplete_MAS = TRUE;
    return;
}

MyGame_ConnectTo_MAS()
```

```

{
    MediusErrorCode result;
    MediusConnectInParams  inParams;
    MediusConnectOutParams outParams;

    memset(&inParams, 0, sizeof(MediusConnectInParams));
    memset(&outParams, 0, sizeof(MediusConnectOutParams));

    // -----
    // Set MediusConnectInParams
    // -----
    MediusSetDefaultConnectInParams(&inParams.ConnectInfo, // NetConnectInfo (Out)
        NULL, // For initial connection pass in NULL for pSessionKey
        "www.mygame.com", // Authentication ServerIP
        10075, // Authentication ServerPort
        1); // Authentication WorldID (1 by default)
    inParams.MaxClientsPerConnection = 256; // Max Clients per Chat-Channel
    inParams.MyConnectCallback = MyConnectCallback_MAS; // Sets as non-blocking
    inParams.pfRemoteClientConnectCallback = NULL;
    inParams.pfRemoteClientDisconnectCallback = NULL;
    memset(&inParams.ConnectInfo.ServerKey, 0, sizeof(RSA_KEY)); //Security Disabled

    // -----
    // Medius Connect (MAS)
    // -----
    result = MediusConnect(&inParams, &outParams, NULL);
    if (result != MediusErrorNone)
    {
        printf("App:[%s][%d]MediusError:'%d'\n", __FILE__, __LINE__, result);
    }
    // -- OutParams --
    printf(" OutParams->\n");
    printf(" NetErrorCode:          '%d'\n", outParams.ErrorCode);

    // =====
    // Wait for MediusConnect to complete
    // =====
    do
    {
        // Need timeout logic
        MediusUpdate();
    } while (g_RtLobby.m_bConnectCallbackComplete_MAS == FALSE);

    return;
}

```

Starting a Session

Once there is a connected state with the MAS, begin a “session” with the Medius platform by calling **MediusSessionBegin()**. The MAS will then manage the creation of a ‘player object’ with the Medius Universe Manager (MUM). If the transaction was successful, a unique session key will be generated and returned to the client in the *SessionKey* field of the **MediusSessionBeginResponse** message. This session key associates the client with the ‘player object’ managed by the MUM. This session key must also be used in all subsequent Medius ‘request’ messages until the user entirely disconnects from the Medius platform.

Note: A session may end by either calling **MediusSessionEnd()** (usually followed by **MediusDisconnect()**) or by a case where the client had not called **MediusUpdate()** within a timely fashion. The Medius platform typically will end a ‘player object’ if **MediusUpdate()** had not been called for 30 seconds (though this 30 second interval is configurable the Medius servers). **MediusUpdate()** (even if no data is being transmitted) services the connection to the Medius platform. If a client’s ‘player object’ ends or times out, the client will need to reconnect to the MAS and re-authenticate.

The sequence for initiating a “session” is as follows:

1. Create an instance of the **MediusSessionBeginRequest** structure.
 - a. **memset()** the structure to ‘0’.
 - b. Set *MessageID* to a value of your numbering scheme (if needed).
 - c. Set *ConnectionClass* to a **MediusConnectionType**.

Note: Based on whatever network hardware device configuration you selected during the network combination screen, match that type with **MediusConnectionType**. This information is valuable when determining “how many Modem users are there versus Broadband users?” This information is available on the MUM server.

2. Call **MediusSessionBegin()** with the respective request message and callback.
3. When the **MediusSessionBeginResponse** message is received, if *StatusCode*=**MediusSuccess** then save off the *SessionKey* for later use.

If *StatusCode*!=**MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the common set):

- a. **MediusBeginSessionFailed (-1000)** – The Medius platform could not create a session for the current player. Be sure you are connected to the MAS server.

Sample – Starting a Session

```
// -----
// Starting a Session
// -----
// (Warning) - This code sample is not production quality. It does not manage
// an application level lobby 'State-Machine'. See SCE-RT samples for a more
// complete sample

void MySessionBeginCallback(MediusSessionBeginResponse* pResponse,
                           void* pUserData)
{
    switch (pResponse->StatusCode)
    {
    case MediusSuccess:
        // Save of the 'player object's SessionKey
        strncpy(g_RtLobby.m_SessionKey,
                pResponse->SessionKey,
                SESSIONKEY_MAXLEN-1);
        g_RtLobby.m_SessionKey[SESSIONKEY_MAXLEN-1]='\0';
        break;
    default:
        break;
    }
}
```

```

    }

    g_RtLobby.m_bSessionBeginComplete = TRUE;
    return;
}

void MyGame_SessionBegin()
{
    MediusErrorCode result;
    MediusSessionBeginRequest request;

    memset(&request, 0, sizeof(MediusSessionBeginRequest));

    // -----
    // Set MediusSessionBeginRequest
    // -----
    strcpy(request.MessageID, "1");
    request.ConnectionClass = Ethernet; // Todo: set based on "eznetcfg"

    // -----
    // Medius Session Begin
    // -----
    result = MediusSessionBegin(&request, MySessionBeginCallback, NULL);
    if (result != MediusErrorNone)
    {
        printf("App:[%s][%d]MediusError:'%d'\n", __FILE__, __LINE__, result);
    }

    // =====
    // Wait for SessionBegin to complete
    // =====
    do
    {
        // Need timeout logic
        MediusUpdate();
    } while (g_RtLobby.m_bSessionBeginComplete == FALSE);

    return;
}

```

Version Information and Build Timestamps

The Medius API provides methods to gain access to the following information:

- Medius client library version
- Medius client library build time stamp
- Currently connected MAS version or MLS version

This information is very helpful when contacting SCE-RT developer support (scert-support@scea.com) and it is highly recommended that you integrate it in your application's code base.

To retrieve the version of the Medius client library, pass in a string of 64 characters to **MediusVersionClient()** that will then be populated with the version string.

To retrieve the build timestamp of the Medius client library, pass in a string of 64 characters to **MediusGetBuildTimeStamp()** that will then be populated with the build timestamp string.

To retrieve the currently connected version of the MAS or MLS, instantiate and populate a **MediusVersionServerRequest** structure for use with **MediusVersionServer()**. The Medius platform (MAS or MLS) will then send a **MediusVersionServerResponse** message to the client with populated server version information.

Sample – Version Information and Build Timestamps

```

// -----
// Version Information and Build Timestamps
// -----
MediusVersionClient(g_RtLobby.m_MediusVersion);
MediusGetBuildTimeStamp(g_RtLobby.m_MediusBuildInfo);
NetGetClientVersion(g_RtLobby.m_DmeVersion);
NetGetBuildTimeStamp(g_RtLobby.m_DmeBuildInfo);

printf("->Medius Client Version:  '%s'\n", g_RtLobby.m_MediusVersion);
printf("->Medius Build Stamp:      '%s'\n", g_RtLobby.m_MediusBuildInfo);
printf("->DME Client Version:      '%s'\n", g_RtLobby.m_DmeVersion);
printf("->DME Build Stamp:         '%s'\n", g_RtLobby.m_DmeBuildInfo);

void MyGetMediusServerVersionCallback(MediusVersionServerResponse* pResponse,
                                      void* pUserData)
{
    switch (pResponse->StatusCode)
    {
        case MediusSuccess:
            // -----
            // Get Medius Server Version was a Success
            // -----
            printf("->MAS or MLS Server Version:  '%s'\n", pResonse->VersionServer);
            break;
        default:
            break;
    }

    g_RtLobby.m_bGetMediusServerVersionComplete = TRUE;
    return;
}

void MyGame_GetMediusServerVersion()
{
    MediusErrorCode result;
    MediusVersionServerRequest request;

    memset(&request, 0, sizeof(MediusVersionServerRequest));

    // -----
    // Set MediusVersionServerRequest
    // -----
    strcpy(request.MessageID, "1");
    strcpy(request.SessionKey, g_RtLobby.m_SessionKey);

    // -----
    // Medius Server Version
    // -----
    result = MediusServerVersion(&request, MyGetMediusServerVersionCallback, NULL);
    if (result != MediusErrorNone)
    {
        printf("App:[%s][%d]MediusError:'%d'\n", __FILE__, __LINE__, result);
    }
}

```

```

    }

    // =====
    // Wait for Server Version to complete
    // =====
    do
    {
        // Need timeout logic
        MediusUpdate();
    } while (g_RtLobby.m_bGetMediusServerVersionComplete == FALSE);

    return;
}

```

Establishing an Authenticated State

Before establishing an Authenticated State (account login), first refer to both the “Platform Requirement Functionality” section (discussing usage policy, privacy policy, and announcements) and the “Co-location” section (discussing Medius platform deployment strategies). Your application may bypass this functionality during early development, but you will need to take these items into consideration before your application is submitted to QA/Format and Production.

Once a ‘session’ (player object) is created, the session needs to be in an authenticated state. You can register a new account (or log in with an existing account) or have the client establish an anonymous login (applications will typically do one or the other). If the account login (or anonymous login) is successful, the session will then be in an authenticated state.

The Medius functions associated with establishing an authenticated state are as follows:

- **MediusAccountRegistration()** – Creates a new account
- **MediusAccountLogin()** – Logs into an account
- **MediusAnonymousLogin()** – Anonymous login (account exists for the lifetime of the session)
- **MediusAccountLogout()** – Logouts out of account

Registering a New Account

Populate the **MediusAccountRegistrationRequest** structure and call **MediusAccountRegistration()**. Currently only *AccountName* and *Password* are required fields. Account names must be unique. It is possible that some game applications will hide much of the registration process from the player. When registering a new Medius Account, it is required that the username and password be Unified Community (UC) compliant.

Unified Community

The Unified Community feature set for SCE-RT was born out of the necessity to bring online titles into compliance with the efforts and mandate of the SCEI Corporate Initiative Identity Federation (IDF) for unifying accounts across all game universes, and ultimately providing a central lobby for gamers that will provide community, news, etc.

The key to any unified community comes down to a simple core requirement – a standardized account name and password specification.

All titles that adhere to the specification for a UC-compatible account name and password will ultimately be compliant with the SCE-RT UC, and eventually the IDF. SCE-RT has created this requirement without changing any of the existing APIs for online game integration.

In addition, having a centralized account, SCE-RT now provides for a central buddy list, ignore list and cross-title instant messaging. Again, there are no API changes required to implement these features, however there will be some UI considerations for your development teams. You should

consult closely with SCE-RT (scert-support@scea.com) to discuss how to implement these features from a UI perspective.

Encoding

UTF-8 – In this instance, the printable subset of UTF-8 commonly known as US-ASCII with certain exclusions

Account Name Specification

Character Set – A-Z, a-z, 0-9, (dash) -, (period) ., (underscore) _
 Length – 5 characters min. (5 + NULL), 16 characters max. (15 + NULL)
 Case – Account names are case insensitive

Password Specification

Character Set – A-Z, a-z, 0-9, (dash) -, (period) ., (underscore) _
 Length – 4 characters min. (4 + NULL), 16 characters max. (15 + NULL)
 Case – Passwords are case insensitive

All other characters are restricted. If *AccountName* and *Password* specifications are not met when connecting to a 2.10 SCE-RT server, **MediusAccountRegistrationRequest** will fail.

The sequence for registering a new account is as follows:

1. Create an instance of the **MediusAccountRegistrationRequest** structure.
 - a. **memset()** the structure to '0'.
 - b. Set *MessageID* to a value of your numbering scheme (if needed).
 - c. Set *SessionKey* to the value you saved from **MediusSessionBeginResponse**.
 - d. Set *AccountType* to a *MediusAccountType* (currently only *MediusMasterAccount* is supported).
 - e. Set *AccountName* to the string provided by the client.
 - f. Set *Password* to the string provided by the client.

2. Call **MediusAccountRegistration()** with the respective request message and callback.
3. If *StatusCode*=**MediusSuccess** when the **MediusAccountRegistrationResponse** message is received, the account has been created and you can proceed to call **MediusAccountLogin()**.

Other notes about **MediusAccountRegistrationResponse**:

- a. *AccountID* – Primary key associated with player's account information in the Medius database. Numerous Medius API calls use the *AccountID* to direct messages to other clients and request client information (search for *AccountID* in the Medius API for further uses). *AccountID* will always be > 0 for accounts that are persisted in the Medius database.
4. If *StatusCode*!=**MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. If you are having trouble registering, be sure that your *AccountName* and *Password* are Unified Community-compliant. Possible error conditions (in addition to the common set):
 - a. **MediusAccountAlreadyExists (-999)** – *AccountName* was not unique.

Account Login

A player must log into an account (thus setting the user's session to an authenticated state) to play online. Any player that does not have an existing account must register a new account. A player that already has an account can bypass registration and call **MediusAccountLogin()** directly.

The sequence for performing an account login is as follows:

1. Create an instance of the **MediusAccountLoginRequest** structure.
 - a. **memset()** the structure to '0'.
 - b. Set *MessageID* to a value of your numbering scheme (if needed).

- c. Set *SessionKey* to the value you saved from **MediusSessionBeginResponse**.
- d. Set *AccountName* to the string provided by the client.
- e. Set *Password* to the string provided by the client.
2. Call **MediusAccountLogin()** with the respective request message and callback.
3. If *StatusCode*=**MediusSuccess** when the **MediusAccountLoginResponse** message is received, the login to the Medius platform is successful and the session will be in an authenticated state.

Other notes about **MediusAccountLoginResponse**:

- a. **AccountID** – (Value > 0) Same value from **MediusAccountRegistrationResponse**.
- b. **AccountType** – As set from **MediusAccountRegistrationRequest**.
- c. **MediusWorldID** – This is the default ‘Chat Channel’ ID reserved upon the pending connection to the MLS. After the connection to the MLS is complete, you can call **MediusGetChannelInfo()** to get information about the currently connected chat channel.
- d. **ConnectInfo** – (Very Important) This is the *NetConnectionInfo* needed to connect to the MLS, which is the next step.
4. If *StatusCode*!=**MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (in addition to the common set):
 - a. **MediusLoginFailed (-995)** – General account login failure.
 - b. **MediusAccountNotFound (-998)** – The account name was invalid. User should be prompted to either register a new account or to attempt to log in with a new account name.
 - c. **MediusInvalidPassword (-978)** – The password was invalid to a valid account name.
 - d. **MediusAccountLoggedIn (-997)** – Either someone has already logged in with this account or the player ungracefully disconnected from the Medius platform and must wait approximately two minutes for the authenticated session (player object) to completely timeout.
 - e. **MediusFilterFailed (-967)** – Login failed because the *AccountName* did not pass the automatic vulgarity filter check.

If a user decides to log in with a different account or wishes to create a new account after their session has already been authenticated, they will need to call **MediusAccountLogout()** to set the session back to an unauthenticated state. Once back in an unauthenticated state, the application should disconnect from the MLS and reconnect to the MAS (if not already connected to the MAS). However, when gracefully disconnecting from the Medius platform (in general), you do not need to worry about calling **MediusAccountLogout()** because the **MediusSessionEnd()** call will fully log out the player (see “Disconnecting From the Medius Platform”).

Sample – Account Login and Registering a New Account

```
// -----
// Account Login and Registering a New Account if 'AccountNotFound'
// -----
// (Warning) - This code sample is not production quality. It does not manage
// an application-level lobby 'State-Machine'. See SCE-RT samples for a more
// complete sample

void MyAccountCreateCallback(MediusAccountRegistrationResponse* pResponse,
                             void* pUserData)
{
    switch (pResponse->StatusCode)
    {
    case MediusSuccess:
        // Account registration was a success
        break;
    default:
        break;
    }
}
```

```

    g_RtLobby.m_bAccountCreateComplete = TRUE;
    return;
}

void MyGame_AccountCreate()
{
    MediusErrorCode result;
    MediusAccountRegistrationRequest request;

    memset(&request, 0, sizeof(MediusAccountRegistrationRequest));

    // -----
    // Set MediusAccountRegistrationRequest
    // -----
    strcpy(request.MessageID, "1");
    strcpy(request.SessionKey, g_RtLobby.m_SessionKey);
    request.AccountType = MediusMasterAccount;
    strcpy(request.AccountName, "MyAccountName");
    strcpy(request.Password, "MyAccountPassword");

    // -----
    // Medius Account Registration
    // -----
    result = MediusAccountRegistration(&request, MyAccountCreateCallback, NULL);
    if (result != MediusErrorNone)
    {
        printf("App:[%s][%d]MediusError:'%d'\n", __FILE__, __LINE__, result);
    }

    // =====
    // Wait for Account Registration to complete
    // =====
    do
    {
        // Need timeout logic
        MediusUpdate();
    } while (g_RtLobby.m_bAccountCreateComplete == FALSE);

    return;
}

void MyAccountLoginCallback(MediusAccountLoginResponse* pResponse,
                           void* pUserData)
{
    switch (pResponse->StatusCode)
    {
    case MediusSuccess:
        // -----
        // Account login was a success; therefore, save the
        // NetConnectInfo to the MLS for the upcoming redirected
        // MediusConnect() call to the MLS.
        // You may want to save the MediusWorldID here also - this is
        // the MediusWorldID of the default chat channel that has been
        // reserved for you upon a successful connection to the MLS.
        // -----
        memcpy( &(g_RtLobby.m_LobbyConnectInfo),
                &(pResponse->ConnectInfo),
                sizeof(g_RtLobby.m_LobbyConnectInfo));
        break;
    case AccountNotFound:

```

```

        // -----
        // Account Create Request
        // -----
        MyGame_AccountCreate();
    default:
        break;
}

    g_RtLobby.m_bAccountLoginComplete = TRUE;
    return;
}

void MyGame_AccountLogin()
{
    MediusErrorCode result;
    MediusAccountLoginRequest request;

    memset(&request, 0, sizeof(MediusAccountLoginRequest));

    // -----
    // Set MediusAccountLoginRequest
    // -----
    strcpy(request.MessageID, "1");
    strcpy(request.SessionKey, g_RtLobby.m_SessionKey);
    strcpy(request.AccountName, "MyAccountName");
    strcpy(request.Password, "MyAccountPassword");

    // -----
    // Medius Account Login
    // -----
    result = MediusAccountLogin(&request, MyAccountLoginCallback, NULL);
    if (result != MediusErrorNone)
    {
        printf("App:[%s][%d]MediusError:'%d'\n", __FILE__, __LINE__, result);
    }

    // =====
    // Wait for Account Login to complete
    // =====
    do
    {
        // Need timeout logic
        MediusUpdate();
    } while (g_RtLobby.m_bAccountLoginComplete == FALSE);

    return;
}

```

Anonymous Logins

There is an alternative to the traditional “register/login” method. Many game applications will not perform any type of data collection. These applications will instead use “anonymous” logins. An anonymous login does not require the above-mentioned **MediusAccountRegistration()** process. Rather, the **MediusAnonymousLogin()** should be performed after **MediusSessionBegin()**. The response will be very similar to that of a normal **MediusAccountLogin()**. Connection details for the Medius Lobby Server are provided, along with a temporary *AccountID* (< 0) that can be used for the life of the session.

There are several key limitations with anonymous logins:

- No database storage
- All account information lasts only for the client session
- No ladder support

- No buddy list support
- No clan support

The sequence for performing an anonymous account login is as follows:

1. Create an instance of the **MediusAnonymousLoginRequest** structure.
 - a. **memset()** the structure to '0'.
 - b. Set *MessageID* to a value of your numbering scheme (if needed).
 - c. Set *SessionKey* to the value you saved from **MediusSessionBeginResponse**.
 - d. Set *SessionDisplayName* to the string provided by the client. This will be the anonymous player's name for as long as they are connected. Does not need to be unique; therefore, you may potentially see other players with the same screen name.
 - e. Set *SessionDisplayStats* to an (optional) string provided by the application. Each application will decide if and how they wish to use this field. For example, if your application has custom avatars that represent each client in the lobby, you could encode that information here
2. Call **MediusAnonymousLogin()** with the respective request message and callback.
3. If *StatusCode*=**MediusSuccess** when the **MediusAccountLoginResponse** message is received (note that **MediusAnonymousLogin()** shares the same callback and response message type as **MediusAccountLogin()**), the login to the Medius platform is successful and the session will be in an authenticated state.

Other notes about **MediusAccountLoginResponse**:

- a. *AccountID* – (Value < 0) Temporary value that exists for the lifetime of the player's session.
 - b. *AccountType* – Ignore (set to *MediusMasterAccount*).
 - c. *MediusWorldID* – This is the default 'Chat Channel' ID reserved upon the pending connection to the MLS. After the connection to the MLS is complete, you can call **MediusGetChannelInfo()** to get information about the currently connected chat channel. .
 - d. *ConnectInfo* – (**Very Important**) This is the *NetConnectionInfo* needed to connect to the MLS, which is the next major step.
4. If *StatusCode*!=**MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (in addition to the common set):
 - a. **MediusLoginFailed (-995)** – General anonymous login failure.
 - b. **MediusFilterFailed (-967)** – Login failed because the *SessionDisplayName* did not pass the automatic vulgarity filter check.

Reconnecting to the MLS

Once the 'session' (player object) is in an authenticated state you can disconnect from the MAS, reconnect to the MLS, and fully utilize the "lobby" technology that Medius provides.

All **Notes (1–4)** in the 'Connecting to the MAS' section also apply to establishing a connection to the MLS. The sequence for reconnecting to the MLS is as follows:

1. Call **MediusDisconnect()** to disconnect from the MAS (this disconnects the Medius connection).
2. Create an instance of the **MediusConnectInParams** structure:
 - a. **memset()** the structure to '0'.
 - b. Set *ConnectInfo* to the *NetConnectionInfo* you saved from **MediusAccountLoginResponse**.
 - c. Set *MaxClientsPerConnection* to a number between 0 and 256 (This must be set to *MaxPlayersPerChannel* in *medius.txt* (256 default). *Medius.txt* is used by the MLS server to define the max number of clients per chat-channel).
 - d. Set *MyConnectCallback* to a callback function of type **MediusTypeConnectCallback**. By doing so, **MediusConnect()** will operate as a non-blocking function call. Using **MediusConnect()** in a blocking fashion is no longer supported as of the SCE-RT 2.9 Release.

- e. Set **pfRemoteClientConnectCallback** to NULL.
- f. Set **pfRemoteClientDisconnectCallback** to NULL.
3. Call **MediusConnect()** referencing the appropriate Medius Lobby Server (MLS) *IP/Port/WorldID* address information as described in the *MediusConnectInParams* set above.
4. Verify the return code for **MediusConnect()** was set to *MediusErrorNone*. The application's state-machine is responsible for 'timing out' **MediusConnect()** pending response callback (30 seconds is common). Be sure to call **MediusUpdate()** while your application is waiting for the response callback. If a timeout occurs, the application's state-machine should transition to a reconnect state.
5. Once the connect callback is triggered, verify that the *Connection* pointer (*HDME handle to the DME*) is != NULL. If the *Connection* pointer is == NULL, then there was an error connecting to the MLS (that is, the synchronizing handshakes with the MLS did not resolve into a connected state). Set "success" or "failure" to your state-machine here. If the result is "success", there is a connection to the MLS and you are ready to fully utilize Medius' "lobby" technology.
6. **Very Important:** A **MediusConnect()** call to the MLS will place a player in a default 'chat channel'; therefore, the first thing you should do after a connect callback to the MLS returns success is call **MediusUpdateClientState()** with *UserAction* set to the *MediusUserAction* type 'JoinedChatWorld'. You need to do this even if your application is not aware of chat channels.

Summary: At this point the player is fully logged in with the MLS, connected to a default lobby chat channel (whether your application utilizes chat functionality or not), and can proceed to use the full suite of Medius functions. The next section discusses the issues involved with both disconnecting and being disconnected from the Medius platform. A following section illustrates managing Medius reports to ensure connection to the Medius platform is maintained.

Sample – Connecting to the MLS

```
// -----
// Connecting to the MLS
// -----
// (Warning) - This code sample is not production quality. It does not manage
// an application-level lobby 'State-Machine'. See SCE-RT samples for a more
// complete sample

void MyConnectCallback_MLS(HDME Connection, void* pUserData)
{
    if (Connection != NULL)
    {
        // (Success)
        // As a design note, pUserData could set state in your state-machine that
        // flagged success or failure
    }

    g_RtLobby.m_bConnectCallbackComplete_MLS = TRUE;
    return;
}

MyGame_ConnectTo_MLS()
{
    MediusErrorCode result;
    MediusConnectInParams inParams;
    MediusConnectOutParams outParams;
    MediusDisconnectParams dp;
    MediusUpdateUserState state;

    memset(&inParams, 0, sizeof(MediusConnectInParams));
    memset(&outParams, 0, sizeof(MediusConnectOutParams));
    memset(&state, 0, sizeof(MediusUpdateUserState));
}
```

```

// Disconnect from MAS
result = MediusSetDefaultDisconnectParams(&dp);
dp.MyDisconnectCallback = NULL; // NULL is the blocking version
dp.DisconnectReason = NetDisconnectNormal; //Reason for disconnect

result = MediusDisconnect( &dp, NULL );

// -----
// Set MediusConnectInParams
// -----
inParams.ConnectInfo = g_RtLobby.m_LobbyConnectInfo; // Set from
                                                    // AccountLoginReponse

inParams.MaxClientsPerConnection = 256;
inParams.MyConnectCallback = MyConnectCallback_MLS; // Sets as non-blocking
inParams.pfRemoteClientConnectCallback = NULL;
inParams.pfRemoteClientDisconnectCallback = NULL;

// -----
// Medius Connect (MLS)
// -----
result = MediusConnect(&inParams, &outParams, NULL);
if (result != MediusErrorNone)
{
    printf("App:[%s][%d]MediusError:'%d'\n", __FILE__, __LINE__, result);
}
// -- OutParams --
printf(" OutParams->\n");
printf(" NetErrorCode:           '%d'\n", outParams.ErrorCode);

// =====
// Wait for MediusConnect to complete
// =====
do
{
    // Need timeout logic
    MediusUpdate();
} while (g_RtLobby.m_bConnectCallbackComplete_MLS == FALSE);

// *****
// Note: VERY IMPORTANT TO DO THIS
// *****
// Set 'player object' to a state where it is ready to perform 'lobby'
// functionality.
strncpy(state.SessionKey, g_RtLobby.m_SessionKey, SESSIONKEY_MAXLEN-1);
state.SessionKey[SESSIONKEY_MAXLEN-1]='\0';
state.UserAction = JoinedChatWorld; // ←
MediusUpdateClientState(&state);

return;
}

```

Disconnecting From the Medius Platform

A client can disconnect gracefully or ungracefully from the Medius platform. Disconnection may be necessary and/or may occur for a number of reasons, including:

- Conclusion of the online game (and possibly returning to the application's offline features)
- Logon with a different account
- The occurrence of a fatal network error
- The need for a PlayStation®2 power cycle

Graceful Disconnects

The sequence for initiating a graceful Medius platform disconnect is:

1. Call **MediusSessionEnd()** to remove the server-side 'player object'.
2. Call **MediusDisconnect()** to disconnect from the MLS or MAS. Be sure to specify the **NetDisconnectReason**.

A graceful disconnect from the Medius platform is recommended whenever possible, especially when a client encounters a fatal network error. Instead of requiring the user to power cycle the PlayStation®2, the application should gracefully disconnect (if possible) and allow the user to reconnect with the same information provided initially.

Note: A player who disconnects ungracefully (for example, a power cycle due to application lock-up), then attempts to reconnect and perform a **MediusAccountLogin()** will receive a status code indicating *MediusPlayerAlreadyLoggedIn*. The player's account will timeout in approximately 60–90 seconds.

Ungraceful Disconnects

If you are new to the Medius API, you may find that detecting the cause of ungraceful disconnects is sometimes challenging.

After a connection to the MAS or the MLS has been established, you must call **MediusUpdate()** (every frame if possible) to update/service the Medius connection and determine if incoming or outgoing messages need processing. If for some reason you do not call **MediusUpdate()** within 30–45 seconds of a previous call, the MLS or MAS will automatically close your Medius connection.

Special Note: SCE-RT client libraries are built for a number of platforms. If you are developing for the PlayStation®2 you must consider that the PlayStation®2 DME client library utilizes a 'high performance counter' on the PlayStation®2 (the MIPS5900 294Mhz timer 'register c0'). This timer utilizes a 32-bit value that fills up within approximately 14 seconds; the software needs the opportunity to reset the timer before it rolls over (that is, calls **NetUpdate()**). Thus, a PlayStation®2 online client must call **MediusUpdate()** within 14 seconds to ensure all internal timing calculations are correct; if not, internal timing functions may become ineffective and undefined behavior may result. Although your application may appear fine, the server-side timeout will occur in 30-45 seconds. For debugging builds, you may have those 30–45 seconds to work with. However, Format/QA release builds must adhere to the 14-second rule. It is highly recommended that you wrap/encapsulate the **MediusUpdate()** and **NetUpdate()** functions and let your application keep track of the elapsed time between these calls (with either **NetGetLocalTime()** or your own timing function). If your application detects an elapsed time > 14 seconds then **ASSERT()**.

The following are common reasons why **MediusUpdate()** may not have been called within 14 seconds on a client PlayStation®2:

- Hit a break point in the debugger. You have 14 seconds to resume before a disconnect takes place. You may need to devise a creative way to interpret network data in real-time (perhaps visually or in a log file).
- The application is blocking due to a long load time off the DVD. If you find such long load times occurring, you may want to either place **MediusUpdate()** calls between resource loads or spin off a separate thread during these heavy loads and call **MediusUpdate()** from there.
Note: If you end up with two threads that call Medius API (or DME API) functions, note that these APIs are not thread-safe; therefore, it is recommended that your second thread only calls **MediusUpdate()**. You may need your own semaphore (or state) in your application that prevents Medius or DME API calls from being made in the first thread while the second thread is running.
- Somehow your application reached a state where it is no longer calling **MediusUpdate()**. As long as your game is online, you must call **MediusUpdate()**. If your application is “In Game”, but you failed to call **MediusUpdate()**, you may still play your game but your Medius connection will be severed and you will not be able to return to the application’s Lobby when the current game has finished.
- If the MLS or MAS has closed your connection (the Medius connection always being a ‘TCP socket’), and you call **MediusUpdate()**, you will see this TTY error message reported from the DME’s messaging layer:

```
RECV/IP: sceInetRecv() failed, error -508, sceINETE_CONNECTION_DOES_NOT_EXIST,
no connection was established or connection has been closed
```

MediusUpdate() will return -10 (*MediusErrorGeneral*) while MLS or MAS log files will report this disconnect as:

```
00:00:00 TIME_CHECK World 1, Client 0, timeout value 30000, elapsed time 30
00:00:00 RT_MSG_SERVER rt_msg_server_client_disconnect message: world 1, client
0, reason 4
```

Server Time

The **MediusGetServerTime()** function returns the server's local time and time zone from any production server. This includes the Linux versions of the Medius Authentication Server (MAS), Medius Lobby Server (MLS), and the Medius Universe Information Server (MUIS). Getting the server time is currently not available for the Win32 versions of the Medius servers. The time returned is in seconds starting with Jan 1, 1970.

The server time is also sent to the client application via the *TimeStamp* field of a **MediusGenericChatFwdMessage**. This means that any time a chat message is received by the client application, the server time will be available.

What is a 'World'?

The SCE-RT SDK is focused on the partition of 'worlds' on any given server (MUM, MAS, MLS, MPS, GS, and peer-to-peer 'hosts'). This allows more optimized bandwidth loading and cost-effective use of server hardware, as multiple applications can run multiple game instances on a single set of servers, depending on the demands of the applications.

An SCE-RT server operates multiple 'worlds' simultaneously, and generally polls each world for activity one-at-a-time. Every world is given equal priority, whether for two users/connections or 256 users/connections.

The Medius API refers to different *WorldID* types, sometimes within the same function, which may cause confusion. The major types of *WorldIDs* that you should know about are:

- **MediusWorldID** (lobby world objects "chat channels") – This is an index number that refers to a chat channel that is unique across the entire Medius platform infrastructure. No two chat channels can have the same *MediusWorldID*, whether or not they're on the same MLS.
- **MediusWorldID** (game world objects) – This is the same concept as chat channel *MediusWorldIDs*. No two 'game world objects' will ever have the same *MediusWorldID*, as this is a unique identifier for that game's instance represented on the Medius platform. A chat channel and a 'game world object' instance may have matching *MediusWorldIDs*, but this does not reflect any correlation between the two worlds, as they are kept and managed in two separate sets of indexes.
- **WorldID** (DME 'game world') – This is the "physical" *WorldID* a player is actually connected to on either a GS or peer-to-peer 'host'. Traditionally, a server is thought of as having an IP Address and a Port Number. This has been extended into a third dimension with *WorldID*, where you connect to a discrete 'game world' on the given IP/Port, and cannot directly interact with any other 'game worlds' on the same server.

The next section discusses the various SCE-RT Network Topologies and should help you visualize the concept of 'world' management.

Reporting (Maintaining Medius Platform Connections and Objects)

There are four types of reports: **Server Reports**, **World Reports**, **Player Reports**, and **End Game Reports**. Each report type is discussed in detail in this section.

The Medius platform relies on a variety of periodic report messages to manage its in-memory object collection database (MUM Object Collections) of online **game server objects**, **game world objects**, and **player objects**. To stress the importance of reporting (because failure to make these update calls and/or reports may result in the Medius platform closing your connections) this section touches upon a number of topics before officially introducing them (such as creating/joining a game, and the MGCL API).

The Medius (and MGCL) functions detailed in this section are:

- **MediusUpdate()** – Maintains Medius connection
- **MediusSendWorldReport()** – Updates game information
- **MediusSendEndGameReport()** – Informs Medius that the game has ended
- **MediusSendPlayerReport()** – Updates player information
- **MediusUpdateClientState()** – Updates player state
- **MGCLUpdate()** – Maintains MGCL connection
- **MGCLSendServerReport()** – Updates host information

Depending on whether your application uses a “Client/Server” or a “Peer-to-Peer” network topology, refer to the appropriate diagrams (Figure 2 and/or Figure 3) to help understand the discussion of “Maintaining Medius Platform Connections and Objects”:

Figure 2: Maintaining Medius Platform Connections and Objects for “Client/Server” Games

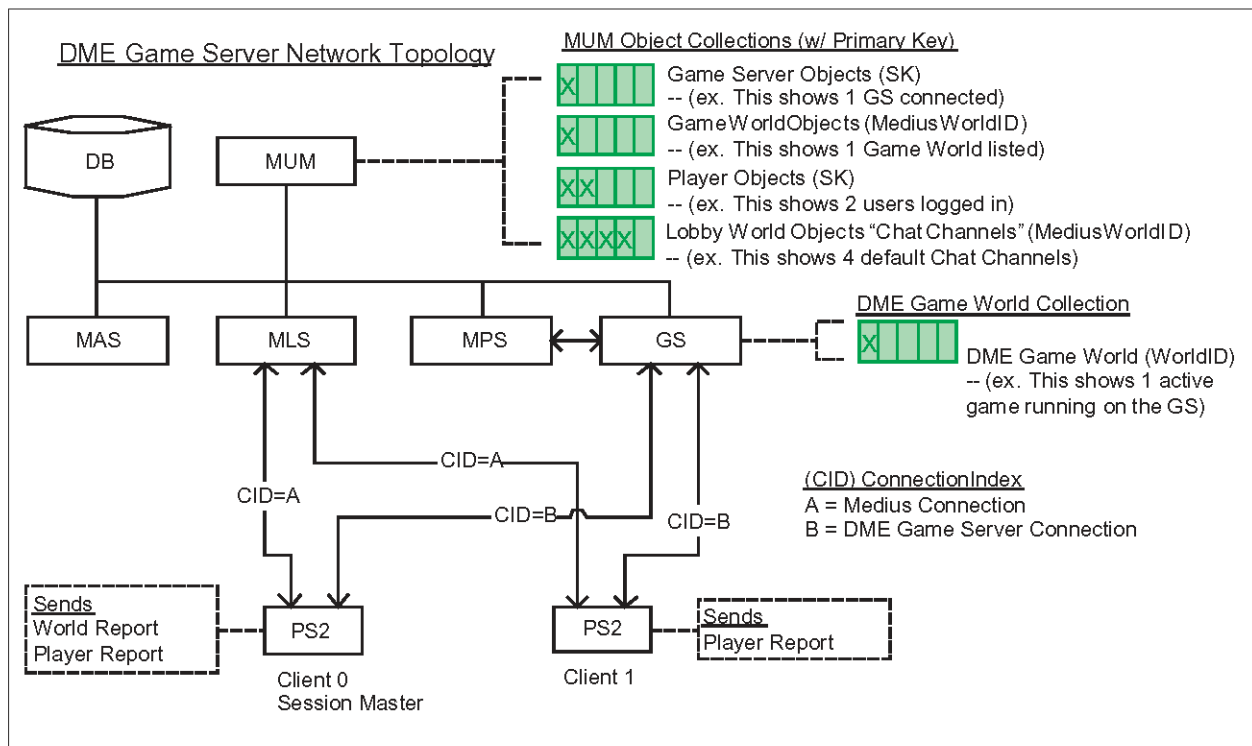
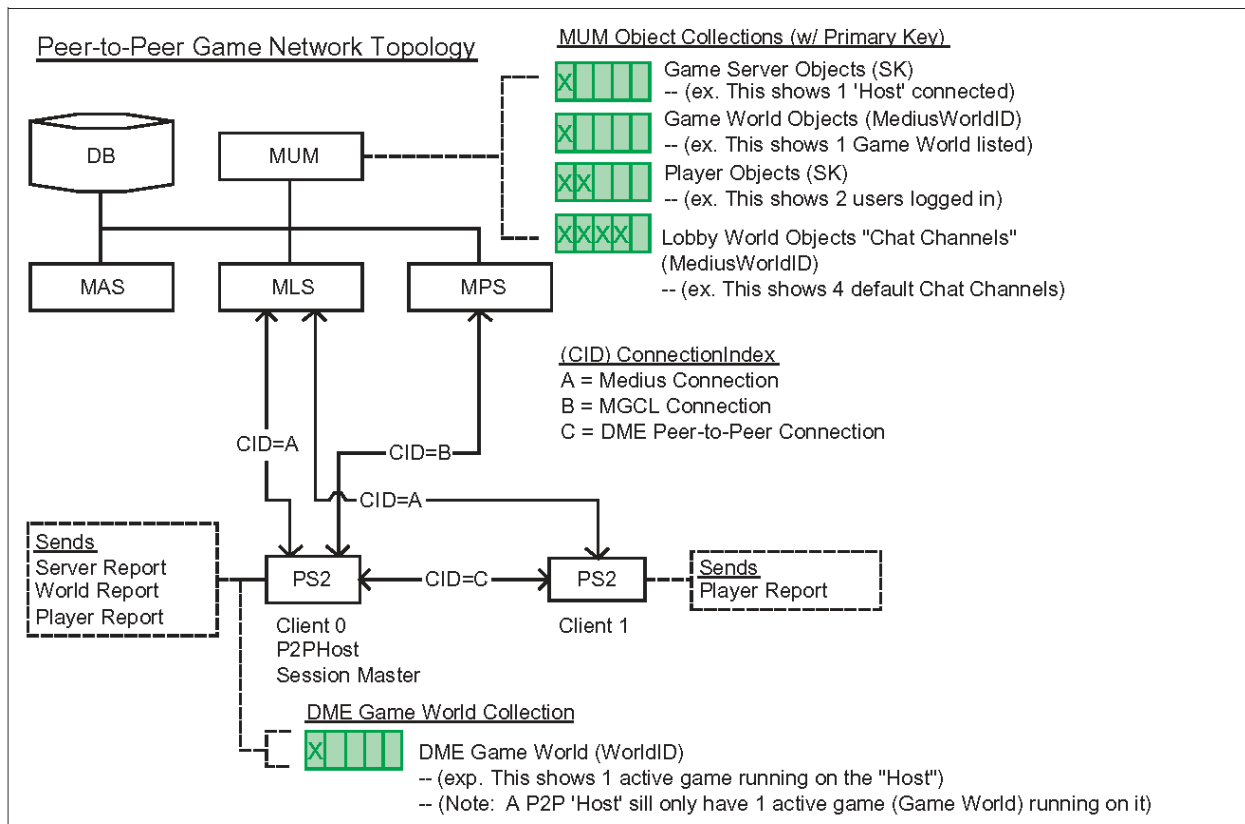


Figure 3: Maintaining Medius Platform Connections and Objects for “Peer-to-Peer” Games



Understanding MUM Game World Objects and DME Game Worlds

It is important to understand the difference between MUM 'game world objects' and DME 'game worlds'.

If a Client/Server Game

1. A DME 'game world' is a single instance of a game that resides on a DME Game Server (GS). Thus, a client can only be connected to a single DME 'game world' on a GS at any one time.
2. A GS hosts a collection of DME 'game worlds' that are indexed by "WorldID".
3. A call to **MediusCreateGame()** causes the MUM to manage the transaction of creating/starting a 'game world' on a GS.
4. If the transaction was successful, a corresponding 'game world object' is created on the MUM and a "MediusWorldID" is generated to index this object.
5. If no players join this game within 120 seconds, the 'game world object' will automatically timeout. The user who called **MediusCreateGame()** must still call **MediusJoinGame()** to join the DME 'game world' they just created.
6. A MUM 'game world object' encapsulates three things: Information about an active DME 'game world' running on a GS; connection information to the 'game world'; and a list of players connected to the 'game world'. The accuracy of the data encapsulated within a 'game world object' is dependent on the various report messages coming from all the clients connected to the corresponding active 'game world'.

7. A player in the application's lobby can get a list of games (via **MediusGetGames()**, which enumerates all the current 'game world objects') then request to join a given 'game world' (via **MediusJoinGame()**, using the respective *MediusWorldID*). If the join request is successful, the DME 'game world's connection information (*NetConnectionInfo*) will be returned so that a **NetConnect()** to this 'game world' can take place.

If a Peer-to-Peer Game

1. A DME 'game world' is a single instance of a game that resides on a peer-to-peer 'host'. This 'game world' is created with a call to **NetHostPeerToPeer()** (which is used for both Online and LAN peer-to-peer games).
2. A peer-to-peer 'host' theoretically can host a collection of DME 'game worlds' that are indexed by "WorldID"; but in practice, a peer-to-peer 'host' will only ever have one active 'game world' running (with *WorldID* set to 1).
3. A call to **MGCLCreateGameOnMeRequest()** causes the MUM to manage the transaction of registering/posting a peer-to-peer host's 'game world' with the Medius platform.
4. If the transaction is successful, a corresponding 'game world object' will be created on the MUM and a "*MediusWorldID*" will be generated to index this object.
5. Unlike a Client/Server game, the player that called **NetHostPeerToPeer()** will not need to call **MediusJoinGame()** to connect to the DME 'game world' created on the host (since the hosting player is assumed to be already connected to itself).
6. A MUM 'game world object' encapsulates three things: Information about an active DME 'game world' running on a peer-to-peer 'host'; connection information to this 'game world'; and a list of players connected to this 'game world'. The accuracy of the data encapsulated within a 'game world object' is dependent on the various report messages coming from all the clients connected to the corresponding active 'game world'.
7. A player in the application's lobby can get a list of games (via **MediusGetGames()**, which enumerates all the current 'game world objects'), then request to join a given 'game world' (via **MediusJoinGame()**, using the respective *MediusWorldID*). If the join request is successful, the DME 'game world's connection information (**NetConnectionInfo**) will be returned so that a **NetConnect()** to this 'game world' can take place.

Report Intervals

It is up to you to schedule the transmission of all reports, as well as to populate the message structure that will be sent with each report. It is recommended that all reports be sent every 25–30 seconds. Failure to send a specified report will cause the respective server-side object to timeout (the exact duration depends on the MUM Object Collection type; most are set to 60–120 seconds).

As discussed in the previous section "Ungraceful Disconnects", you need to consider the same situations that may prevent the sending of Medius reports and calling **MediusUpdate()** (such as hitting compiler breakpoints, blocking application function calls > x amount of time, and so forth).

The accuracy of the application's lobby data is based entirely on the frequency of these reports. Shorter report intervals (and manual report sends) increase the accuracy of the data presented to players in the lobby. Common Medius functions that depend on fresh report information are:

- **MediusGetGames()** – Sends request for list of game worlds (that is, it enumerates all – or a range of – 'game world objects').
- **MediusGetGameInfo()** – Sends request for information about a specific game world (that is, it retrieves information about a specific 'game world object').
- **MediusGetGamePlayers()** – Sends request for a list of players connected to a specific game world (that is, it retrieves the 'game world object's list of players connected to the given 'game world' – accuracy is dependent on player reports).

- **MediusGetPlayerInfo()** – Sends request for information about a specific player (that is, retrieves ‘player object’ information).
- **MediusGetTotalGames()** – Retrieves the total number of games for a given *ApplicationID*.

Special Notes about Peer-to-Peer Host Migration and MediusWorldID

The *SCE-RT SDK Medius Game Communication Library (MGCL) Release 2.10 – Overview* has more information about how to handle Peer-to-Peer Host Migration but a special mention must be made about the *MediusWorldID* used for Medius platform reports due to the host migrating.

How and When a MediusWorldID Can Change

In a client/server game, the *MediusWorldID* (needed for reports) will never change. However, if the host migrates in a peer-to-peer game (as detected by the callback **NetTypePeerToPeerHostChangeCallback** set within *NetConnectInParams*), **MGCLMoveGameWorldOnMe()** must be called when a given client is identified as the new host.

If successful, a new MUM ‘game world object’ is created on the Medius platform to reference the new DME ‘game world’ on the new host’s machine. A new *MediusWorldID* is then specified in the **MediusServerMoveGameWorldOnMeResponse** message.

MediusTypeReassignGameMediusWorldIDCallback will be triggered (as set in *MediusInitializeInParams*) for all other clients. The new *MediusWorldID* is specified in the **MediusReassignGameMediusWorldID** message. Each player should immediately send a (forced/manual) report (based on reporting responsibilities) upon receipt of a new *MediusWorldID*.

Maintaining the Medius Connection

Once a player is fully logged in with the Medius platform, calling **MediusUpdate()** is necessary to ensure that the user’s session (player object) and connection (socket) are maintained (with a 30–45 second timeout threshold).

Additional Notes About MediusUpdate()

1. **MediusUpdate()** manages the processing of sending and receiving Medius messages.
2. The previous section, ‘Ungraceful Disconnects’, describes what happens if you fail to call **MediusUpdate()** within a fixed amount of time. It is highly recommended that you wrap/encapsulate the **MediusUpdate()** function and let your application keep track of the elapsed time between calls (with either **NetGetLocalTime()** or your own timing function). If your application detects an elapsed time > 14 seconds then **ASSERT()**.
3. If **MediusUpdate()** does not return *MediusErrorNone*, then your application should attempt to handle the current error condition (if possible). The only other *MediusErrorCode* results that this function may generate are:
 - a. **MediusErrorNotInitialized** – **MediusInitialize()** or **MediusInitializeBare()** have not been called yet.
 - b. **MediusErrorGeneral** – An internal call to **NetUpdate()** to service/update the Medius connection returned an error code (call **MediusGetLastNetUpdateError()** for more information).

Sending World Reports (Maintaining Game World Objects)

MediusSendWorldReport() is the Medius API function used to send 'world reports'. The PlayStation®2 client designated as the game's session master is responsible for periodically sending a 'world report' over its Medius connection via **MediusSendWorldReport()**. This report message should be sent every 25–30 seconds, primarily to act as a MUM 'game world object' heartbeat.

The MLS uses world report messages to update the state of the respective 'game world object' residing on the MUM. This report can be used to change attributes (on-the-fly) of a 'game world object', such as the game name, current game level, game password, minimum skill level, and so forth.

Additional Notes About MediusSendWorldReport()

1. You only send 'world reports' after successful creation of (and connection to) a game (that is, a DME 'game world'). You can create a new game (and its associated 'game world object', represented with a *MediusWorldID* on the Medius platform) in one of two ways (see "Understanding MUM Game World Objects and DME Game Worlds" for more information):
 - a. If you are creating a client/server game, call **MediusCreateGame()** to create a DME 'game world' on a standalone DME game server (which will then trigger the creation of a 'game world object' on the Medius platform). The player that created the game will then call **MediusJoinGame()** using the *MediusWorldID* in the **MediusCreateGameResponse** message. Other players call **MediusGetGames()** to get a list of 'game world objects' and call **MediusJoinGame()** using the *MediusWorldID* in a given **MediusGameListResponse** message.

Note: It is recommended that the player that created the game call **NetJoin()** as soon as possible so that a session master can be designated and all players will know who is responsible for sending 'world reports'. After **MediusCreateGame()**, you have about 120 seconds to send a 'world report' before the 'game world object' will timeout; nonetheless, you should target (for example) < 10 seconds, being mindful of issues such as a long DVD load time.
 - b. If you are the host of a peer-to-peer game, call **MGCLCreateGameOnMeRequest()** to trigger the creation of a 'game world object' on the Medius platform. Calling **NetHostPeerToPeer()** automatically creates a DME 'game world' on the host, so the host will not need to call **MediusJoinGame()**. As above, other players call **MediusGetGames()** to get a list of 'game world objects' and call **MediusJoinGame()** using the *MediusWorldID* in the **MediusGameListResponse** message.

Note: The host should be responsible for sending 'world reports' until a session master is designated with a call to **NetJoin()**. If a **NetTypeSMChangeCallback** (part of **NetJoinInParams**) gets triggered (because the *SessionMaster* migrated to another client), the new session master should then be responsible for 'world reports'. After a **MGCLCreateGameOnMeRequest()** you have about 120 seconds to send a 'world report' before the 'game world object' will timeout; nonetheless, you should target (for example) < 10 seconds – being mindful of issues such as a long DVD load time.
2. If you stop sending 'world reports', the Medius platform will end/close the 'game world object', thus preventing any new players from finding and joining the game. Nonetheless, the game will continue to be playable on the standalone DME game server or peer-to-peer host's machine until everyone has disconnected and/or returned to the Medius lobby.

3. If a 'game world object' times outs (due to a 'world report' stoppage), the log files will record this as (for example):

```
(MPS Log)
12:15:10  END_WORLD_ON_GAME_SE  Destroy Game Message received from MUM -fwd to
Proxy Server
```

```
(MUM Log)
12:15:10  MUM_TIMER  GameWorld IP 208.236.15.245 PORT 41031 World 1 time out
event
12:15:10  MUM_TIMER  Success sending fwd End Game msg to ConnWorldID 1
ConnClientIndex 2
```

4. If for some reason the sending of 'world reports' resumes after the 'game world object' timed out, they essentially will be ignored by the Medius platform; nonetheless, the log files will still record this as (for example):

```
(MUM Log)
12:20:00  XML_BASE  UpdateGameWorldByIDStart
12:20:00  WORLD  MediusWorldID 1 Game Name PB-P2P-5408
12:20:00  WORLD  WMUpdateGameWorldByID error -404
```

5. One way to verify a 'world report' was successfully processed by the Medius platform is to call **MediusGetGames()**. This will enumerate and return all games listed on the Medius platform with the given *ApplicationID*. If you see your game listed, you're assured that the 'world report' is functioning properly and maintaining the 'game world object' on the Medius platform.

The Sequence for Sending a 'World Report'

1. Create an instance of the *MediusWorldReport* structure.
 - a. `memset()` the structure to '0'.
 - b. Set *SessionKey* to the value you saved from **MediusSessionBeginResponse**.
 - c. Set *MediusWorldID* to the value you saved from:
 - (If client/server game creator) **MediusCreateGameResponse**
 - (If peer-to-peer host) **MediusServerCreateGameOnMeResponse**
 - (If joining) The value passed in for **MediusJoinGame()** (for both client/server and peer-to-peer)
- Note:** Be mindful of whom the current *SessionMaster* is and/or if a peer-to-peer "host" migration took place (causing a new *MediusWorldID* to be assigned).
- d. Set *PlayerCount* to the current DME player count. Use **NetGetValidClientCount()** with either your GS connection index or your peer-to-peer connection index.
 - Note:** Currently, the report's *PlayerCount* field no longer updates the 'game world object's *PlayerCount*. This count is now managed by the various incoming 'player reports'. However, for verification purposes, it is recommended that you still set this field.
- e. Set *GameName* to either the same name used during game creation or a new name. (It is possible to pack additional string information in the *GameName* if your application has any special UI display needs).
- f. Set *GameStats* to an (optional) string provided by the application. Each application will decide if and how they wish to use this field.
- g. Set *MinPlayers* to the same value used during game creation. (For simplicity's sake, do not change).
- h. Set *MaxPlayers* to the same value used during game creation. (For simplicity's sake, do not change).
- i. Set *GameLevel* to an arbitrary value. (Possible usage may be to represent a given map or difficulty setting).

- j. Set *PlayerSkillLevel* to an arbitrary value.
 - k. Set *RulesSet* to an arbitrary value.
 - l. Set *GenericField1* to n (optional) string provided by the application. Each application will decide if and how they wish to use this field.
 - m. Set *GenericField2* to an (optional) string provided by the application.
 - n. Set *GenericField3* to an (optional) string provided by the application.
 - o. Set *WorldStatus* to any *MediusWorldStatus* type. This represents the state that the DME 'game world' is in: Inactive, Staging, Active, or Closed. Your application may handle each state differently.
2. Call **MediusSendWorldReport()** with the respective report message and verify that *MediusErrorNone* was returned (if this function fails you should only expect 'common errors', as declared in 'General Steps in Executing Medius API Function Calls').

Sample – Sending a World Report

```
void MyGame_IssueWorldReport(MediusWorldStatus eWorldStatus)
{
    MediusWorldReport report;
    char szWorldStatus[50];
    MediusErrorCode result;

    memset(&report, 0, sizeof(MediusWorldReport));

    switch (eWorldStatus)
    {
    case WorldInactive:
        strcpy(szWorldStatus, "WorldInactive");
        break;
    case WorldStaging:
        strcpy(szWorldStatus, "WorldStaging");
        break;
    case WorldActive:
        strcpy(szWorldStatus, "WorldActive");
        break;
    case WorldClosed:
        strcpy(szWorldStatus, "WorldClosed");
        break;
    default:
        strcpy(szWorldStatus, "Error");
        break;
    } // End_SWITCH

    printf("(Time:'%0.2ld:%0.2ld:%0.2ld')",
        (long) ((g_iTimeOfLastUpdate / 1000 / 60 / 60)),
        (long) ((g_iTimeOfLastUpdate / 1000 / 60) % 60),
        (long) ((g_iTimeOfLastUpdate / 1000) % 60));

    printf("Sending World Report: (%s) (MWID:%d) (SK:%s) (%d-%s)\n",
        g_RtLobby.m_GameName,
        g_RtLobby.m_iMediusWorldID,
        g_RtLobby.m_SessionKey,
        eWorldStatus,
        szWorldStatus);
    // -----
    // Set World Report
    // -----
}
```

```

    strncpy(report.SessionKey, g_RtLobby.m_SessionKey, SESSIONKEY_MAXLEN-1);
    report.SessionKey[SESSIONKEY_MAXLEN-1]='\0';
    report.MediusWorldID      = g_RtLobby.m_iMediusWorldID;
    report.PlayerCount        = 1; // (Ignored) Handled by Player Reports
    strncpy(report.GameName, g_RtLobby.m_GameName, GAMENAME_MAXLEN-1);
    report.GameName[GAMENAME_MAXLEN-1]='\0';
    strncpy(report.GameStats, "", GAMESTATS_MAXLEN-1);
    report.GameStats[GAMESTATS_MAXLEN-1]='\0';
    report.MinPlayers         = 1; // Use same value from create game
    report.MaxPlayers         = 8; // Use same value from create game
    report.GameLevel          = 1;
    report.PlayerSkillLevel   = 1;
    report.RulesSet           = 1;
    report.GenericField1      = 0;
    report.GenericField2      = 0;
    report.GenericField3      = 0;
    report.WorldStatus        = eWorldStatus;

    // -----
    // Medius Send World Report
    // -----
    result = MediusSendWorldReport(&report);
    if (result)
    {
        printf("App:[%s][%d]MediusErrorCode:'%d'", FILE, LINE, result);
    }

    return;
}

```

Sending an End Game Report (Closing Game World Objects)

MediusSendEndGameReport() is the Medius API function to send an ‘end game report’. There are only two ways for a ‘game world object’ to end/close:

1. timeout, or
2. ‘end game report’ was received.

When a game has determined that it is time to end (that is, the DME ‘game world’ should be closed/shutdown), the client acting as the game’s session master is responsible for sending an ‘end game report’ over its Medius Connection. The MLS will then save the statistics of each player (player object) associated with the given ‘game world object’ to the Medius database and remove the ‘game world object’ from the MUM.

Coordinating the Sending of End Game Reports

You must consider a player’s (player object’s) *Stats* field at the time an ‘end game report’ is received – the last received *Stats* field will be persisted (that is, the last ‘player report’ x-amount of time ago); therefore, if *Stats* are an important aspect of your application you must have the session master coordinate the process of having every client send one last ‘player report’ or **MediusAccountUpdateStats()** call. This process may look something like this:

1. An application level ‘end-game-event’ is triggered to indicate that the current game has ended (because someone crossed the finish line in a race, all hostages have been rescued, the enemy team was eliminated, or some other game-specific condition was reached). This may or may not have occurred on the session master’s machine. A client is then responsible for sending a message (either with a DME message or a field change in a DME NetObject that represented the game’s current state) to the session master and the other clients.

2. Upon receiving this 'end-game-event', every client takes a snapshot of the data necessary to store in their *Stats* field.
3. After the session master has taken a snapshot of their own *Stats* field, the session master then must coordinate directly with each and every client the responsibility of sending one last '*Stats*' update. This would most likely look like an application-defined DME message that says, "Client-x send your final *Stats* update". A client receiving this message would say, "Okay SM, I just finished sending my final *Stats* update" and send a DME message back to the session master indicating as such.
4. Once the session master has received the final *Stats* update from each and every client, s/he is ready to send an 'end game report'.
5. If a client never responds to the session master's request to send one last *Stats* update (perhaps they disconnected), the session master should timeout (ignore) that particular request to that client (with, say, a 10-second timeout threshold) and proceed with the 'end game report'.

Note: If you record 'Player Disconnects' (in regards to Wins, Losses, and Disconnects), it is recommend that after x-amount of time into a game, you manually update (increment) the 'Disconnect' section of a player's *Stats* field; therefore, when step #2 takes place, you must increment the Wins or Losses section and set *Disconnects* to its value prior to the game starting. See the "Ladder Management" section for more information. Be careful of the usage of **MediusAccountUpdateStats()** and recording "Wins, Losses, and Disconnects", because this will immediately persist the player's *Stats* field to the database.

Additional Notes About MediusSendEndGameReport()

1. When the Medius platform receives this report message, all the player's *Stats* fields will be persisted to the Medius database (similar to the functionality of a manual **MediusAccountUpdateStats()** call).
2. In a peer-to-peer game: When a 'game world object' closes on the Medius platform, a **MGCLServerEndGameCallback** will be triggered on the 'host' (as declared in **MGCLInitializeInParams**). This callback will not automatically close the DME 'game world' (basically, you should ignore **MGCLServerEndGameCallback** because it only indicates that the 'game world object' has closed; ignore **MGCLEndGameResponse()** at this point also). You do, however, need to provide additional shutdown logic that involves (for example) the 'host' client sending to all players an application-specific **NetSendMessage()** that indicates, "Disconnect now and return to the Lobby". This way you can coordinate all players leaving at the same time. Otherwise, the 'end game report' will only set every user's 'player object' on the Medius platform to "**JoinedChatWorld**" (just as if you called a **MediusUpdateClientState()**), even though all the clients may still be connected to the DME 'game world' on the peer-to-peer 'host'.
3. In a client/server game: Similar to above, an 'end game report' will not automatically close the DME 'game world' on the GS. The session master should send an application-specific **NetSendMessage()** to all players that indicates, "Disconnect now and return to the Lobby". This way you can coordinate all players leaving at the same time. Otherwise, the 'end game report' will only set every user's 'player object' on the Medius platform to "**JoinedChatWorld**" (just as if you called **MediusUpdateClientState()**), even though all the clients may still be connected to the DME 'game world' on the GS.
4. After the given 'game world object' has been closed, subsequent 'player reports' will eventually log (-404) errors on the MUM (which are essentially ignored by the Medius platform).
5. The normal process when a player exits a game is as follows:
 - a. Call **MediusUpdateClientState()**, with *UserAction* set to **LeftGameWorld**.
 - b. Call **MediusUpdateClientState()**, with *UserAction* set to **JoinedChatWorld**.
 - c. Call **NetLeave()** on the DME 'game world's connection.
 - d. Call **NetDisconnect()** on the DME 'game world's connection. Be sure to specify the **NetDisconnectReason**.

6. Setting *WinningTeam*, *WinningPlayer*, and *FinalScore* (of the **MediusEndGameReport** type) requires special components to be built that describe how to manage “tournament logic”. Custom fields may be added to this report. Contact SCE-RT developer support (scert-support@scea.com) for more information.

The Sequence for Sending an ‘End Game Report’

1. Create an instance of the *MediusEndGameReport* structure.
 - a. **memset()** the structure to ‘0’.
 - b. Set *SessionKey* to the value you saved from **MediusSessionBeginResponse**.
 - c. Set *MediusWorldID* to the value you saved from:
 - (If client/server game creator) **MediusCreateGameResponse**.
 - (If peer-to-peer host) **MediusServerCreateGameOnMeResponse**.
 - (If joining) The value passed in for **MediusJoinGame()** (for both C/S and P2P).

Note: Be aware of who the current *SessionMaster* is and/or (if peer-to-peer) if a peer-to-peer ‘host’ migration occurred (and thus a new *MediusWorldID* has been assigned).
 - d. *WinningTeam* (Not implemented by default).
 - e. *WinningPlayer* (Not implemented by default).
 - f. *FinalScore* (Not implemented by default).
2. Call **MediusSendWorldReport()** with the proper report message and verify that *MediusErrorNone* was returned (if this function fails, you should only expect ‘common errors’, as declared in ‘General Steps in Executing Medius API Function Calls’).

Sample – Sending an End Game Report

```
void MyGame_IssueEndGameReport()
{
    MediusEndGameReport report;
    MediusErrorCode result;

    memset(&report, 0, sizeof(MediusEndGameReport));

    printf("(Time: '%0.2ld:%0.2ld:%0.2ld') ",
           (long) ((g_iTimeOfLastUpdate / 1000 / 60 / 60)),
           (long) ((g_iTimeOfLastUpdate / 1000 / 60) % 60),
           (long) ((g_iTimeOfLastUpdate / 1000) % 60));
```

```

printf("Sending End Game Report:(MWID:%d) (SK:%s)\n",
      g_RtLobby.m_iMediusWorldID,
      g_RtLobby.m_SessionKey);

// -----
// Set End Game Report
// -----
strncpy(report.SessionKey, g_RtLobby.m_SessionKey, SESSIONKEY_MAXLEN-1);
report.SessionKey[SESSIONKEY_MAXLEN-1]='\0';
report.MediusWorldID = g_RtLobby.m_iMediusWorldID;
// If your application utilizes the following fields, contact SCE-RT developer
// support for more information.
report.WinningTeam;           // Currently not implemented
report.WinningPlayer;         // Currently not implemented
report.FinalScore;            // Currently not implemented

// -----
// Medius Send End Game Report
// -----
result = MediusSendEndGameReport(&report);
if (result)
{
    printf("App:[%s][%d]MediusErrorCode:'%d'", FILE, LINE, result);
}
return;
}

```

Sending Player Reports (Maintaining Player Objects)

MediusSendPlayerReport() is the Medius API function used to send 'player reports'. Every client connected to a DME 'game world' is responsible for periodically sending a 'player report' to the MLS. The MLS uses these report messages to update the state of the corresponding 'player object' residing on the MUM.

Note: Not sending 'player reports' will not cause 'player objects' to timeout/close (unlike other server-side objects). Nonetheless, you should still send them periodically to keep the 'player object's data current on the Medius platform (player reports should be sent every 25–30 seconds).

The MUM manages a collection of 'game world objects' and each 'game world object' maintains a list of references (pointers) to 'player objects' that represent clients joined to a particular DME 'game world'. The first time a 'player report' is received (with a given *MediusWorldID*), a reference to that player's 'player object' will be added to the corresponding 'game world object's list of players joined. This 'player report' will additionally increment the *PlayerCount* of the given 'game world object'.

Additional Notes About MediusSendPlayerReport()

1. 'Player reports' are sent after successfully joining a game (or creating a game). Upon joining, you should send a 'player report' immediately to keep the Medius platform informed of the current state.

Note: If you do not send the first 'player report', the given game can still be played, but the Medius platform will not have an accurate *PlayerCount* for that particular game and will have no idea that you ever 'successfully' joined that game. I.e., *PlayerCount* due to a **MediusJoin()** is not managed because the **NetConnect()** may have failed. This is how it is possible to overcome the particular 'race condition' if, for instance, 5 clients all simultaneously attempted to join a 15- or 16-player game.

One way to verify if a player report was successfully processed by the Medius platform is to call **MediusGetGamePlayers()**. This will enumerate all the players associated/connected with the given game's *MediusWorldID*.

2. A player's *Stats* field is not persisted to the Medius database until either a **MediusSendEndGameReport()** is received or the application calls **MediusAccountUpdateStats()**.

Special Note About a Player's Stats Field

A player's *Stats* field (elements 2 through 12) is no longer used in by the Medius platform's "Ladder System". In the past, the Ladder system was limited to only one rankable Ladder. As of version 1.50, the Medius platform has been expanded to support up to 100 rankable Ladders (see the "Ladder Management" section for more information).

Sequence for Sending a 'Player Report'

1. Create an instance of the *MediusPlayerReport* structure:
 - a. **memset()** the structure to '0'.
 - b. Set *SessionKey* to the value you saved from **MediusSessionBeginResponse**.
 - c. Set *MediusWorldID* to the value you saved from:
 - (If client/server game creator) **MediusCreateGameResponse**.
 - (If peer-to-peer host) **MediusServerCreateGameOnMeResponse**.
 - (If joining) The value passed in for **MediusJoinGame()** (for both client/server and peer-to-peer).

Note: (If peer-to-peer) Be aware of whether a peer-to-peer 'host' migration occurred (and thus a new *MediusWorldID* has been assigned).
 - d. Set *Stats* (if utilized) as appropriate (see "Special Note About a Player's Stats Field" above).
2. Call **MediusSendPlayerReport()** with the proper report message and verify that *MediusErrorNone* was returned (if this function fails, you should only expect 'common errors', as declared in 'General Steps in Executing Medius API Function Calls').

Sample – Sending a Player Report

```
void MyGame_IssuePlayerReport()
{
    MediusPlayerReport report;
    MediusErrorCode result;

    memset(&report, 0, sizeof(MediusPlayerReport));

    printf("(Time: '%0.2ld:%0.2ld:%0.2ld')",
           (long) ((g_iTimeOfLastUpdate / 1000 / 60 / 60)),
           (long) ((g_iTimeOfLastUpdate / 1000 / 60) % 60),
           (long) ((g_iTimeOfLastUpdate / 1000) % 60));

    printf("Sending Player Report: (MWID: %d) (SK: %s)\n",
           g_RtLobby.m_iMediusWorldID,
           g_RtLobby.m_SessionKey);

    // -----
    // Set Player Report
    // -----
    strncpy(report.SessionKey, g_RtLobby.m_SessionKey, SESSIONKEY_MAXLEN-1);
    report.SessionKey[SESSIONKEY_MAXLEN-1]='\0';
    report.MediusWorldID = g_RtLobby.m_iMediusWorldID;
    // If your application utilizes the Stats field
    report.Stats;           // Set appropriately
}
```

```

// -----
// Medius Send Player Report
// -----
result = MediusSendPlayerReport(&report);
if (result)
{
    printf("App:[%s][%d]MediusErrorCode:'%d'", FILE, LINE, result);
}
return;
}

```

Maintaining the MGCL Connection

Once a 'host' of a peer-to-peer game has established their MGCL Connection with the Medius platform, you must call **MGCLUpdate()** to ensure that the host's session (game server object) and connection (socket) are maintained (with a 30–45 second timeout threshold).

Note: Having an MGCL Connection established with the Medius platform does not explicitly indicate that a DME 'game world' has been created on the 'host' machine. It simply means that a 'game server object' has been created on the Medius platform and is ready for a **MGCLCreateGameOnMeRequest()** call that will create and associate a 'game world object' with a DME 'game world'.

Additional Notes about MGCLUpdate()

1. **MGCLUpdate()** manages the process of sending and receiving 'server-based' messages. (See the "MGCL Overview" for more information about MGCL).
2. **MGCLUpdate()** exhibits the same properties as **MediusUpdate()** if you fail to call it within a fixed amount of time; you should **ASSERT()** if > 14 seconds elapsed between update calls.

Sending Server Reports (Maintaining Game Server Objects)

MGCLSendServerReport() is the MGCL API function used to send 'server reports'. This report message is sent by each game server (peer-to-peer host) to the Medius Proxy Server (MPS) to notify the Medius platform of the 'host's current state. Once a host's MGCL connection has been established, 'server reports' will need to occur regardless of whether a DME 'game world' has been created yet on the host's machine (server reports should be sent every 25–30 seconds). As a backend service (shielding the application developer), 'server reports' keep the 'game server object' current on the Medius platform while **MGCLUpdate()** maintains/services the 'game server object'.

Additional Notes About MGCLSendServerReport()

1. After the MGCL connection has been established, you must send 'server reports' regardless of whether there is currently a DME 'game world' created yet or not. If a game has not yet been created, you set *ActiveWorldCount* and *TotalActivePlayers* to '0'.
2. A game is created using **MGCLCreateGameOnMeRequest()**, which creates a 'game world object' that encapsulates the *NetConnectInfo* to a **NetHostPeerToPeer()** game (that is, a DME 'game world') with the Medius platform. After a successful call to **MGCLCreateGameOnMeRequest()** a 'game world object' is created on the Medius platform that the 'game server object' will reference. You should then set *ActiveWorldCount* to '1' and enumerate all the players currently connected to the game (using **NetGetValidClientCount()**) and set this for *TotalActivePlayers*.
3. Just as is the case with 'world reports', if you stop sending 'server reports' the Medius platform will end/close both the 'game world object' and the 'game server object', thus preventing any new players from finding and joining the game. The game will continue to be playable on the peer-to-peer host's machine until all current players have disconnected and/or returned to the Medius lobby.
4. If a 'game server object' times out (due to a 'server report' stoppage), the log files will record this as (for example):

```
(MUM Log) This is the only log file that will report this event
10:25:48 MUM_TIMER GameServer time out event Skey 3 ID=0 Status=3
IntermedServerID=0 ClientIndex=0 WID=1
```

5. If for some reason the sending of 'server reports' resumes after the 'game server object' timed out (causing the 'game world object' to close), they essentially will be ignored by the Medius platform; nonetheless, the log files will record this as (for example):

```
(MUM Log)
// Server Report will cause this error message
10:30:37 WORLD WMGetGameServerInfoBySessionKey error -405
// World Report will cause this error message
10:30:39 XML_BASE UpdateGameWorldByIDStart
10:30:39 WORLD MediusWorldID 1 GameName PB-P2P-1710
10:30:39 WORLD WMUpdateGameWorldByID error -107
```

Sequence for Sending a 'Server Report'

1. Create an instance of the **MediusServerReportType** structure:
 - a. **memset()** the structure to '0'.
 - b. Set *SessionKey* to the value you saved from **MediusServerSessionBeginResponse**.
 - c. Set *MaxWorlds* to '1' (this should not change).
 - d. Set *MaxPlayersPerWorld* to the maximum number of clients that may be connected to the DME 'game world' (this will be same value set for *MaxClients* as part of the **NetHostPeerToPeerInParams** structure and should not change).
 - e. Set *ActiveWorldCount* to '0' if no DME 'game world' has been created on the host yet; otherwise, set this to '1' if the host has called **NetHostPeerToPeer()**. It is assumed that only one 'game world' will ever be created on a peer-to-peer host's machine.
 - f. Set *TotalActivePlayers* to the current DME player count (obtained using **NetGetValidClientCount()** with the peer-to-peer connection index).
 - g. Set *AlertLevel* to MGCL_ALERT_NONE (this should not change). This field is primarily used by a GS (DME Game Servers also use the MGCL API to communicate with the Medius platform).
2. Call **MGCLSendServerReport()** with the proper report message and verify that MGCL_SUCCESS was returned.

Sample – Sending a Server Report

```
void MyGame_IssueServerReport()
{
    MediusServerReportType report;
    MGCL_ERROR_CODE result;
    int iConnectedToHost = -1;

    memset(&report, 0, sizeof(MediusServerReportType));

    // -----
    // Set Server Report
    // -----
    strncpy(report.SessionKey, g_RtMGCL.m_MGCLSessionKey, SESSIONKEY_MAXLEN-1);
    report.SessionKey[SESSIONKEY_MAXLEN-1]='\0';
    report.MaxWorlds = 1; // P2P host will only ever have 1 'game world'
    report.MaxPlayerPerWorld = MAX_CLIENTS;
    if (g_GameCreated == TRUE) {
        report.ActiveWorldCount = 1;
    } else{
        report.ActiveWorldCount = 0;
    }
    NetGetValidClientCount(&iConnectedToHost, g_P2P_pHDME);
```



```

if ((iConnectedToHost < 0) || (iConnectedToHost > MAX_CLIENTS))
{
    ASSERT(0);
}
report.TotalActivePlayers = iConnectedToHost;

printf("(Time:%0.2ld:%0.2ld:%0.2ld)",
        (long) ((g_iTimeOfLastUpdate / 1000 / 60 / 60)),
        (long) ((g_iTimeOfLastUpdate / 1000 / 60) % 60),
        (long) ((g_iTimeOfLastUpdate / 1000) % 60));

printf("(MGCL)Sending Server Report:(SSK:%s) (AWC:%d) (TAP:%d)\n",
        report.SessionKey,           // Server Session Key
        report.ActiveWorldCount,
        report.TotalActivePlayers);

// -----
// MGCL Send Server Report
// -----
result = MediusSendPlayerReport(&report);
if (result)
{
    printf("App:[%s][%d]MGCL_ERROR_CODE:'%d'", FILE, LINE, result);
}
return;
}

```

Leaving a Game World

When a player leaves a 'game world' (either gracefully or ungracefully), the player should inform the Medius platform of their intentions so that the lobby will be up-to-date; otherwise, the Medius platform will be relying on timeout events to update and manage its Object Collections.

If a player wishes to leave a 'game world' and return to a lobby (either by their own will or because of an upcoming 'end game report'), they should:

1. Know how to persist their *Stats* field (if required).
2. Call **NetLeave()** to leave a DME 'game world'.
3. Call **NetDisconnect()** to disconnect from either a GS or peer-to-peer host (thus closing the socket).
Note: peer-to-peer hosts would also call this on themselves.
4. Call **MediusUpdateClientState()** with **MediusUpdateUserState's** *UserAction* set to 'LeftGameWorld'. This will remove the 'player object' reference that the 'game world object' maintained; the *PlayerCount* will also be decremented.
5. Call **MediusUpdateClientState()** with **MediusUpdateUserState's** *UserAction* set to 'JoinedChatWorld'. This will set the 'player object' on the Medius platform to a state where it is able to carry out Medius 'lobby' functionality.

If a player wishes to leave a 'game world' and shut down online connectivity, they should:

1. Know how to persist their *Stats* field (if required).
2. Call **NetLeave()** to leave a DME 'game world'.
3. Call **NetDisconnect()** to disconnect from either a GS or peer-to-peer host (thus closing the socket).
Note: peer-to-peer hosts would also call this on themselves.
4. Call **MediusUpdateClientState()** with **MediusUpdateUserState's** *UserAction* set to 'LeftGameWorld'. This will remove the 'player object' reference that the 'game world object' maintained; the *PlayerCount* will also be decremented.
5. Call **MediusDisconnect()** to close the Medius connection.

6. Call **MediusClose()** (if **MediusInitialize()** was called earlier) or **MediusCloseBare()** (if **MediusInitializeBare()** was called earlier) to clean up/free all connection-related resources in regards to the Medius connection index.
7. If you called **MediusCloseBare()** in step 6, call **NetClose()**.
8. The application is now ready to unload IRX modules.
Note: Not all of the PlayStation®2 network IRX modules may be unloaded; therefore, resetting the IOP may be required. See the PlayStation®2 network documentation for the issues involved with performing this task.
9. At this point, the player will (most likely) be back on the application's main selection screen, where they may choose to play either a single player or multiplayer game.

If a player disconnects by turning off their PlayStation®2 or by some other disastrous event, the Medius platform will eventually timeout all relevant server-side objects as appropriate and/or automatically migrate hosting responsibilities and NetObject ownership responsibilities. In such a case, note the following:

1. Be aware of whether you need to persist a player's *Stats* field.
2. A 'player object' will naturally timeout due to a lack of **MediusUpdate()** calls in about 30–40 seconds. If your application uses account logins, players must wait for their corresponding 'player object' to timeout before they will be able to login again..
3. A 'game world object' will naturally timeout due to lack of 'world reports' in about 120 seconds.
4. If the DME detects that a hosting player has timed out, a **NetTypePeerToPeerHostChangeCallback** will be triggered.

Chat Channel Management

After a user is fully logged into the Medius platform, they will be assigned to a default chat channel from which they can fully utilize “lobby” technology. This includes chatting, sending instant messages, and creating buddy lists. Users can also navigate to different chat channels or create new chat channels. Other basic “lobby” functionality includes getting a list of players on a given chat channel and, of course, finding, creating, and joining games (which will be discussed in the next section, “Game Management”).

This “lobby” technology as a whole (if fully integrated into your application) will help build a solid foundation for creating an “Online Community” for your title that can prove vital in the long-term success of your game.

The following Medius functions associated with Chat Channel Management and Chat Messages are described in this section:

- **MediusBanPlayer()** – Prevents a player from joining a given chat channel
- **MediusGenericChatSetFilter()** – Enables or disables a given chat type that can be received by the server
- **MediusCreateChannel()** – Creates a chat channel
- **MediusFindPlayer()** – Searches for player by *AccountID* or *AccountName*
- **MediusFindWorldByName()** – Searches for a chat channel by name
- **MediusGetChannelInfo()** – Gets detailed chat channel information
- **MediusGetChannels()** – Gets a list of all chat channels
- **MediusGetChannels_ExtraInfo()** – Gets a list of all chat channels, along with additional information
- **MediusGetLobbyPlayerNames()** – Gets a list of player names in a chat channel
- **MediusGetLobbyPlayerNames_ExtraInfo()** – Gets a list of player names in a chat channel, along with additional information
- **MediusGetLobbyPlayers()** – Gets a list of players in a chat channel
- **MediusGetPlayerInfo()** – Gets player information by *AccountID*
- **MediusGetTotalChannels()** – Gets a total count of chat channels
- **MediusGetTotalUsers()** – Gets a total count of players
- **MediusGetWorldSecurityLevel()** – Determines if the chat channel has a password
- **MediusJoinChannel()** – Requests to join a chat channel
- **MediusSendGenericChatMessage()** – Sends a chat message
- **MediusSetAutoChatHistory()** – Chat message history to receive
- **MediusSetWorldLobbyFilter()** – Sets a chat channel filters
- **MediusTextFilter()** – Implements server-side vulgarity checks

Preventing a Player from Joining a Chat Channel

The **MediusBanPlayer()** function (with *MediusApplicationType* set to ‘LobbyChatChannel’) allows you to provide logic in your application to prevent a specific player from joining a given chat channel. Setting a ‘ban’ on a particular player will prevent the player from receiving a successful **MediusJoinChannel()** response for a given chat channel. Currently, there is no “Kick” feature. This ‘ban’ can be put in place for a specified number of seconds or for the lifetime of the chat channel.

Disabling or Enabling Medius Chat

The **MediusGenericChatSetFilter()** function allows for the disabling or re-enabling of a given chat type that can be received by the server. **MediusChatMessageType** currently lists the various chat types that can individually be enabled or disabled including “broadcast messages, whisper messages, global broadcast messages, clan chat messages, and buddy chat messages”.

If the application has a **MediusTypeGenericChatFwdMessageCallback** declared, then all of the given chat types are enabled by default. If a player is in-game, for example, they may consider disabling some of the chat messages (like broadcast) but keep clan chat and buddy chat.

Creating a Chat Channel

Creating a chat channel is accomplished using the **MediusCreateChannel()** call. The **MediusCreateChannelRequest** data structure contains chat channel attributes such as chat channel name, minimum number of players, maximum number of players, chat channel password, and so forth. Upon receipt of a **MediusCreateChannelRequest** message, the MLS will manage the transaction of locating an available chat server (this could be any MLS that is connected to a given MUM) and reserving a ‘lobby world object (chat channel)’ identifiable by a newly generated *MediusWorldID* on both the MLS and MUM servers. The Medius platform response to the client will include this new *MediusWorldID*, which is then used to join the chat channel via **MediusJoinChannel()**.

Filterable attributes for chat channels can be set one time and one time only by setting *GenericField1* to *GenericField4* of **MediusCreateChannelRequest** during creation time. See “Filtering Chat Channels” below for more information.

Searching for a Player

The **MediusFindPlayer()** function searches for another player using either *AccountID* or *AccountName* (exact string match). The response message will return the *MediusWorldID* and *MediusApplicationType*, from which it can be determined if the other player is in a chat channel or in-game.

Searching for a Chat Channel

The **MediusFindWorldByName()** function (with **MediusFindInWorldType** set to “FindInLobbyWorld”) searches for a chat channel by Name (exact string match). The response message will return the *MediusWorldID* of the chat channel and other information.

Getting Chat Channel Information

Given a *MediusWorldID*, the **MediusGetChannelInfo()** function will return detailed information about the specified chat channel (such as the current number of people in the chat channel and the maximum number allowed).

Retrieving a List of Chat Channels

Calling **MediusGetChannels()** (and **MediusGetChannels_ExtraInfo()**) will return a list of chat channels accessible by the current *ApplicationID*. The Medius platform will retrieve the list of chat channels, returning them one-at-a-time to the client. The *EndOfList* field is set to 1 when the last chat channel has been returned.

If more detail is required about individual chat channels, **MediusGetChannelInfo()** can be called to retrieve all available information about the specified channel. Calling **MediusGetChannels_ExtraInfo()** will return information that otherwise would require calls to both **MediusGetChannels()** and **MediusGetChannelInfo()**.

The Medius platform is capable of hosting chat channels for multiple game applications simultaneously. However, users from one application will not be able to see chat channels from another application.

Controlling the Number of Chat Channels Returned

In order to manage bandwidth usage, you can regulate the number of chat channels returned to the requesting client by using the *PageID* and *PageSize* fields of the **MediusChannelListRequest** structure.

Example:

Set *PageSize* = 10 and *PageID* = 1 to get chat channels (1–10).

Set *PageSize* = 10 and *PageID* = 2 to get chat channels (11–20).

Set *PageSize* = 10 and *PageID* = 3 to get chat channels (21–30).

Note: Currently, the chat channel list returned is in no particular order. If you need the list in a particular order, buffer the list into your own collection type and sort it.

Retrieving a List of Players in a Chat Channel

Calling **MediusGetLobbyPlayers()** will return a list of all of the players connected to a specified chat channel. The Medius platform will retrieve the list of players, returning them one-at-a-time to the client. The *EndOfList* field is set to 1 when the last player has been returned. If more detail is required about individual players, **MediusGetPlayerInfo()** can be called to retrieve all available information about the specified player.

The lighter-weight versions of this function – **MediusGetLobbyPlayerNames()** and **MediusGetLobbyPlayerNames_ExtraInfo()** – reduce bandwidth by returning only the ‘names’ of the players connected to a specified chat channel.

Note: Currently, the player list returned is in no particular order. If you need the list in a particular order, buffer the list into your own collection type and sort it.

Getting Player Object Information

Given an *AccountID*, the **MediusGetPlayerInfo()** function will return detailed information about the specified player (no personal account profile information is transmitted). Response message information includes:

- *AccountName* – Name of the player with the given *AccountID*.
- *ApplicationID* – Current game that the player is playing. Your application would need to know other *ApplicationIDs* to use this information. For example, Soccer, Football, and Baseball applications may share the same Medius platform but each have their own *ApplicationID*; thus, an “Online Community” is created for these sports titles.
- *PlayerStatus* – Indicates if the specified player is disconnected, in a chat channel, or in-game.
- *ConnectionClass* – Indicates if the given player has a Modem or Broadband connection. This is of the type *MediusConnectionType* (set during **MediusSessionBegin()**).
- *Stats* – Current *Stats* info for the player. Used for keeping track of x-number of score fields, and/or packing special data (perhaps to display lobby avatar information).

Getting the Total Chat Channel Count

The **MediusGetTotalChannels()** function will return the total number of chat channels created for a given *ApplicationID*. This will return all chat channels running on each MLS.

Note: The total count of chat channels will only be for a given “region”, even if your application supports co-location. Because *LocationID* is stored in the user’s ‘player object’, the location will be calculated automatically.

Getting the Total Player Count

The **MediusGetTotalUsers()** function will return the total number of users connected to the Medius platform for a given *ApplicationID*.

Note: The total count of users will only be for a given “region”, even if your application supports co-location. Because *LocationID* is stored in the user’s ‘player object’, the location will be calculated automatically.

Chat Channel Security

The **MediusGetWorldSecurityLevel()** function (with *MediusApplicationType* set to “LobbyChatChannel”) allows you to determine the security state of the specified chat channel. The response message will return a *MediusWorldSecurityLevelType* field that indicates if the chat channel is:

- Public – anyone may join
- PasswordProtected – **MediusJoinChannel()** requires a password “LobbyChannelPassword” to be submitted, or
- Closed – no further **MediusJoinChannel()** calls will succeed on this given chat channel.

Note: A password can be set on a chat channel only during creation, when calling **MediusCreateChannel()**. Setting *LobbyPassword* to a non-empty string will automatically set the chat channel’s security state to *MediusWorldPasswordProtected*.

Joining a Chat Channel

The **MediusJoinChannel()** function sends a request to join an existing chat channel.

To join a chat channel, do the following:

1. Obtain the *MediusWorldID* of the chat channel to be joined from a previous call to **MediusGetChatChannels()** or **MediusCreateChannel()**.
2. Populate the **MediusJoinChannelRequest** structure and call **MediusJoinChannel()**. The server response will include the *NetConnectionInfo* structure to be used when reconnecting to the MLS. Because a single MUM may have x-number of MLSs connected to it, chat channels may span across many machines. This is why you need to disconnect, then reconnect.
3. Call **MediusDisconnect()** to disconnect from the current MLS and chat channel.
4. Call **MediusConnect()** to reconnect to a specified MLS and chat channel, passing in the *NetConnectionInfo* from the **MediusJoinChannelResponse** message.
5. (**Very important**) Populate the **MediusUpdateUserState** structure with *UserAction* set to **JoinedChatWorld**, and call **MediusUpdateClientState()**. (Do this every time you go from one chat channel to another.)

Sending a Chat Message (Broadcast or Instant Messaging)

The **MediusSendGenericChatMessage()** function allows players to send chat messages to other players. To send a chat message, do the following:

1. Populate the **MediusGenericChatMessage** data structure. Set *MessageType* = *Broadcast* to send a message to all players in the same chat channel. Set *MessageType* = *Whisper* and *TargetAccountID* = *AccountID* of the target player to send a directed message to a single player.
2. Call **MediusSendGenericChatMessage()** to send the chat message.
3. The player receiving the chat message will have their **MediusTypeGenericChatFwdMessageCallback** triggered.

Notes: If you wish to disable “Whisper” chat messages when a player is in-game, disable “Whisper” chat with **MediusGenericChatSetFilter()** – You can experiment with **MediusReassignCallbackGenericChatFwdMessageCallback** if you wish to have these chat messages handled differently based on the ‘state’ of your application (for example, different ‘zones’ of your application may handle chat differently).

If your application requires server-side vulgarity filter checks on chat messages (using **MediusTextFilter()**), see the “Vulgarity Filtering” section of this document.

Receiving a Chat Message

The callback function that handles the incoming chat message (**MediusTypeGenericChatFwdMessageCallback**) is registered at initialization using **MediusInitialize()** or **MediusInitializeBare()**. The **MediusGenericChatFwdMessage** response message will contain the chat message, the *OriginatorAccountID* and *OriginatorAccountName* of the source of the message, and the message type (Broadcast/Whisper). This callback can be changed at any time by calling **MediusReassignCallbackGenericChatFwdMessageCallback()**.

Chat Message History

The **MediusSetAutoChatHistory()** function sets the number of historical chat messages to automatically receive when joining a chat channel. For example, if **AutoChatHistoryNumMessages** is set to 5, then when the user joins a chat channel, they will see the last 5 chat messages that were posted on that chat channel so that a user would have some idea of what the current conversation entails.

MediusSetAutoChatHistory() should be set after a **MediusSessionBegin()** and after **MediusSetLocalizationParms()** so that the application properly receives the chat message history when the application disconnects from the MAS server and connects to the MLS server for the first time.

Only broadcast chat messages within the given chat channel are kept for historical purposes. Whisper chat, buddy chat, and clan chat are not kept in the per-channel history buffers.

The application’s design should decide up front what value **AutoChatHistoryNumMessages** will be so that all versions of the application are consistent. Recommend values are between [0..10].

The chat message history is received by the client via the **MediusTypeGenericChatFwdMessageCallback** (just like normal chat messages). Each **MediusGenericChatFwdMessage** has a *TimeStamp* field; therefore, it is up to the application to decide if it wishes to display a chat message that is older than a given time-frame. The function **MediusGetServerTime()** is called to get the “current time”.

Filtering Chat Channels

The **MediusSetLobbyWorldFilters()** function will filter the list of lobby worlds (chat channels) returned by a call to **MediusGetChannels()**. Filterable attributes are set one time and one time only during a **MediusCreateChannel()** call (by setting *GenericField1*, *GenericField2*, *GenericField3*, and *GenericField4*). Filterable attributes can be set for default chat channels in the MLS server’s configuration file as well.

In a system with a large number of lobby worlds (1000+), the impact of calling **MediusGetChannels()** in its current implementation not only has a tremendous performance impact on the system but also causes the Medius Client to receive much more information than it really needs. This should be avoided, and result sets for a list of lobby worlds should be more discriminating (that is, the result set returned by a request for a list of lobbies needs to return only those worlds that match specified criteria).

The process for setting a chat channel filter is as follows:

1. You must determine what *MediusLobbyFilterMaskLevelType* is to be used. Once set, this can never be changed. It is a contract between the client and the server for the production lifetime of your title. The *FilterMaskLevelType* correlates to which Generic Fields of a Lobby World (chat channel) are used for filterable attributes. The filter takes place by performing binary operations on 32, 64, 96, or 128 bits. If you determine that only one Generic Field is to be used for filtering, then you have 32-bit resolution for our binary operations. If all four Generic Fields are to be used for filtering, then you have 128-bit resolution. Note, however, that a Chat Channel's Generic Fields are just that; thus, some titles may wish to place intrinsic value in these Generic Fields (for example items that trigger special icons on the GUI).
2. Populate the **MediusSetLobbyWorldFilterRequest** structure and call **MediusSetLobbyWorldFilters()**. The server response will include **MediusCallbackStatus**.
Note: You can safely ignore every field of the response message except for *StatusCode* and *MessageID*. The other fields are there to echo back to the client the current filters set. This is done to safely persist internal state in the Medius client library in the event of disconnection/reconnection with the Medius platform during a jump between chat channels.
3. Call **MediusGetChannels()** and review the list of lobby worlds (chat channels) returned. Verify that various chat channels did get filtered (in or out is based on **MediusLobbyFilterType**).

Filtering for game lists is done a bit differently than filtering for chat channels because it is based on setting operations that perform searches on a dynamically changing game list. Chat channel filtering, however, is a bit more simplistic – it just passes a bitwise-AND or bitwise-OR operator by comparing each bit of the Chat Channel's *GenericField* specified by *MediusLobbyFilterMaskLevelType* and what is currently set for *FilterMask1*, *FilterMask2*, *FilterMask3*, and/or *FilterMask4* in the **MediusSetLobbyWorldFilterRequest** structure.

Game Management

After a user is fully logged in the Medius platform, they will be assigned to a default chat channel from which they can fully utilize “lobby” technology. Many different actions can be performed within a lobby chat channel. Additional to the features discussed in the “Chat Channel Management” section, players can request a list of players or games and/or search for a particular player or game – from which (perhaps as a result of interacting with the application’s “Online Community”) they can decide to create a new game or join an existing game.

The following Medius functions associated with ‘Game Management’ are described in this section:

- **MediusBanPlayer()** – Prevents a player from joining a given ‘game world object’
- **MediusCreateGame()** – Creates a ‘game world object’ for C/S
- **MGCLCreateGameOnMeRequest()** – Creates a ‘game world object’ for P2P
- **MediusFindPlayer()** – Searches for a player by *AccountID* or *AccountName*
- **MediusFindWorldByName()** – Searches for a ‘game world object’ by name
- **MediusGetGameInfo()** – Gets detailed ‘game world object’ information
- **MediusGetGamePlayers()** – Gets ‘game world object’s list of players
- **MediusGetGames()** – Gets a list of ‘game world objects’
- **MediusGetGames_ExtraInfo()** – Gets a list of ‘game world objects’
- **MediusGetPlayerInfo()** – Gets player information by *AccountID*
- **MediusGetTotalGames()** – Gets a total count of ‘game world objects’
- **MediusGetTotalUsers()** – Gets a total count of players
- **MediusGetWorldSecurityLevel()** – Does the ‘game world object’ have a password?
- **MediusJoinGame()** – Requests to join a DME ‘game world’
- **MediusVoteToBanPlayer()** – Vote to ban player out of a game

Preventing a Player from Joining a Game

The **MediusBanPlayer()** function (with *MediusApplicationType* set to ‘Game’) allows you to provide logic in your application to prevent a specific player from joining a given game. Setting a ‘ban’ on a particular player will prevent the player from receiving a successful **MediusJoinGame()** response for a given ‘game world object’. Currently, there is no “Kick” feature (although a “Kick” feature can be implemented directly at the DME layer while in-game). This ‘ban’ can take place for a specified number of seconds or for the lifetime of the ‘game world object’.

Searching for a Player

The **MediusFindPlayer()** function searches for another player by either *AccountID* or by *AccountName* (exact string match). The response message will return the *MediusWorldID* and *MediusApplicationType* to determine if the player is in a chat channel or in-game.

Searching for a Game

The **MediusFindWorldByName()** function (with *MediusFindInWorldType* set to “FindInGameWorld”) searches for a ‘game world object’ by *Name* (exact string match). The response message will return the *MediusWorldID* of the ‘game world object’ and other detailed information.

Getting Game World Object Information

Given a *MediusWorldID*, the **MediusGetGameInfo()** function will return detailed information about a 'game world object', which is the Medius platform's representation of an active DME 'game world'. Detailed game information may include the following (which may change at runtime due to 'world reports'):

1. The current *GameLevel* (which can change from, for example, level 1 to level 2)
2. The required *PlayerSkillLevel* (for example, novice or expert)
3. The current *PlayerCount* (calculated from the number of incoming 'player reports')
4. The current *GameStats* – your application may display special UI graphics based on this field
5. The *GameName*
6. The *RulesSet* (for example, friendly fire enabled/disabled)
7. *GenericField(1,2,3)* – use of this information is application-specific
8. The *WorldStatus* indicates that the DME 'game world' is in one of the following states : Inactive, Staging, Active, or Closed
9. The *GameHostType* indicates the type of game: client/server, integrated server, peer-to-peer, or LAN

Retrieving a List of Players in a Game World Object

Calling **MediusGetGamePlayers()** will return a list of all of the players associated with a specified 'game world object'.

Players connected to a DME 'game world' are required to send 'player reports'; these reports are routed to the 'game world object' (using *MediusWorldID*) residing on the Medius platform. When the 'game world object' receives these reports, the user's 'player object' is added to the 'game world object's list of players in this world. If a player connected to a DME 'game world' never sends a player report, the 'game world object' will never know that the player is connected and the Medius platform (lobby) will not have accurate information about the actively running DME 'game world's state. A 'game world object' will remove a 'player object' from its player list if the user either stops calling **MediusUpdate()** or if **MediusUpdateClientState()** is called with *LeftGameWorld*.

The Medius platform will retrieve the list of players, returning them one-at-a-time to the client. The *EndOfList* field is set to 1 when the last player in the list has been returned.

Note: Currently, the player list returned is in no particular order. If you need ordering performed, buffer the list into your own collection type and sort it.

Retrieving a List of Game World Objects

Calling **MediusGetGames()** (and **MediusGetGames_ExtraInfo()**) will return a list of 'game world objects' accessible by the current *ApplicationID*. The Medius platform is capable of listing 'game world objects' for multiple game applications simultaneously. However, users from one application will not be able to see other 'game world objects' from another application.

The Medius platform will retrieve the list of 'game world objects', returning them one-at-a-time to the client. The *EndOfList* field is set to 1 when the last 'game world object' in the list has been returned.

If more detail is required about individual 'game world objects', a **MediusGetGameInfo()** call can be performed to retrieve all available information about the specified 'game world object'. By calling **MediusGetGames_ExtraInfo()** you can get information that would otherwise calls to both to **MediusGetGames()** and **MediusGetGameInfo()**.

Controlling the Number of 'Game World Objects' Returned

You can regulate the number of 'game world objects' returned to the requesting client (to manage bandwidth usage) with the *PageID* and *PageSize* fields of the **MediusGameListRequest** structure.

Example:

Set *PageSize* = 10 and *PageID* = 1 to get 'game world objects' (1–10).

Set *PageSize* = 10 and *PageID* = 2 to get 'game world objects' (11–20).

Set *PageSize* = 10 and *PageID* = 3 to get 'game world objects' (21–30).

And so on...

Note: Currently, the 'game world object' list returned is in no particular order. If you need some sort of ordering performed, buffer the list into your own collection type and sort.

Getting Player Object Information

The **MediusGetPlayerInfo()** function (via *AccountID*) will return detailed information about a given player (no personal account profile information is ever exposed). Response message information includes:

1. *AccountName* – Name of the player with the given *AccountID*.
2. *ApplicationID* – Current game that player is playing (your application would have to know other *ApplicationIDs* beforehand. (Example: Soccer, Football, and Baseball applications may share the same Medius platform but each would have their own *ApplicationID*; thus, an "Online Community" is created for these sports titles.)
3. *PlayerStatus* – Indicates if a player (via *AccountID*) is Disconnected, in a chat channel, or in-game.
4. *ConnectionClass* – This is of the type *MediusConnectionType* (set during **MediusSessionBegin()**) which indicates if the given player has a Modem or Broadband connection.
5. *Stats* – Current Stats info for that player. Can be used for keeping track of x-number of score fields, and/or packing special data (perhaps to display lobby avatar information).

Getting the Total Game World Object Count

The **MediusGetTotalGames()** function will return the total number of 'game world objects' created for a given *ApplicationID*. This will return all 'game world objects' running on each MPS.

Note: If your application supports co-location, then the total count of 'game world objects' will only be for a given "region" (since *LocationID* is stored in the user's 'player object', this will be calculated automatically).

Getting the Total Player Count

The **MediusGetTotalUsers()** function will return the total number of users connected to the Medius platform for a given *ApplicationID*.

Note: If your application supports co-location, then the total count of users will only be for a given "region" (since *LocationID* is stored in the user's 'player object', this will be calculated automatically).

Game World Object Security

The **MediusGetWorldSecurityLevel()** function (with *MediusApplicationType* set to "Game") will allow us to determine the security state of the given 'game world object'. The response message will return a *MediusWorldSecurityLevelType* field that indicates if the 'game world object' is: *Public* (anyone may join), *PasswordProtected* (**MediusJoinGame()** requires a password "*GamePassword*" to be submitted), or *Closed* (no further **MediusJoinGame()** calls will succeed on this given 'game world object'). A successful

MediusJoinGame() response message contains the *NetConnectInfo* of the DME ‘game world’ you wish to **NetConnect()** with.

Note: A ‘game world object’s password is initially set during creation time via **MediusCreateGame()** or **MGCLCreateGameOnMeRequest()**. A ‘game world object’s password can change while a game is in progress due to a change of the password field as part of an incoming ‘world report’.

Creating a Game

Creating an online game is achieved through the Medius API via either the **MediusCreateGame()** or **MGCLCreateGameOnMeRequest()** calls. Based on whether your application uses either a “Client/Server” or a “Peer-to-Peer” network topology, refer to the following diagrams (Figure 2 and Figure 3) in the “Reporting (Maintaining Medius Platform Connections and Objects)” section to help in the discussion of creating games.

If Creating a Client/Server Game

1. A DME ‘game world’ is a single instance of a game that resides on a DME Game Server (GS). Thus, a client can only be connected to a single DME ‘game world’ on a GS at any one time.
2. A GS hosts a collection of DME ‘game worlds’ that are indexed by “*WorldID*”.
3. A call to **MediusCreateGame()** causes the MUM to manage the transaction of creating/starting a ‘game world’ on a GS.
4. If the transaction was successful, a corresponding ‘game world object’ will be created on the MUM and a “*MediusWorldID*” will be generated to index this object.
5. If no players join this game within 120 seconds, this ‘game world object’ will automatically timeout. The user that called **MediusCreateGame()** will still need to call **MediusJoinGame()** to join the DME ‘game world’ they just created.
6. A MUM ‘game world object’ encapsulates three things: Information about an active DME ‘game world’ running on a GS, connection information to this ‘game world’, and a list of players connected to this ‘game world’. The accuracy of the data encapsulated within a ‘game world object’ (after being created) is dependent on the various report messages coming from all the clients connected to the corresponding active ‘game world’.

If Creating a Peer-to-Peer Game

1. A DME ‘game world’ is a single instance of a game that resides on a peer-to-peer ‘host’. This ‘game world’ is created with a call to **NetHostPeerToPeer()** (which is used both for Online and LAN peer-to-peer games).
2. A peer-to-peer ‘host’ theoretically can host a collection of DME ‘game worlds’ that are indexed by “*WorldID*”; but in practice, a peer-to-peer ‘host’ will only ever have 1 active ‘game world’ running (with *WorldID* set to 1).
3. A call to **MGCLCreateGameOnMeRequest()** causes the MUM to manage the transaction of registering/posting a peer-to-peer host’s ‘game world’ with the Medius platform.
4. If the transaction was successful, a corresponding ‘game world object’ will be created on the MUM and a “*MediusWorldID*” will be generated to index this object.
5. Unlike a Client/Server game, the player that called **NetHostPeerToPeer()** will not need to call **MediusJoinGame()** to connect to the DME ‘game world’ created on the host (since the hosting player is assumed to be already connected to itself).
6. A MUM ‘game world object’ encapsulates three things: Information about an active DME ‘game world’ running on a peer-to-peer ‘host’, connection information to this ‘game world’, and a list of players connected to this ‘game world’. As above, the accuracy of the data encapsulated within a ‘game world object’ (after being created) is dependent on the various report messages coming from all the clients connected to the corresponding active ‘game world’.

Joining a Game

The **MediusJoinGame()** function is used to request connection information from a 'game world object' (via *MediusWorldID*) to join an existing DME 'game world'.

Joining Either a Client/Server Game or a Peer-to-Peer Game

For both client/server games and peer-to-peer games, a player in the application's lobby can get a list of games (via **MediusGetGames()** – which enumerates all the current 'game world objects') then requests to join a given 'game world' (via **MediusJoinGame()** – using the respective *MediusWorldID*). If the join request was successful, the DME 'game world's connection information (*NetConnectionInfo*) will be returned so that a **NetConnect()** to this 'game world' can take place.

Note: If the corresponding **NetConnect()** to the DME 'game world' is successful, then the client will have two simultaneous connections established – connected both to an MLS and to a GS (or peer-to-peer 'host').

It is important to note that the Medius Connection is maintained throughout the online session and is used for sending game-related report messages to the Medius platform while a player is in-game.

If the Medius Connection is lost, in-game play may continue, but once that game has completed the user will need to initiate a new connection to the MAS to begin a new session.

Vote Player Out of a Game

The **MediusVoteToBanPlayer()** function provides the client with the ability to vote players out of a game who are observed to be cheating or otherwise undermining the spirit of good game play.

While a game is in progress, players that observe another player cheating or otherwise behaving undesirably can cast a vote with Medius to kick the player out of the game and ban him/her from re-joining the game. A client can only submit one vote per player, i.e., a player cannot submit multiple votes for the same offending player. Vote counts for each player are maintained by Medius. When a vote comes in for a player and his/her vote count is incremented, a check is made if the updated vote count has exceeded a specified threshold. Threshold is a configurable percentage of the current player count in the game. If the threshold is reached, the player will be banned from the game (and added to the game's banned player list). Also, a player disconnect message will be sent from Medius to either the DME Game Server (for client-server games) or to the peer-to-peer Host, which in turn will forcefully close the player's game connection.

Design Considerations

When leveraging **MediusVoteToBanPlayer()** functionality, take these issues into consideration:

1. A client side ban vote tallying system must still be managed by the client application and therefore employ whatever voting criteria is valid for the game. When a vote is cast, use a DME Net Message to send a message to all the players in a given game. Each application should manage their own tally count and it is up to the application if they wish to display to the offending player's current ban vote count.
2. If and when a ban vote count threshold has been reached to ban a player, all clients currently in the game should call **MediusVoteToBanPlayer()** to submit their 'vote' to ban the player from the game. Having ALL clients submit their vote concurrently, 'distributes' the responsibility of kicking a player across all clients instead of a single client which would have all sorts of problems. Thus, in the case of a 20-player game, each client application will have their own tally count, and if (say) 11 votes to ban a player is received (the determined client side threshold) over a period of time, each application will automatically call **MediusVoteToBanPlayer()** (all 20 clients "even the player's client application that is being reported as the offending client" will call this function to ensure the ban takes place). Even if the player has 'hacked' the game with a cheating device to prevent their **MediusVoteToBanPlayer()** to be called, the other 19 clients will still succeed. In practice, this works very well. Even if a number of

clients disconnected of their own free will right when this happened, the server side ban count threshold will scale due to the active user count.

3. The **MediusVoteToBanPlayer()** function is NOT meant to support allowing players to randomly submit votes to Medius to ban a player for whatever reason they want throughout the life of a game. This is too open to abuse – there will always be abusers and we are not signing on to provide the 'perfect, abuse-free, game-generic' solution.
4. The **MediusVoteToBanPlayer()** function is a tool to enforce a ban on a player (once the game's vote tally system has been triggered to actually ban a player) and prevent them from being able to rejoin a game once disconnected.
5. For client-server applications, if a player has been banned the DME Game Server will auto-disconnect the player from the game.
6. For peer-to-peer applications, there is a *DmeClientIndex* as part of **MediusServerJoinGameResponse** (MGCL API) that the host needs to set for incoming clients attempting to join (this is the index of the reserved player slot as seen by the DME). **MGCLInitializeInParams** also has a **MGCLServerDisconnectPlayerCallback** that the peer-to-peer host will get triggered if the Medius servers have declared it is time to ban a player from the game. For peer-to-peer applications, if a player has been banned the peer-to-peer host will have their **MGCLServerDisconnectPlayerCallback** triggered with the *DmeClientIndex* of the player that needs to be disconnected. The host should then send a DME Net Message to that player telling them to disconnect themselves from the game.

Account Management

The following Medius functions associated with 'Account Management' are described in this section:

- **MediusAccountDelete()** – Requests to delete an existing account
- **MediusAccountGetID()** – Gets *AccountID* from *AccountName*
- **MediusAccountGetProfile()** – Gets an account profile
- **MediusAccountUpdatePassword()** – Sets an account password
- **MediusAccountUpdateProfile()** – Sets an account profile
- **MediusAccountUpdateStats()** – Sets an account stats

Deleting an Account

The **MediusAccountDelete()** function is used to request to delete the currently logged-in account.

Getting an Account ID

The **MediusAccountGetID()** function is used to determine the *AccountID* of a player with a given *AccountName*.

Getting an Account Profile

The **MediusAccountGetProfile()** function is used to retrieve profile information of the currently logged-in account. Each application decides if they wish to have account profiles persisted for each user. Profiles are nice to have for creating mailing lists, etc.

Updating an Account Password

The **MediusAccountUpdatePassword()** function is used to update the password of the currently logged-in account.

Updating an Account Profile

The **MediusAccountUpdateProfile()** function is used to update the profile of the currently logged-in account. Each application decides if they wish to have account profiles persisted for each user. Profiles are nice to have for creating mailing lists, etc.

Updating Account Stats

The **MediusAccountUpdateStats()** function is used to update the *Stats* field of the currently logged-in account. Each application will determine what value the *Stats* field is used for. For example, *Stats* can be used to keep track of wins or losses and/or can also be used in the case if you display special "lobby" avatar information.

Buddy Lists

Every user account has a buddy list (anonymous users do not), which is stored in the Medius database and is managed through the following Medius API calls:

- **MediusAddToBuddyList()** – Adds a player to buddy list
- **MediusBuddyAddConfirmation()** – If confirmation is required, grants permission
- **MediusBuddyGetPermission()** – If confirmation is required, asks for permission
- **MediusGetBuddyInvitations()** – Gets buddy invitations
- **MediusGetBuddyList()** – Gets players in buddy list
- **MediusGetBuddyList_ExtraInfo()** – Gets players in buddy list
- **MediusRemoveFromBuddyList()** – Removes a player from buddy list

Retrieving a Player's Buddy List

1. Call **MediusGetBuddyList()** (or **MediusGetBuddyList_ExtraInfo()**).
2. The Medius platform will query the database for all buddies associated with the current player. The server will then respond with one or more **MediusGetBuddyListResponse** messages. The response message for each buddy will indicate if they are:
 - a. *Disconnected* – User is not logged in the Medius platform.
 - b. *InAuthWorld* – User's 'player object' is not yet in an authenticated state.
 - c. *InChatWorld* – User's 'player object' is in a lobby state.
 - d. *InGameWorld* – User's 'player object' is in an in-game state.

Adding a User to a Player's Buddy List

1. Populate the **MediusAddToBuddyListRequest** structure with the *AccountID* of the user to be added.
2. Call **MediusAddToBuddyList()**. The Medius platform will add the user associated with the *AccountID* in the request message to the buddy list of the user associated with the *SessionKey* field in the request message.

Removing a User From a Player's Buddy List

1. Populate the **MediusRemoveFromBuddyListRequest** structure with the *AccountID* of the user to be removed.
2. Call **MediusRemoveFromBuddyList()**. The Medius platform will remove the user associated with the *AccountID* in the request message from the buddy list of the user associated with the *SessionKey* field in the request message.

Buddy List Permissions

Optionally (as designed by the application developer), a player (source) can be required to obtain permission from another player (target), prior to adding that player to a buddy list (both players must be online). This process consists of sending a request message for permission to the target player who will then accept or deny the request. The MLS forwards the target player's response message to the source player. If the request for permission was accepted, the source player proceeds to call **MediusAddToBuddyList()** as described above.

Requesting Permission

1. Populate the **MediusBuddyGetPermissionRequest** structure with the (target) *AccountID* of the desired user to request permission.
2. Call the **MediusBuddyGetPermission()** function.

Responding to a Request for Permission

(If currently online) The callback function that handles a forwarded request message for permission (**MediusTypeAddToBuddyListFwdConfirmationRequestCallback**) is registered at initialization via **MediusInitialize()** or **MediusInitializeBare()**.

Note: It is general practice to reassign the callback function above to NULL when the game starts, unless your game can handle buddy list invitations while in game.

(If offline, then online) Call **MediusGetBuddyInvitations()** to request a list of buddy list invitations while offline. Players that call **MediusBuddyGetPermission()** with account information of a player that is offline will be queued in the Medius Database for the offline user (queue max size is currently 64).

When the request for permission (**MediusAddToBuddyListFwdConfirmationRequest**) message is received by the targeted player, the following message should appear in your application's GUI:

Allow "Player's Name" to add "You" to their buddy list? (Accept/Deny)

The targeted player will then populate the **MediusAddToBuddyListFwdConfirmationResponse** data structure with **MediusCallbackStatus** = *RequestAccepted* or *RequestDenied*.

To send the target player's response message, call **MediusBuddyAddConfirmation()**. The server will forward this message to the source player.

Ignore Lists

Every user account has an ignore list (anonymous users do not), which is stored in the Medius database and is managed through the following Medius API calls:

- **MediusAddToIgnoreList()** – Adds a player to ignore list
- **MediusGetIgnoreList()** – Gets players in ignore list
- **MediusRemoveFromIgnoreList()** – Removes a player from ignore list

A player will not receive Medius chat messages from users in their ignore list.

Retrieving a Player's Ignore List

1. Call **MediusGetIgnoreList()**.
2. The Medius platform will query the database for all players ignored associated with the current player. The server will then respond with one or more **MediusGetIgnoreListResponse** messages. The response message for each ignored player will indicate their *AccountID* and *AccountName* as well as if they are:
 - a. *Disconnected* – User is not logged in the Medius platform.
 - b. *InAuthWorld* – User's 'player object' is not yet in an authenticated state.
 - c. *InChatWorld* – User's 'player object' is in a lobby state.
 - d. *InGameWorld* – User's 'player object' is in an in-game state.

Adding a User to a Player's Ignore List

1. Populate the **MediusAddToIgnoreListRequest** structure with the *AccountID* of the user to be added.
2. Call **MediusAddToIgnoreList()**. The Medius platform will add the user associated with the *AccountID* in the request message to the ignore list of the user associated with the *SessionKey* field in the request message.

Removing a User From a Player's Ignore List

1. Populate the **MediusRemoveFromIgnoreListRequest** structure with the *AccountID* of the user to be removed.
2. Call **MediusRemoveFromIgnoreList()**. The Medius platform will remove the user associated with the *AccountID* in the request message from the ignore list of the user associated with the *SessionKey* field in the request message.

Medius Dynamic List System

Introduction

The Dynamic List service (DList) is designed to provide an API that allows the game developer a way to subscribe to the various list content available through the Medius server set.

This system is a departure from the style of previous Medius APIs. In order to provide a completely customizable content delivery mechanism, DList was built upon a simple meta-object protocol. This allows the game developer to subscribe to specific lists (e.g., CLAN, BUDDY, etc.) and to choose which data fields and events the game receives.

The DList service delivers events and data from multiple non-synchronized data and event sources. Because of this, there will be times when rows will be partially populated until the data is available.

Interface

Since DList is a meta-object protocol, the API has been tailored to reflect the nature of the object system. Each object within the protocol is accessed through an interface structure.

```
typedef const struct
{
    ...
    MediusBool      (*refresh)      (MediusDListSubscription *pSubscription);
    ...
} MediusDListInterface;
```

In order to manipulate an object, the developer must first obtain a pointer to the interface implementation.

```
MediusDListInterface *gpDListInterface;
int main()
{
    gpDListInterface = GetMediusDListInterface();
    ...
}
```

The developer can then utilize the interface to manipulate and interact with the meta-objects and the Medius Dynamic List service.

```
MediusBool doBuddyRefresh()
{ return gpDListInterface->refresh( pMyList ); }
```

Type System

The Dynamic List service (DList) is built upon a simple meta-object protocol. Objects within this protocol are specified and accessed via the MediusDType system (DType).

A DType combines the DMetaType defined by SCE-RT with a field map (MediusDFieldMap) that the developer uses to describe how to access a game object.

After you have created a valid field map, the developer creates a DType that can be used throughout his program to interact with DList objects.

```
MediusDType *gpMyBuddyObjDType
= gpDTypeInterface->create ( GetMediusDPlayerListMetaType()
, &MyBuddyListFieldMap );
```

While creation of a DType is relatively efficient, it does involve walking the supplied field map and allocating a DType object. To reduce this overhead, you should allocate types early and re-use them throughout the life of your game.

Note: As with all resources that are allocated, a type must be destroyed when it is no longer needed. A type references internal Medius data structures and is therefore only valid AFTER a call to the **MediusInitialize()** and BEFORE the call to **MediusClose()**. Every DType should be freed BEFORE calling **MediusClose()**.

With the created type, you can now subscribe to a list:

```
gpDListInterface->subscribe    ( &pSubscription
                                , &transactionId
                                , MEDIUS_DLIST_BUDDY
                                , 0
                                , 0
                                , gpMyBuddyObjDType
                                , MEDIUS_DLIST_ALL_EVENTS
                                , pMyUserCallback
                                , pUserData );
```

Then, update your game object from an argument list:

```
gpDArgListInterface->popRowData ( pArgList
                                , gpMyBuddyObjDType
                                , pMyBuddyListObject);
```

The following is a more complete example that includes how to define a field map reference.

Callbacks

After a subscription is established, the game will receive solicited and unsolicited events related to that subscription on the registered callback.

The user callback of the game needs to address several different reactions to events that are delivered from the server.

```
void MyCallback ( MediusDListAction action
                  , MediusCallbackStatus status
                  , MediusTransactionId transactionId
                  , MediusExceptionEvent *pExcept
                  , unsigned short    rowsLeft
                  , MediusDRowId      rowId
                  , MediusDArgList    *pArgList
                  , void *            pUserData )
{
    MyList *pList = pUserData;
    MyRow *pRow = NULL;
```

The following are exceptions that may dictate specific reactions to errors and exceptional events from the server related to this or to all subscriptions.

```
    if (pExcept)
    {
        Log("Received exception extent %d error %d",pExcept->exceptExtent, pExcept->error);
        switch(pExcept->exceptExtent)
        {
            case MEDIUS_EXCEPT_NONE:
                Log("This should never happen");
                break;
            case MEDIUS_EXCEPT_ABORT_TRANSACTION:
                MyMarkTransactionForRetry(pList, transactionId, pExcept->retryTimeout);
```

```

        break;
    case MEDIUS_EXCEPT_FAIL_TRANSACTION:
        MyMarkTransactionFailed(pList, transactionId);
        break;
    case MEDIUS_EXCEPT_ABORT_CONTEXT:
        MyMarkListBad(pList);
        break;
    case MEDIUS_EXCEPT_FAIL_CONTEXT:
        MyMarkListBad(pList);
        MyMarkSessionBad();
        break;
    case MEDIUS_EXCEPT_FAIL_SESSION:
        MyMarkListBad(pList);
        MyMarkSessionBad();
        break;
    default:
        Log("A new exception that we don't handle");
    }

    return;
}

```

This depicts tracking of transaction ids to ensure request/response time.

```

if (transactionId)
{
    MyStopWaiting(pList, transactionId);
}

```

The manipulation of the game's list object is related to the MEDIUS_DACTION enumeration sent from the server. This action dictates how the *pArgList* containing the data sent from the server would be applied to the game's list object.

```

if ( rowsLeft > 0 )
    MyMarkListInUpdate(pList);

if (pArgList)
{
    switch(action)
    {
    case MEDIUS_DACTION_NOEVENT:
        Log("This should never happen!");
        break;
    case MEDIUS_DACTION_ERROR:
        Log("Recieved error exception handled above");
        break;
    case MEDIUS_DACTION_STATUS:
        Log("Recieved status %d", status);
        break;

    case MEDIUS_DACTION_UPDATE:
        pRow = MyFindRow( pList, rowId );
        if (pRow)
        {
            gpMediusDList->popRowData( pArgList, pList->pType, pRow );
        }
        break;
    case MEDIUS_DACTION_ADD:
        pRow = MyAllocRow( pList, rowId );
        gpMediusDList->popRowData( pArgList, pList->pType, pRow );
        break;
    }
}

```

```

        case MEDIUS_DACTION_DELETE:
            MyDeleteRow( pList, rowId );
            break;
        case MEDIUS_DACTION_REFRESH:
            pRow = MyFindRow( pList, rowId );
            if ( ! pRow )
            {
                pRow = MyAllocRow( pList, rowId );
            }
            gpMediusDList->popRowData( pArgList, pList->pType, pRow );
            break;
        default:
            Log("New action %d, we don't handle and ignore", action);
    }

}

if ( rowsLeft == 0 )
    MyMarkListOutOfUpdate( pList );

}

```

Tailoring the Subscription

The DList API provides a customizable content delivery mechanism. The content of a subscription and the events that the game receives are tailored not only through the specified service level but also the fields specified in the DType through the field map.

In addition to controlling which information will trigger a callback, the developer can associate a callback with a specific interest callback, and then he can associate that interest callback with multiple lists.

Subscribing

In order to receive events, you must describe which events you want to receive and on which callback you want to receive it.

Each subscription is relative to a single list. A specific list is indicated by its *MediusListId* and its *relationId*. While you may subscribe to multiple lists with the same listId, each must have a unique *relationId*:

(e.g. *relationId* == *clanId* for `MEDIUS_DLIST_CLAN_MEMBER`).

With a valid list indicated, specify the DType for the subscription. The DType will tell the subscription how to unpack data to your list object. It will also indicate which events you are interested in receiving. See “”.

To further specify which events to receive, indicate the service level at which you are interested. See `MediusDListServiceLevel`.

Note: A subscription is only valid while connected as a lobby server. You should unsubscribe before you change from one lobby to another, and re-subscribe when you attach to the new lobby. Since it is impossible to fully guarantee that all list events will be delivered at all times when switching lobby servers, it is necessary to re-create/subscribe to any list whenever you switch lobby servers.

Setting an Interest Callback

To filter the events and data from multiple lists, the developer can associate a callback (**pUserCallback**) with those lists along with the fields that the callback would like to see through the specified type (*pType*).

One of the most important characteristics of an interest callback is that it receives only one event, regardless of whether there are overlapping members on associated lists. For example, if a player is a member of both you Buddy and ClanMember list, and if his online status changes, an interest callback associated with both the Buddy and ClanMember list will only receive ONE event.

Note: The fields specified by *pType* are used to filter callback events rather than actually subscribe to those fields. Only events that contain all of the desired fields trigger the callback.

Associating an Interest Callback

In order for an interest callback to receive events, it must be associated with at least one subscription. Once it is associated, it allows the developer to select the specific subscriptions (within a common interest) to propagate events to the interest callback.

Usage

```
gpDListInterface->associateCB( pMyPlayerInterestCB, pMyBuddyListSubscription );
```

This will associate **pMyPlayerInterestCB** with *pMyBuddyListSubscription*, so that any events that are received by *pMyBuddyListSubscription* match the criteria associated with **pMyPlayerInterestCB**. When it is set, then the associated **pInterestCallback** will be called.

Service Levels

The service level tailors the interaction with the server to the specific events that a game is interested in for a list. If the game is not displaying the buddy list, it may only need to know when a player goes off or on line to display a message. This is accomplished by subscribing with the MEDIUS_DLIST_CHANGE_EVENTS service level.

First, you must have a list to update. To maintain this list on a consistent basis requires MEDIUS_DLEVEL_ALL_EVENTS to receive, add, and delete events. It must complete all of these functions to keep this list up-to-date.

By subscribing to MEDIUS_DLEVEL_REFRESHED, a refresh will be requested as part of the subscription process.

Game List Row Object Declaration

List row object contains all currently available interest data members.

Your player row list may have any subset of these data members and also have data members that are not related to DList. The name and order within the structure is also unimportant.

It is expected that you may have a different structure for each list type that you subscribe to (e.g., BUDDY, CLAN_MEMBER, etc.)

The Type and Size of these fields MUST match these definitions:

```
typedef struct
{
    int                PlayerID;
    int                LobbyID;
    int                GameID;
    MediusPlayerStatus PlayerStatus;
    char               playerName[ACCOUNTNAME_MAXLEN];
    char               lobbyName[LOBBYNAME_MAXLEN];
    char               gameName[GAMENAME_MAXLEN];
    unsigned char      playerStats[ACCOUNTSTATS_MAXLEN];
    unsigned char      playerOnline;
} PlayerListRow;
```

Field Specification Definition

The other way that the developer tailors the content, and to an extent the events that the game receives, is by specifying the fields the game wants to receive. This reduces the bandwidth to only what is necessary to deliver the fields in which the game is interested.

In addition, since some events are tied to specific fields, these events will also be culled, if the field is not required.

The field specification maps individual structure members to the interest fields. The specifications are checked within `gpDTypeInterface->create()` and an error will return if any specification does not match the one from these definitions.

It is expected that you will not only have a different field map, specification, and DType for each list type (e.g., BUDDY, CLAN_MEMBER etc.), but that you will also have a different set for each subset of fields for which you want to subscribe. Another example of this is a DType which is used to set an interest callback.

The order of these field definitions is important. You may use any subset of these fields. (Comment out the ones you do not need.).

Note: Remember to change the type name of the structure to your structure's type name and the structure member names to your structure's member names.

```
static MediusDFieldSpec PlayerListRow_Spec[]
={ MediusDSpecInt32 ( MEDIUS_PLIST_PLAYER_ID,      PlayerListRow_, PlayerID )
  , MediusDSpecCString( MEDIUS_PLIST_PLAYER_NAME,  PlayerListRow_, playerName )
  , MediusDSpecUInt32 ( MEDIUS_PLIST_PLAYER_STATUS, PlayerListRow_, PlayerStatus )
  , MediusDSpecInt32 ( MEDIUS_PLIST_LOBBY_ID,      PlayerListRow_, LobbyID )
  , MediusDSpecCString( MEDIUS_PLIST_LOBBY_NAME,   PlayerListRow_, LobbyName )
  , MediusDSpecInt32 ( MEDIUS_PLIST_GAME_ID,       PlayerListRow_, GameID )
  , MediusDSpecCString( MEDIUS_PLIST_GAME_NAME,    PlayerListRow_, GameName )
  , MediusDSpecUChar ( MEDIUS_PLIST_ONLINE,        PlayerListRow_, PlayerOnline )
  , MediusDSpecUChar ( MEDIUS_PLIST_PLAYER_STATS,  PlayerListRow_, PlayerStats )
};
```

Field Map Definition

The field map serves as a description of the Field Specification list to feed into `gpDTypeInterface->create()`.

Notice that the type specifier (static) is external to the **DEF_FIELD_MAP()** macro so that you may customize the visibility of the map. In practice, there should never be a reason to declare a field map as anything other than static.

```
static DEF_FIELD_MAP(PlayerListRow_FieldMap_, PlayerListRow_Spec_, PlayerListRow_);
```

DType Declaration

The type points to the field map so the field map and field specification should maintain at least the same life expectancy as the type. Usually defining the field map, specification, and static is sufficient since they do not change.

Note: Although a type may be declared static, it depends upon values within Medius. Because of this, a type is only valid between **MediusInit()** and **MediusClose()**. A type should be created AFTER **MediusInit()** and should always be destroyed BEFORE **MediusClose()**.

```
static MediusDType *pPlayerListRow_DType_ = NULL;
```


DType Creation

Note: Creating a type allocates memory that must be destroyed when no longer useful. In addition, since a type is allocated within Medius, it is advisable to create your types immediately after **MediusInit** to reduce the possibility of fragmentation.

```
static MediusDType *
GetPlayerInterestType_()
{
    if (! pPlayerListRow_DType_)
    {
        pPlayerListRow_DType_
            = gpDTypeInterface->create( GetMediusDPlayerListMetaType()
                                         , &PlayerListRow_FieldMap_ );
    }

    return pPlayerListRow_DType_;
}
```

DType Destruction

Although a type may be declared static, it depends upon values within Medius.

Because of this a type is only valid between **MediusInit()** and **MediusClose()**. A type should be created AFTER **MediusInit()** and should always be destroyed BEFORE **MediusClose()**.

```
static void
DestroyPlayerInterestType()
{
    gpDTypeInterface->destroy( pPlayerListRow_DType_ );
    pPlayerListRow_DType_ = 0;
}
```

Platform Requirement Functionality

Certain functions are required and must be used in specific ways, as described in the *Medius Platform Requirements and Guidelines* document. Refer to the *Medius Platform Requirements and Guidelines* document for specific information about these requirements, as well as the appropriate use of the functions.

After a player has created a “session” with the Medius platform (while still connected to the MAS) and before the account login (or anonymous login), the player should first be presented with the ‘Usage Policy’ then the ‘Privacy Policy’, and finally with any application specific ‘Announcements’.

Medius API functions discussed in this section:

- **MediusGetPolicy** – Gets ‘usage policy’ and ‘privacy policy’
- **MediusGetAnnouncements** – Gets all unread announcements
- **MediusGetAllAnnouncements** – Gets all announcements
- **MediusSetMessageAsRead** – Marks a particular announcement as read

Retrieving the Usage Policy and the Privacy Policy

There are two types of policies that must be present to a user before that are allowed to play online – Usage Policy and Privacy Policy. These policies are entered into the database by production and technology personnel.

Invoking **MediusGetPolicy()** with the desired policy type will return x-number of **MediusGetPolicyResponse** messages (each message is of size POLICY_MAXLEN). Buffer up these messages until you receive *EndOfText* = 1 and then display to the user.

Retrieving Announcements

Announcements are entered into the database by production and technology personnel, **not** by users. Announcements are retrieved through **MediusGetAnnouncements()** or **MediusGetAllAnnouncements()** API calls.

When a new announcement is stored in the database, it is keyed to be specific to a certain application, or applicable system-wide to all application users.

Calling **MediusGetAnnouncements()** will only retrieve unread announcements keyed to the application the user has logged in with. Alternatively, calling **MediusGetAllAnnouncements()** will return all of the application-specific messages **in addition to** all system-wide announcements (which will also then be marked as “read”).

Invoking **MediusGetAnnouncements()** will return x-number of **MediusGetAnnouncementsResponse** messages (each message is of size ANNOUNCEMENT_MAXLEN). Buffer up these messages until you receive *EndOfText* = 1 and then display to the user. This will return all unread announcements.

Invoking **MediusGetAllAnnouncements()** will return (just like above) x-number of **MediusGetAnnouncementsResponse** messages – this time, you’ll just get more response messages. This will return all announcements regardless of their ‘marked-as-read’ flags.

Invoking **MediusSetMessageAsRead()** will mark a particular announcement as read (via the *AnnouncementID* returned from **MediusGetAnnouncementsResponse**).

Co-location

Medius platform servers can be deployed in a variety of different ways to help PlayStation®2 clients find lobby servers and game servers (and peer-to-peer hosts) that may be closest to their “region”. When a client is presented with a list of “regions” (while connected to a MAS), the client should pick the “region” closest to where he/she is playing in order to (hopefully) minimize Internet traffic and improve performance by minimizing latency. The process of strategically deploying Medius platform servers in this manner is called co-location.

Medius functions that deal with co-location are:

- **MediusGetLocations()** – Gets a list of locations “regions”
- **MediusPickLocation()** – Picks a location “region”

Co-location Example

As an example of co-location, you may, for a given application:

1. Deploy 1 MUM, 1 (oracle-based) Medius database, and 4 MAS servers in San Diego. A DNS lookup of “mygame.mycompany.com” will return the IP address of the load-balancer that will direct the incoming client to 1 of the 4 MAS servers that have been set up. The application will always initially connect to the MAS servers in San Diego to establish an authenticated state and to retrieve a list of locations “regions” to pick from.
2. Deploy three MLS servers in each location; for example, three each in San Diego, Chicago, New York, and San Francisco for a total of 12 MLS servers. Each would connect to the MUM in San Diego. Each of these MLS servers would have their configuration file “medius.txt” set with a “*LocationID*” from 0-N. You may assign San Diego *LocationID*=0, Chicago *LocationID*=1, New York *LocationID*=2, and San Francisco *LocationID*=3. (You can of course add and/or remove MLS servers at anytime to or from any geographic location). After you assign *LocationIDs*, update the co-location table in the Medius database hosted in San Diego.
3. Just as above, you would match 1-to-1 an MPS for each MLS (though in practice, you would actually scale this number based on load); therefore, a MPS server would be set up in the same locations as the MLS servers and, of course, each would connect to the MUM in San Diego. Each of the MPS servers would have their configuration file “mps.txt” set with the same “*LocationID*” that you set up for the MLS servers.
4. If the application were a Client/Server game, you would also need to set up DME Game Servers at each of the locations. Each of the GS servers would have their configuration file “DmeMedius.cfg” set with the same “*LocationID*” that you set up for both the MLS and MPS servers. When the GS servers are brought online, they will initially connect to the MAS, then automatically be redirected to a MPS in that “region”.

By choosing a specific location, a request for a game/chat channel list will be limited to games/chat channels hosted by game/lobby servers in the same “region”. Similarly when creating a game/chat channel, it will be hosted by a game/lobby server with the same *LocationID* as the player who originated the request.

If the application is a Client/Server game, a user will want to choose a GS that is closest to them, to have the best possible gaming experience. This is also true if you are the ‘host’ of a peer-to-peer game, and Chicago (for example) is selected for the “region” after you logged in. If other players join the game, you know that they also picked Chicago as their “region”. Therefore, it’s possible that the round trip latencies to and from each of the peers in the game would be minimal (in contrast to peers spread across the world).

Co-location Process

MediusGetLocations() and **MediusPickLocation()** allow a client to retrieve a list of locations and select a location respectively. Once a *LocationID* has been 'picked', this will automatically be stored in the player's 'player object' on the Medius platform, and no other Medius API function will ever ask for this *LocationID* again.

Recommend process if using co-location:

1. UI prompts user for username and password (saves until actual login at step 8).
2. User connects to the MAS.
3. User starts a "session" with the Medius platform.
4. User retrieves Usage Policy, Privacy Policy, and application's Announcements.
5. Application then calls **MediusGetLocations()** to get a list of Locations. The application then would display these locations to the user.
6. User will scroll through the available locations and select one. The application would then call **MediusPickLocation()**. If success, that *LocationID* will be stored in the user's 'player object' on the Medius platform. The developer will never have to worry about passing in *LocationID* to any other Medius functions since it is automatically handled from this point on.
7. After a location has been selected, the user may now login (**MediusAccountLogin()** or **MediusAnonymousLogin()**). The response message to the login will redirect the user to a MLS setup in the "region" they selected. If the account login response message returned a failed status code, either the respective location "regional" Medius servers may not online or somehow the user got disconnected from the network. Contact SCE-RT (scert-support@scea.com) or the respective IT group that upkeeps your SCE-RT Medius servers.
8. Call **MediusDisconnect()** (from MAS) then call **MediusConnect()** with the *NetConnectInfo* of the MLS that was returned to you from the **MediusAccountLoginResponse** message.

Ladder Management

To enable Ladders for your title, you must setup a Medius Database Caching Server (MDBCS). The primary purpose of the MDBCS is to unload CPU processing and database transactions from the other Medius platform servers to support Ladder Management.

The following Medius functions associated with 'Ladder Management' are described in this section:

- **MediusGetLadderStatsWide()** – Retrieves all ladder stats for given player
- **MediusGetTotalRankings()** – Lists total number of players in ladder ranking
- **MediusLadderList_ExtraInfo()** – Retrieves a list of players in a ladder ranking
- **MediusLadderPosition_ExtraInfo()** – Gets a ladder ranking for given player
- **MediusLadderPositionFast()** – Gets the approximate new ladder ranking for a given player
- **MediusUpdateLadderStatsWide()** – Sets a player's ladder stats (to be saved to database)

Getting a Player's Current Ladder Stats and Updating

The **MediusGetLadderStatsWide()** function is used to request for a player's current ladder stats information. Each player has between [0..LADDERSTATSWIDE_MAXLEN] available ladder categories to save a score to (currently up to 100 categories are available as of the v1.50 release). Each element of the array takes an int value. To set a 'score' for a ladder category, just set a value to the respective element in the array (indexed by *LadderStatIndex*) and call **MediusUpdateLadderStatsWide()** to request to save the new ladder score to the Medius database. Normally, an update to a score is in the form of an addition or subtraction to the current score.

Note: A call to **MediusUpdateLadderStatsWide()** will persist the player's ladder stats immediately to the Medius database; however, it will typically take between 1-5 minutes (configurable) for the MDBCS to refresh it's cached ladder rank scores. Therefore, if you need an approximate new ladder ranking score for a player (such as when they finish a game and would like to see how much they may have improved their ladder ranking score) then call **MediusLadderPositionFast()** to get an idea of what their new rank will be. This function will compare the player's old score as seen in the MDBCS cached ladder rank scores and use a sliding window algorithm that will slide up or down looking at neighboring player's raw scores and will return a new position based on this lookup (it will not update the MDBCS's cached ladder ranks).

Total Number of Players in a Ladder Ranking

The **MediusGetTotalRankings()** function is used to request the total number of players in a given ladder ranking. Set the *LadderStatIndex* to get the total number of players that have a non-zero ladder rank score in the respective category.

Getting a List of Player's in a Ladder Ranking

The **MediusLadderList_ExtraInfo()** function is used to request a list of player's in a ladder ranking sorted by their rank positions that is calculated by their current rank score. Set *LadderStatIndex* to an index between [0.. LADDERSTATSWIDE_MAXLEN] to choose which ladder category you wish to get a list of players for. The player with the highest score in the given ladder rank will have a position of 1. Player's that have a rank score of zero will not be returned in this list.

Controlling the Number of 'Ladder Ranks' Returned

You can regulate the number of 'ladder ranks' returned to the requesting client (so that you can manage bandwidth usage and GUI lists) with the *StartPosition* and *PageSize* fields of the **MediusLadderList_ExtraInfoRequest** structure.

Example:

Set *PageSize* = 10 and *StartPosition* = 1 to get 'ranks' (1–10).

Set *PageSize* = 10 and *StartPosition* = 2 to get 'ranks' (11–20).

Set *PageSize* = 10 and *StartPosition* = 3 to get 'ranks' (21–30).

And so on...

Getting a Player's Current Ladder Ranking

The **MediusLadderPosition_ExtraInfo()** function is used to determine the ladder ranking of a particular player. Set *LadderStatIndex* to an index between [0.. LADDERSTATSWIDE_MAXLEN] to choose which ladder category you wish to get a position for. The player with the highest score in the given ladder rank will have a position of 1. If the player has a rank score of zero, **MediusNoResult** will be returned with position set to zero.

Token Management

The Medius Token Reservation Service provides the application the ability to reserve and release a application-defined ID or “Token” with Medius in order to restrict usage of that Token to a single entity. An entity can be either a Player Account or Clan. The token posted is guaranteed unique within its category. Once a token is reserved, subsequent calls to reserve the same token within the same category will fail until the token is released. Tokens can be added, updated, and deleted. Players can only add account-specific tokens to their own account. Currently, only clan leaders can add clan-specific tokens for their Clan. A Player or Clan leader can reserve multiple tokens.

The following Medius APIs are associated with ‘Token Management’:

- **MediusTokenCategoryType** – Valid token types
- **MediusTokenActionType** – Valid token actions
- **MediusTokenRequest** – **MediusToken()** request structure
- **MediusToken()** – Request to add/update/remove a token

Clan Management

Every user account has the ability to manage Clans (anonymous users do not), which is stored in the Medius database and is managed through the following Medius API calls:

Clan Management

- **MediusCreatClan()** – Requests Clan creation
- **MediusDisbandClan()** – Requests the disbanding of a Clan
- **MediusGetMyClans()** – Gets a list of Clan memberships
- **MediusGetClanByName()** – Gets Clan information from a given Clan name
- **MediusGetClanByID()** – Gets Clan information from a given *ClanID*
- **MediusTransferClanLeadership()** – Requests transfer of Clan leadership to another player

Clan Membership Management

- **MediusAddPlayerToClan()** – Requests player addition to a Clan
- **MediusRemovePlayerFromClan()** – Requests player removal from a Clan
- **MediusGetClanMemberList()** – Gets a list of players in a Clan
- **MediusGetClanMemberList_ExtraInfo()** – Gets a list of players in a Clan

Clan Invitations Management

- **MediusInvitePlayerToClan()** – Request to invite player to Clan (by *AccountID*)
- **MediusInvitePlayerToClan_ByName()** – Request to invite player to Clan (by *Name*)
- **MediusCheckMyClanInvitations()** – Get list of Clan invitations
- **MediusGetClanInvitationsSent()** – Get list of Clan invitations sent
- **MediusRespondToClanInvitation()** – Respond to Clan invitation
- **MediusRevokeClanInvitation()** – Revoke Clan invitation

Clan Messages Management

- **MediusGetMyClanMessages()** – Get list of Clan messages for given player
- **MediusSendClanMessage()** – Request to send Clan message
- **MediusModifyClanMessage()** – Request to modify Clan message
- **MediusDeleteClanMessage()** – Request to delete Clan message
- **MediusGetAllClanMessages()** – Get list of all Clan messages

Clan Team Challenges Management

- **MediusRequestClanTeamChallenge()** – Request to perform a Clan team challenge
- **MediusRespondClanTeamChallenge()** – Respond to a Clan team challenge
- **MediusRevokeClanTeamChallenge()** – Revoke Clan team challenge
- **MediusConfirmClanTeamChallenge()** – Request to confirm Clan team challenge
- **MediusGetClanTeamChallengeHistory()** – Get list of all Clan team challenges ever requested
- **MediusGetClanTeamChallenges()** – Get list of current Clan team challenges

Clan Ladder Ranking System Management

- **MediusClanLadderList()** – Get list of Clan ladder ranks
- **MediusClanLadderPosition()** – Get current Clan ladder rank with given *ClanID*
- **MediusGetLadderStatsWide()** – Request to get current Clan ladder stats
- **MediusUpdateLadderStatsWide()** – Request to update Clan ladder stats
- **MediusUpdateClanStats()** – Request to update current Clan stats

Creating a Clan

When a player creates a Clan, they will automatically be designated as the 'Leader' of the newly created Clan. A player can only be the leader of one Clan at a time (i.e. create one Clan), but can have numerous Clan memberships with other Clans. Some applications are designed with the concept of each player having only one Clan membership at a time; nonetheless, this is not a requirement imposed by the Medius architecture.

1. Populate the **MediusCreateClanRequest** structure with the name of the Clan you wish to create.
2. Call **MediusCreateClan()**. The Medius platform will then attempt to create a Clan for this user. If successful, a response message with a generated *ClanID* will be returned.
3. If *StatusCode* != **MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the comment set):
 - a. **MediusAlreadyLeaderOfClan (-992)** – User can only create one Clan per *ApplicationID* (title).
 - b. **MediusClanNameInUse(-970)** – A Clan already exists with the given name.

Disbanding a Clan

Only the leader of a Clan can disband the Clan.

1. Populate the **MediusDisbandClanRequest** structure with the *ClanID* of the Clan you wish to disband.
2. Call **MediusDisbandClan()**. The Medius platform will then attempt to disband the given Clan. If successful, a response message with *StatusCode* set to **MediusSuccess** will be returned.
3. If *StatusCode* != **MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the comment set):
 - a. **MediusNotClanLeader (-990)** – User was not the leader of the Clan they requested to disband.

Getting a List of Clan Memberships

A player can only be the leader of one Clan at a time (i.e. create one Clan), but can have numerous Clan memberships with other Clans. The following sequence of events will return a player's current list of Clan memberships.

1. Populate the **MediusGetMyClansRequest** structure with a 'Start' number and a 'PageSize' number. Medius lists start with the number '1'. If you set the 'PageSize' to a value of (say) 10 and 'Start' to a value (say) 1, then you will be returned elements 1-10 (thus, setting 'Start' to 2 will return elements 11-20, etc.)
2. Call **MediusGetMyClans()**. The Medius platform will then return a **MediusGetMyClansResponse** message for each Clan the player is a member of. If successful, each response message will have the *StatusCode* set to **MediusSuccess**. Your application will need to be in a state where it is waiting for a response message with the 'EndOfList' field set to != 0 (thus signaling the end of the list).
3. If *StatusCode* != **MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the comment set):
 - a. **MediusNoResult (1)** – The user was not a member of any Clan.

Getting Clan Information

Given a *ClanID* or a Clan's Name, you can request more information about a Clan from the Medius platform. Information returned describes 'Clan Leader', 'Clan Stats', and 'Clan Status' information.

1. Given a *ClanID*, call **MediusGetClanByID()**. This will return a **MediusGetClanByIDResponse** message.
2. Given a Clan's Name, call **MediusGetClanByName()**. This will return a **MediusGetClanByNameResponse** message.
3. If *StatusCode* != **MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the comment set):
 - a. **MediusClanNotFound (-971)** – The Clan was not found.

Transferring Clan Leadership

The leader of a Clan can request to transfer Clan leadership to another player (if this player is not already a leader of a Clan). The old leader will still be a member of the Clan but will no longer be able to perform Clan functions that require Clan Leadership privileges.

1. Populate the **MediusTransferClanLeadershipRequest** structure with the *AccountID* of the player you wish to transfer Clan Leadership to. (**Note:** The *NewLeaderAccountName* field is ignored).
2. Call **MediusTransferClanLeadership()**. The Medius platform will then attempt to transfer the Clan Leadership status to the given user. If successful, a response message with *StatusCode* set to **MediusSuccess** will be returned.
3. If *StatusCode* != **MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the comment set):
 - a. **MediusNotClanLeader (-990)** – The user requesting to transfer Clan Leadership was not the current Clan Leader.

Adding Members to a Clan

The leader of a Clan can request (from the Medius platform) to add additional members to the Clan. The **MediusAddPlayerToClan()** call should be used with care because it will add a member to the given Clan without the members consent; therefore, it is recommended to use the **MediusInvitePlayerToClan()** (or **MediusInvitePlayerToClan_ByName()** (sender) and **MediusRespondToClanInvitation()** (receiver) calls to manage adding members to a Clan. Nonetheless, a developer may come across the need where **MediusAddPlayerToClan()** is needed (for example during testing or if a custom application level Clan invitation scheme has been devised).

1. Populate the **MediusAddPlayerToClanRequest** structure with the *AccountID* of the player you wish to add to the Clan along with a Welcome message (that will automatically trigger a Medius Clan message to be sent to the new member).
2. Call **MediusAddPlayerToClan()**. The Medius platform will then attempt to add the given user to the leader's Clan. If successful, a response message with *StatusCode* set to **MediusSuccess** will be returned.
3. If *StatusCode* != **MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the comment set):
 - a. **MediusNotClanLeader (-990)** – The user requesting to add a member to the Clan was not the current Clan Leader.

Removing Members from a Clan

The leader of a Clan can request (from the Medius platform) to remove a member from a Clan. No special messages will be sent to a member that was removed a Clan (this would need to be managed at the application level).

1. Populate the **MediusRemovePlayerFromClanRequest** structure with the *AccountID* of the player you wish to remove from the Clan.
2. Call **MediusRemovePlayerFromClan()**. The Medius platform will then attempt to remove the given member from the leader's Clan. If successful, a response message with *StatusCode* set to **MediusSuccess** will be returned.
3. If *StatusCode* != **MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the comment set):
 - a. **MediusNotClanLeader (-990)** – The user requesting to remove a member from the Clan was not the current Clan Leader.

Getting a List of Members in a Clan

The following sequence of events will return a list of members (Account Names) in a Clan. Each application must decide if they wish to allow anyone to view a Clan's membership list or if users are only able to view their own Clan's membership list (this is easily managed by *ClanID* utilization).

1. Populate the **MediusGetClanMemberListRequest** (or **MediusGetClanMemberListRequest_ExtraInfo**) structure with the *ClanID* of the Clan you wish to view Clan members with.
2. Call **MediusGetClanMemberList()** (or **MediusGetClanMemberList_ExtraInfo()**). The Medius platform will then return a **MediusGetClanMemberListResponse** (or **MediusGetClanMemberListResponse_ExtraInfo**) message for each Clan member. If successful, each response message will have the *StatusCode* set to **MediusSuccess**. Your application will need to be in a state where it is waiting for a response message with the '*EndOfList*' field set to != 0 (thus signaling the end of the list).
3. If *StatusCode* != **MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition. Possible error conditions (additional to the comment set):
 - a. **MediusNoResult (1)** – There were no members in the given Clan or the Clan does not exist.

Clan Invitations Management

The following group of functions facilitates 'Clan Invitations' management. Typically, the workflow is as follows:

1. Clan Leader issues a Clan invitation to a target user (**MediusInvitePlayerToClan()** uses a player's *AccountID* while **MediusInvitePlayerToClan()** uses a player's *AccountName*).
2. Target user retrieves their Clan invitations list.
3. Target user responds to the Clan invitation they wish to join (by responding with **MediusClanInvitationsResponseStatus** set to '**ClanInvitationAccept**').
4. The Medius platform will automatically internally call **MediusAddPlayerToClan()** for a user that responded with a '**ClanInvitationAccept**' status.

Additional Notes:

1. Calling **MediusInvitePlayerToClan()** will send a Clan invitation to target user account (only the Clan Leader can successfully carry out this transaction).
2. Calling **MediusGetClanInvitationsSent()** will return a list of Clan invitations sent (only the Clan Leader can retrieve this list). The list returned will only show Clan invitations that target users have not responded to.

3. Calling **MediusRevokeClanInvitation()** will revoke a Clan invitation sent (only the Clan Leader can successfully carry out this transaction). If a player has already accepted the Clan invitation, the response message will not be set with a **MediusSuccess** status).
4. Calling **MediusCheckMyClanInvitations()** will return a list of Clan invitations sent to 'me' calling user.
5. Calling **MediusRespondToClanInvitation()** allows a user to respond to a Clan invitation that was sent to them. Setting the *ResponseStatus* field to '**ClanInvitationAccept**' will automatically add them to the given Clan. After responding to a Clan invitation, the invitation will be removed from the target client's Clan invitation list. Currently, responding with a '**ClanInvitationDecline**' or '**ClanInvitationUndecided**' both have the same effect – removes the Clan invitation from both the sender and receiver and does not add player to the given Clan. Contact SCE-RT (scert-support@scea.com) if you need a Clan message to be issued to the Clan Leader if a player does not accept a Clan invitation.

Clan Messages Management

The following group of functions facilitates 'Clan Messages' management. Typically, the workflow is as follows:

1. Clan Leader issues a Clan message to their Clan.
2. Clan members request a list of all their 'unread' Clan messages. The Medius platform will then automatically set each Clan message flagged as 'read' on the Medius platform for each Clan message received. A player can still request to re-read Clan messages if needed with the 'GetAll' option.

Additional Notes:

1. Calling **MediusSendClanMessage()** will send a Clan message to all Clan members (only the Clan Leader can successfully carry out this transaction). These messages are stored in the Medius Database for each Clan member's account.
2. Calling **MediusGetMyClanMessages()** will return a list of all 'unread' Clan messages. Each Clan message received will be marked as 'read' in the Medius Database.
3. Calling **MediusGetAllClanMessages()** will return a list of all 'unread' and 'read' Clan messages. This will not flag Clan messages as 'read' in the Medius Database.
4. Calling **MediusDeleteClanMessage()** will delete a Clan message that was sent by the Clan Leader (only the Clan Leader can successfully carry out this transaction). This Clan message will no longer appear when a Clan member calls **MediusGetMyClanMessages()** or **MediusGetAllClanMessages()**.
5. Calling **MediusModifyClanMessage()** will modify a Clan message that was sent by the Clan Leader (only the Clan Leader can successfully carry out this transaction). Clan members will see the modified Clan message if the call either **MediusGetMyClanMessages()** or **MediusGetAllClanMessages()**.

Clan Team Challenges Management

The following group of functions facilitates 'Clan Team Challenges' management. Typically, the workflow is as follows:

1. Clan Leader issues a Clan Team Challenge to a target Clan.
2. Target Clan retrieves their Clan Team Challenges list.
3. Target Clan responds to the Clan Team Challenge.
4. Clan Leader (from step 1) retrieves their Clan Team Challenges list.
5. Clan Leader (from step 1) confirms Clan Team Challenge.

Confirming a Clan Team Challenge just sets the status as 'confirmed'. It is up to the application developer on how it wishes to manage 'confirmed' challenges. A custom message can be sent with the 'send' and the 'respond' Clan Team Challenge calls (this should include information on how the Clan Leaders will schedule when to play against each other.)

Design Note: Currently there is no 'Medius Mail System', if so the Clan Leaders could further send correspondence with each other when and how to schedule a time to play.

Additional Notes (only Clan Leaders may successfully carry out these transactions):

1. Calling **MediusRequestClanTeamChallenge()** will send a Clan Team Challenge to the target Clan.
2. Calling **MediusGetClanTeamChallenges()** and **MediusGetClanTeamChallengeHistory()** will return a list of Clan Team Challenges (both sent and received).
3. Calling **MediusRespondClanTeamChallenge()** will respond to a Clan Team Challenge.
4. Calling **MediusRevokeClanTeamChallenge()** will delete a Clan Team Challenge sent (therefore will no longer show up in a **MediusGetClanTeamChallenges()** call).
5. Calling **MediusConfirmClanTeamChallenge()** allows the Clan Leader that issued the Clan Team Challenge to 'confirm' an 'accepted' Clan Team Challenge.

Clan Ladder Management

To enable Clan ladders for your title, you must setup a Medius Database Caching Server (MDBCS). The primary purpose of the MDBCS is to unload CPU processing and database transactions from the other Medius platform servers to support Clan ladder management.

The following group of functions facilitates Clan ladder management. Typically, the workflow is as follows:

1. After a game is complete, the application calculates a new Clan rank (raw score) to be submitted to the Medius platform. Currently, there are from [0..LADDERSTATS_MAXLEN] available rankable Clan ladder rank categories to choose from (100 as of v1.50). Typically, the application would save an old Clan rank and increment or decrement the value to create a new Clan rank (possibly scaling the delta based on the other teams Clan rank).
2. Any Medius client would then be able to get a sorted list of Clans that indicates which Clan is first place all the way down to last place based on each rankable Clan ladder rank category.

Additional Notes:

1. Calling **MediusUpdateLadderStatsWide()** (with the **MediusLadderTypeClan** enumeration type set) will post a new Clan rank (raw score) to the Medius platform. It is recommended that all necessary rank categories [0..LADDERSTATS_MAXLEN] are set at once so that only one call to **MediusUpdateLadderStatsWide()** is needed. Each element of the array takes an int value. To set a 'score' for a Clan ladder category, just set a value to the respective element in the array (indexed by *LadderStatIndex*).

Note: Currently, only the Clan leader can successfully call this function to post Clan Ladder scores. If the Clan leader dropped out of the game for any reason; unfortunately, there is currently no way for the scores to be posted. Therefore, to ensure players finish the game, you may wish to decrement a Clans score by a few points, and the only way to regain the points is to actually complete a game.

2. Calling **MediusGetLadderStatsWide()** (with the **MediusLadderTypeClan** enumeration type set) will request for a Clan's current ladder stats information. Each clan has between [0..LADDERSTATSWIDE_MAXLEN] available Clan ladder categories to store Clan ladder rank information to.

Note: It is recommended that every player in a game call this function at the beginning of each round so that this information will be available (as noted above) just in case if a player (other than the Clan leader) has been declared as the player responsible for calling **MediusUpdateLadderStatsWide()**.

3. Calling **MediusUpdateClanStats()** updates the *Clan Stats* field. Even though Clan stats are not used directly in calculating Clan ladders, it can still store helpful information such as setting formulas and weighted values from which new Clan ranks can be calculated. Additional information such as 'Team Frags', last time guild played as a team, etc. Can be embedded in the Stats string for GUI display and or helping in calculating new Clan ranking raw scores. Call either **MediusGetClanByID()** or **MediusGetClanByName()** to gain access to Clan 'stats' information.
4. Calling **MediusClanLadderList()** will return an ascending list of Clans in a ladder system (first place to last place). Setting *ClanLadderStatIndex* between [0..LADDSTATS_MAXLEN] will determine which ladder category is returned.

Note: You can regulate the number of 'clan ladder ranks' returned to the requesting client (so that you can manage bandwidth usage and GUI lists) with the *StartPosition* and *PageSize* fields of the **MediusClanLadderListRequest** structure.

Example:

Set *PageSize* = 10 and *StartPosition* = 1 to get 'ranks' (1–10).

Set *PageSize* = 10 and *StartPosition* = 2 to get 'ranks' (11–20).

Set *PageSize* = 10 and *StartPosition* = 3 to get 'ranks' (21–30).

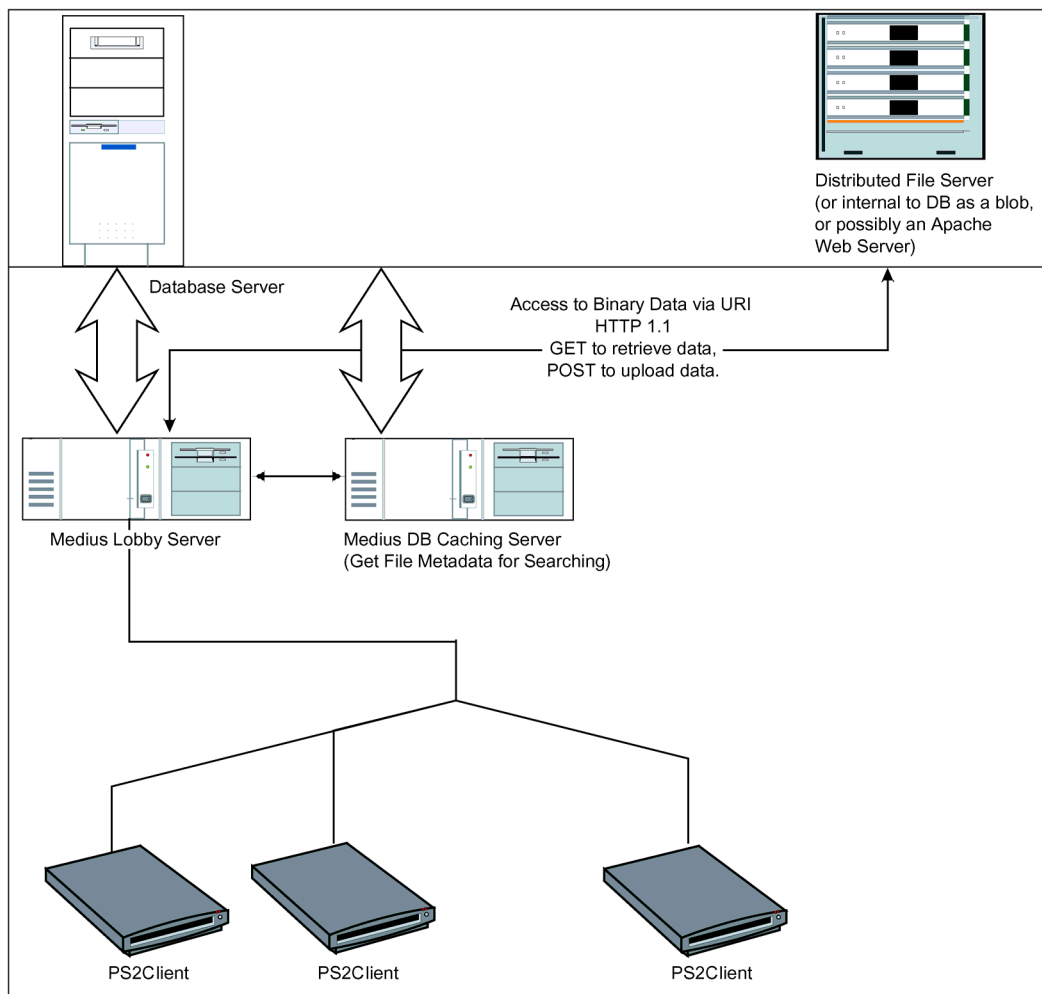
And so on...

5. Calling **MediusClanLadderPosition()** will return an exact position that a Clan is in a Ladder system. Setting *ClanLadderStatIndex* between [0..LADDERSTATS_MAXLEN] will determine which ladder category is returned.

Medius File Services (MFS)

As a plugin to the Medius platform, Medius File Services (MFS) is a SCE-RT technology that allows PS2 clients to store persistent game-specific data not already addressed by the Medius or SVO API. Files are created, uploaded, downloaded and destroyed against the user's authenticated account on the Medius servers. By default MFS is not enabled on Medius Servers – be sure to contact scert-support@scea.com to enable MFS for your title before attempting to work with the MFS API. As of the v1.50 release, the MFS API (MediusFileServices.h and MediusFileServicesTypes.h) is available in the standard SCE-RT client library release.

Figure 4: Overall Architecture of MFS



The Client Perspective

The general concept is that you have PlayStation®2 consoles out on the Internet running MFS-enabled software. The player will, at some point, want to save their game data or share some game-specific data online with other online players. They would create the file, upload the contents and receive confirmation that the file has been posted to the server. Clients can also request a list of available files, select one and download the contents based on Unix-like permissions.

By default the current release caps the FileQuota size a user has to 20 KB. This can be changed on a title-by-title basis. Games with a heavy online community are recommended to keep their file sizes down. It is also recommended that titles keep their file transfer transaction counts low. For example, downloading a file when a user logs in and then again when an online game has completed would be normal usage. It is not recommended to be used by any automatic retrieval methods, such as when a user joins a chat channel he/she will then download everyone's MFS game-specific data that are joined to the same chat room.

MFS Back End Overview

All file transfers, sent or received, are tunneled through the Medius Lobby Server. A high number of file requests or large transfers will incur a large hit on the MLS. This will affect all Lobby-related functionality in your title.

Medius File Services, on the server side, works with the Apache web server to manage your game's file repository through Common Gateway Interface (CGI) applications developed by SCE-RT. As clients post data to be saved in the repository or request files from the repository, HTTP is used by the back end to resolve these requests. This is a very scalable architecture, able to be load-balanced with standard off-the-shelf tools to grow to handle any level of demand. As your title's demand on the Medius File Services changes over time, the system can be modified to suit your needs and provide only the level of service required.

Specifically, when a client posts a file to the server, that file is received by the lobby server and transferred into the file repository transparently. Then, when other clients request the file, regardless of which lobby server is hosting their session, a request is made to the back-end Apache-driven repository to retrieve the file. Once retrieved from the back end, the file is then delivered down to the client through the lobby server.

A client begins its file download request by first opening the file. When a client wants to post/upload a file into the repository, it begins by attempting to create the file. Information about the newly created file is first stored into the database for faster lookups in the future. This also gives the server platform a chance to validate the file according to certain rules. Once the file is validated and permissions are verified, the file is actually uploaded into the repository via the Apache-driven back-end.

Each user is given a file quota (20 KB by default), or a maximum amount of space they are allowed to occupy within the repository. This quota information, much like information about files themselves, is stored in the database and associated with the user's online account. When attempting to create any file in the repository, the client's account quota is checked to ensure they have enough free space. Quotas are a virtual concept and are configurable by the system administrator to help balance the consumption of resources within the file repository.

Medius File Services API

Initialization

Medius File Services does not require any additional initialization steps aside from the normal **NetInitialize()** and **MediusInitializeBare()** or **MediusInitialize()**.

Uploading a File

To upload a file using MFS you will need to issue **MediusFileCreate()** to start creation of the file. It is very important to prohibit the player from entering a forward slash or backslash in the Filename. This will create a directory on the file repository. However, you (the programmer) may use directories. Do not allow the player to create directories.

MediusUploadFile() starts an upload. Once an upload has begun, the **UploadCallback** will begin to fire. One of the members to the callback struct by UploadCallback is **iXferStatus**. When **MediusFileUploadResponse.iXferStatus=MEDIUS_FILE_XFER_STATUS_END** the upload has completed and you may now issue the command to close the file. You also have the opportunity to cancel the upload before **MEDIUS_FILE_XFER_STATUS_END** has been reached.

To tell Medius to close a file you must call **MediusCloseFile()**. It is only after **MediusCloseFile()** that the file will then exist on the file repository and show up with **MediusFileListFiles()**.

Downloading a File

To download a file you will need to call **MediusFileDownload()**. The callback to this function will keep firing until the download has completed (**MediusFileDownloadReposnse.iXferStatus** = **MEDIUS_FILE_XFER_STATUS_END**). **Note:** On very small downloads this callback will only be called once and the **iXferStatus** will never equal **MEDIUS_FILE_XFER_STATUS_INITIAL**.

Canceling a Upload or Download

If a file is currently being downloaded or uploaded, a call to **MediusFileCancelOperation()** will cancel the transaction and the Upload and Download callbacks will cease to fire.

Listing Files

To retrieve a listing of files that have been previously uploaded and have not yet been deleted, you can call **MediusFileListFiles()**. **MediusFileListFiles()** will allow you to filter the listing of files based on different criteria such as file names or file size. This can be found in the **MediusFileListRequest** struct.

Deleting a File

Deletion of a file is performed by calling **MediusFileDelete()** and filling out the appropriate **MediusFileDeleteRequest** struct. It is important to note that no file is ever deleted in the file repository. It is however marked as deleted in the database and will not show up when using **MediusFileListFiles()**. If a new file is created with the same name as a previously deleted file, the previously deleted file will then be overwritten with the new file data.

MediusFileServices Sample

Please refer to the “MediusFileServices Sample” for more details on how to perform the above tasks.

Vulgarity Filtering

The SCE-RT SDK provides the ability to filter content for vulgarity. All text-based content sent to the Medius platform infrastructure can be filtered for vulgarity based on a server-side dictionary. The software performs both hard matches and soft matches for a wide variety of words, in whatever language the server is set up with.

Filtering is enabled for all SCEA, SCEI and SCEE first party titles. If you would like to enable the filtering capabilities for a third party title, please contact developer support at scert-support@scea.com for further help.

Vulgarity Filtering Process

There is only one API function used for vulgarity filtering, **MediusTextFilter()**. This function will ask the Medius platform if the "Text" field of the **MediusTextFilterRequest** message should "pass/fail" or "search-and-replace" due to server-side vulgarity checks.

Vulgarity checks should normally be placed on the following elements (list may change):

1. "AccountName" submitted during **MediusAccountRegistration()** – (use "pass/fail").
2. "SessionDisplayName" submitted during **MediusAnonymousLogin()** – (use "pass/fail").
3. "Message" submitted during **MediusSendGenericChatMessage()** – (use either "pass/fail" or "search-and-replace").
4. "GameName" submitted during **MediusCreateGame()** – (use "pass/fail").
5. "LobbyName" submitted during **MediusCreateChannel()** – (use "pass/fail").
6. "ClanName" submitted during **MediusCreateClan()** – (use "pass/fail").
7. If your application supports in-game chat directly with the DME (via **NetSendMessage()** and/or **NetSendApplicationMessage()**), since your Medius connection will still be open, you may consider (depending on the "Audience Rating" of your game) if you want to call **MediusTextFilter()** before you send in-game chat messages (of course LAN games will not have a Medius connection established; therefore, take this into account as you code this up) – (use either "pass/fail" or "search-and-replace").

Before you execute any of the functions in the list above, call **MediusTextFilter()** to verify that the given text string is appropriate. You may need to experiment with a mini 'state-machine' and/or 'message queue' to get 'non-blocking' behavior to occur while a particular vulgarity check is taking place so as to only call the given function if a successful vulgarity check occurred and/or a "search-and-replace" operation modified the submitted text string.

Typically, if doing a "pass/fail", if a "fail" was received, you should inform the user that the "text" they submitted did not pass the vulgarity check, and ask them to re-enter information. Hopefully our noble and distinguished user will not try to 'outsmart' the server-side vulgarity checks, but if they do, personnel dedicated to monitoring SCE-RT server activity can take correct measures.

AntiCheat

Overview

AntiCheat in general is a very broad topic encompassing everything from client-side obfuscation to router Access Control Lists (ACLs). SCE-RT would like to encourage developers to contact us (scert-support@scea.com) regarding client-side defensive programming techniques. Be advised that such things are best adopted rather early in the development cycle. This document, however, covers the AntiCheat service offered by the Medius Lobby Server (MLS).

The MLS uses plug-ins to determine title-specific logic for AntiCheat. These plug-ins are called "Detectives". While it is possible for a developer to create a totally custom Detective for its title-specific logic, SCE-RT recommends using the generic Detective which can be configured on a per-title basis. (Programming a Detective occurs at the C++ level, but reconfiguring the generic Detective takes place in an XML configuration file owned by SCE-RT's production staff.) Using the generic Detective allows developers to gain significant AntiCheat benefit without spending much effort on development.

Note: In order for AntiCheat to work, the title must have issued a MediusDnasSignature. This is a TRC requirement for all online titles.

Henceforth, this document discusses decisions the developer may need to make in order to use the generic Detective correctly for a given title.

Trigger Events

The generic Detective issues a series of challenges against each client. Once a particular client has passed all these challenges, the client will not be challenged again. The series of challenges is initiated when a client (which has not yet passed) makes an API call corresponding to one of the events below.

Using the generic Detective, there are nine trigger events that can be used as a Trigger Event. These correspond (roughly) to Medius API calls:

- LobbyConnect
- JoinGame
- LeaveGame
- PlayerReport
- StatsReport
- ChatMessage
- GetChannelList
- GetGameList
- GetMyClans

One or more of these trigger events can be chosen for a given title. For example, the Detective could be configured with both **LobbyConnect** and **GetGameList**. Normally, we assume that a title will not want AntiCheat challenges to begin until after the user has clicked through their End User License Agreement (EULA). This is the primary consideration in choosing your trigger events. Other considerations might include the way the title uses the lobby servers. (For instance, if they rapidly change lobbies before settling down on one for a game, they might not want to use the **LobbyConnect** event.) Ideally, the player will have acknowledged his EULA before ever connecting to the MLS (on the MAS, for instance). All of these decisions should be coordinated with SCE-RT support.

Cheat Challenges

The challenges can be of various types and include: BinaryChallenge, BinaryIopChallenge, MatchRawMemory, and ThreadCount. These are usually used to make certain that each player is playing on code that is approved by the developer. Many cheat devices function by modifying the game's instructions in memory.

SCE-RT already has images of many popular cheat devices. These can be seamlessly “dropped in” to any title's MLS configuration. This is by far the simplest way for a title to participate in AntiCheat. The next step involves creating what is known as a “Master Integrity Image” for your title. This is basically a copy of the instructions used by the title. These are obviously title-specific and it is probably best to coordinate the creation of this image with SCE-RT support.

Cheat Policies

Cheat policies are rules that are applied to the player after a detection (or number of detections) have occurred. In addition to detection events, cheat policies are also applied upon account login after a ban or suspension is instated. Once a detection has occurred, the incident is recorded and the client is “kicked” by the server. A lookup of applicable Cheat Policies occurs for the detected client's *ApplicationID*. There are currently three types of cheat policies:

1. DNAS Signature-based ban
2. *AccountID* suspension
3. Simple detection and kick from server

Any or a combination of the cheat policies can be applied to detection events. For example, a title may want the following policies applied for cheat detections:

- First detection, no suspension applied. This is considered a warning.
- Second detection, one-hour suspension is applied to the DNAS_Signature that was detected.
- Third and fourth detection, two-hour suspension is applied to the DNAS_Signature that was detected.
- Fifth and sixth detection, one-day suspension is applied to DNAS_Signature that was detected.
- Seventh suspension and on, permanent suspension of *AccountID*.

Cheat Messages

Cheat Messages are messages that are sent to the player detected by AntiCheat. They may also be sent upon Account Login when the client has a time-based ban applied. Cheat Messages are configured in the database. For a particular cheat policy there may exist a corresponding cheat message. Cheat Messages are localized based on language ID. For a given title, there may exist a number of cheat messages for each language ID that the title supports per cheat policy. There may only be one cheat message per cheatpolicyid/languageid pair.

Examples of MsgText in a Cheat Message

“You have been detected cheating or using a cheat device CHEAT_TIMES. Your Player Name has been terminated. For more information please call SCEA Consumer Services at 1-800-345-7669.”

“You have been detected cheating or using a cheat device CHEAT_TIMES. You have been banned from accessing APPLICATIONNAME for BANNED_TIME and your Player Name has been terminated. You will be able to access APPLICATIONNAME again in PLAY_AGAIN. For more information please call SCEA Consumer Services at 1-800-345-7669.”

Some variables that Cheat Messages may use:

CHEAT_TIMES – Number of cheat detection events recorded for a particular *AccountID*.

MAX_CHEATS – Value of maxcheatcount for a particular *ApplicationID*.

PLAY_AGAIN – Remaining ban time for DNAS time-based ban.

ACCOUNTNAME – *AccountName* string for a particular *AccountID*.

APPLICATIONNAME – Game name for a particular *ApplicationID*.

BANNED_TIME – Length of ban time for DNAS time-based ban.

What AntiCheat Can and Cannot Do

Broadly, what AntiCheat can do is examine the memory and/or state of the PlayStation®2 itself.

AntiCheat cannot examine traffic between clients at the DME layer since it is an MLS-based service. (So, it has no hope of detecting modern lag devices.) Also, please bear in mind that AntiCheat is not foolproof, and indeed cannot be made so. AntiCheat's purpose is to make it difficult for the average user to cheat.

Internationalization (Multi-Byte Language Support)

The SCE-RT SDK supports multi-byte character sets. Applications that need to perform data storage and retrieval with multi-byte character sets will be able to do so by using UTF-8 encoding (Unicode) by reading and writing appropriately into the single byte character related API data structures. Language types currently supported: English, Japanese, Korean, Italian, Spanish, German, and French. If you wish to have additional *MediusLanguageTypes* defined, contact SCE-RT developer support (scert-support@scea.com).

Note: This will limit the amount of character data to what would normally be allowed by a single byte character set. For example, “*AccountName*” is of size `ACCOUNTNAME_MAXLEN` (32). Using Extended ASCII you can have a character string of up to 32 characters. However, if you are using Unicode, each Unicode character may need 2-3 bytes for each UTF-8 character (depending on the particular character being used). In this case data is still limited to a 32 byte string.

The Medius function that provides internationalization is:

- **MediusSetLocalizationParams()** – Sets internationalization parameters

Example of Internationalization

The Medius API function used to support internationalization is **MediusSetLocalizationParams()**. If you do not call this, by default “US English” is set for *MediusLanguageType* (`MediusLanguage_USEnglish`) with “Extended ASCII” set as the default *MediusCharacterEncoding* type (`MediusCharacterEncoding_ISO8859_1`). After calling this function, the ‘player object’ residing on the Medius platform will be set with the specified localization parameters. The general process is as follows:

1. Call **MediusConnect()** to connect to the MAS.
2. Start a “Session” with the Medius platform.
3. Call **MediusSetLocalizationParams()** to set our language and encoding type.
4. Call **MediusGetPolicy()** to get the specific application’s ‘usage policy’ and ‘privacy policy’. Based on the *MediusLanguageType* set for the current user, there will be a policy entry in the Medius database for that language type to be returned. If no ‘usage policy’ or ‘privacy policy’ has been set up, an empty string will be returned.
5. Call **MediusGetAnnouncements()** or **MediusGetAllAnnouncements()** to retrieve the current announcements for your application. Based on the *MediusLanguageType* set for the current user, there will be various ‘announcement’ entries in the Medius database for that language type to be returned. If no ‘announcements’ have been set up, an empty string will be returned.
6. Proceed with “establishing an authenticated state” (Account Login).

As you can see, the Medius platform can support many language types for a given application. Even though the application may not properly display character string data (due to different Unicode character sets) as seen with (for example) player names and chat messages, users with different languages can all still play games together. It is the application developer’s responsibility to convert character strings for the given language into the respective encoding type. For more information on how to convert your particular language to UTF-8, search the Internet with this keyword (**+“UTF-8” conversion library**).

If you wish to have your application designed so that players with a particular language only see players with the same language, then you will either need to set up additional Medius servers (with a different URL or IP combinations) or set up Co-Location parameters to your Medius platform (see the “Co-location” section for more information).

The Process for Setting Internationalization Parameters:

1. Create an instance of the **MediusSetLocalizationParamsRequest** structure.
 - a. **memset()** the structure to '0'.
 - b. Set *MessageID* to a value of your numbering scheme (if needed).
 - c. Set *MessageType* to a *MediusMessageType*.
 - d. Set *Language* to a *MediusLanguageType*.
2. Call **MediusSetLocalizationParams()** with the respective request message and callback.
3. When the **MediusStatusResponse** message is received, if *StatusCode*=**MediusSuccess** then your language type and encoding type will be set.
If *StatusCode*!=**MediusSuccess**, be sure to transition through your lobby state-machine to handle the current error condition.

Medius Universe Information Server (MUIS)

The SCE-RT SDK supports connection to a (MUIS) Medius Universe Information Server. The MUIS provides the client with a list of Universes available for a particular application. An application may wish to have Universes (Medius platform server sets) spread out geographically around the world, configured with different languages, and strategically deployed closer to locations with a high user base so as to provide users with lower ping times in effect providing an optimal playing experience due to lower latencies; nonetheless, still providing users the option to play against users around the world. This is a step up from co-location – which at any one time has only one MUM managing an entire Medius platform server set. As expected, a user would need to create a new account on each Universe they connect to.

For example, if you have four Medius platform server sets for an application operational and deployed in: San Diego, England, Korea, and Japan (each server set includes 1-MUM, n-MAS, n-MLS, n-MPS, 1-MCS and (n-GS)), a single MUIS will read a list of Universes from a configuration file (which is re-read after a set time, so more Universes can be added later) then will establish a connection to each of the Universe's MUM to obtain current information regarding those Universes. After a client connects to the MUIS and retrieves a list of Universes, they should do a DNS lookup of the selected Universe's DNS name and port number, and then continue with a **MediusConnect()** to the Universe's MAS (The Universe's DNS name and port number will be mapped to a MAS for that server set).

The Medius functions that provides Medius Universe Information from a MUIS are:

- MediusGetUniverseInformation()** – Retrieves Medius Universe Information from a MUIS
- MediusUpdateUniverseInformation()** – To be called each frame instead of MediusUpdate()

The MUIS (as of v2.9) will return the following in a list (with multilingual support):

- Universe Name (and ID)
- Fully qualified (DNS) Domain Name
- Port number
- A Universe description
- Status (Available/Unavailable)
- Current User Count as seen by the Universe's MUM
- Universe's Max User count
- (Billing information if enabled)
- Extended Info customizable per title. Usually used for patching by storing patch version and patch URL.
- SVO (SCE-RT View Online) start URL

Connecting to the MUIS

Connecting to a MUIS is just like connecting to a MAS for the first time (in calling **MediusSetDefaultConnectInParams()** and enabling security) The general process is as follows:

1. Call **MediusConnect()** to connect to the MUIS.
2. Call **MediusUpdateUniverseInformation()** (instead of **MediusUpdate()**) each frame to process incoming/outgoing messages for the MUIS.
3. Call **MediusGetUniverseInformation()** to retrieve a list of Universes and Universe News

Be sure to set/mask InfoType (of **MediusGetUniverseInformationRequest**) with the following standard set of **MediusUniverseInformationType** values:

INFO_NEWS | INFO_ID | INFO_NAME | INFO_DNS | INFO_DESCRIPTION | INFO_STATUS

4. User chooses which Universe to connect to (by selecting a returned DNS string and port combination from one of the response message returned from the **MediusGetUniverseInformation()** call).
5. When the *EndOfList* flag is set to true during a **MediusUniverseVariableInformationResponse** message, **MediusUpdateUniverseInformation()** will automatically disconnect from the MUIS with an internal call to **MediusDisconnect()**. It is safe to still call **MediusUpdateUniverseInformation()** each frame, but is recommended to not do so (unlike **MediusUpdate()** which would return 'Not Connected' error return codes –this is why the MUIS has it's own 'update' call).
6. Perform a DNS lookup with the selected DNS string.
7. Call **MediusConnect()** with the IP returned from the DNS lookup along with the port number to connect to the MAS of the chosen Universe. At this point we should be calling **MediusUpdate()** each frame instead of **MediusUpdateUniverseInformation()**.
8. Proceed with “establishing an authenticated state” (Account Login).

The Process for Requesting Universe Information:

1. Create an instance of the **MediusGetUniverseInformationRequest** structure.
 - a. **memset()** the structure to '0'.
 - b. Set *MessageID* to a value of your numbering scheme (if needed).
 - c. Set *InfoType* to each (masked value of) *MediusUniverseInformationType*. (*INFO_NEWS* | *INFO_ID* | *INFO_NAME* | *INFO_DNS* | *INFO_DESCRIPTION* | *INFO_STATUS*) are recommended.
 - d. Set *CharacterEncoding* to a *MediusCharacterEncodingType*. (**MediusCharacterEncoding_ISO8859_1** set by default).
 - e. Set *Language* to a *MediusLanguageType* (**MediusLanguage_USEnglish** set by default).
2. Call **MediusGetUniverseInformation()** with the respective request message and callback.
3. If *InfoType* of **MediusGetUniverseInformationRequest** was set with *INFO_NEWS*, a number of News (chunks of data) messages will be returned. *EndOfText* set to != '0' will indicate all News data has been sent.
4. For each Medius server set (i.e. Universe) a response message will be sent indicating a DNS name and port of a MAS to connect to. *EndOfList* set to != '0' indicates that all elements (response messages) of the list have been sent.
5. User should then select a Universe, call **MediusConnect()** to the Universe's MAS (i.e. with Universe's DNS name and port number after a DNS lookup returned a IP address for the DNS name).

Running the Medius Servers

The Medius Lobby Server, Medius Authentication Server and Medius Universe Manager are currently distributed as Redhat Linux 9 (glibc-2.3.2-27.9.7) and Win32 products. The Linux builds have not been tested on any other Linux distributions.

The following instructions detail how to run the database-disabled versions of the Medius Lobby Server and Medius Authentication Server. Please contact us (scert-support@scea.com) for details if you wish to set up and host your own development database server. Enabling and disabling database, as well as other functionalities, is performed through appropriate configuration files. More details on these configuration files can be found in the associated “readme” file in this release.

Operating the Medius Server Executables

1. Copy the desired Medius Universe Manager (MUM) executable file and appropriate configuration file (included within this release) to a folder on the machine that will act as the Medius Universe Manager.
2. Copy the desired Medius Lobby Server (MLS) executable file and appropriate configuration file (included within this release) to a folder on the machine that will act as the Medius Lobby Server.
3. Copy the desired Medius Authentication Server (MAS) executable file and appropriate configuration file (included within this release) to a folder on the machine that will act as the Medius Authentication Server.
4. Copy the desired Medius Proxy Server (MPS) executable file and appropriate configuration file (included within this release) to a folder on the machine that will act as the Medius Proxy Server.
5. (If Client/Server) Copy the desired DME Game Server (GS) executable file and appropriate configuration file (included within the DME release) to a folder on the machine that will act as the DME Game Server. This can be the same server as the Medius Lobby Server, Medius Authentication Server or Medius Universe Manager; this will drastically affect the performance of the servers, and is only suitable for basic functionality development.
6. Modify the Medius Lobby Server and Medius Authentication Server configuration files. Ensure that the DefaultConfigLobbyPassword value is set to nothing (empty string), unless you wish to enforce a password on all lobby channels. Here you will also find the port that the server will be running on, as well as the IP/Port that the Medius Lobby Server or Medius Authentication Server should use to connect to the Medius Universe Manager.
7. Modify the DME Server configuration file. MediusReportIP and Port should be set to the IP and Port address of the machine hosting the Medius Authentication Server. ReportInterval should be set, in milliseconds, to the rate that the server should send Server Reports to the MLS (this is recommended to be 30000). Finally, if running both servers on the same machine, ensure that this is set to a different port.
8. Modify the Medius Universe Manager configuration file. The Medius Universe Manager can be configured to run on any port.
9. Run the Medius Universe Manager executable file. Wait approximately 10 seconds for the server to initialize.
10. Run the Medius Lobby Server executable file. Wait approximately 10 seconds for the server to initialize and establish a connection with the Medius Universe Manager specified in the configuration file.
11. Run the Medius Authentication Server executable file. Wait approximately 10 seconds for the server to initialize and establish a connection with the Medius Universe Manager specified in the configuration file.
12. Run the DME Game Server executable file. Wait approximately 10 seconds for the server to initialize and begin sending reports to the Medius Authentication Server.

13. Begin making connections and sending requests to the Medius Authentication Server and Medius Lobby Server. Depending on the type of request, certain backend requests and data will flow to the other servers to service each request.

This page intentionally left blank.