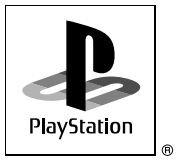


SCE-RT SDK Distributed Memory Engine (DME) API Release 2.10

Overview

March 2005

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.

Table of Contents

This page intentionally left blank.

About This Manual

This is the *SCE-RT SDK Distributed Memory Engine (DME) API Release 2.10 – Overview*.

The SCE-RT Distributed Memory Engine (DME) builds upon the RTIME Interactive Networking Engine. This document provides an overview of the SCE-RT DME, including examples.

The DME is an interface to the SCE-RT networking engine. It simplifies the development of multi-user interactive games and applications. Games can be developed in either peer-to-peer or client/server architectures.

Changes Since Last Release

SCE-RT 2.10 DME now adds support for the PlayStation Portable (PSP™). Many of the structs previously defined in dme.h have been moved to rt_nettypes.h.

Deprecated APIs

The following API's are being deprecated, no new software should use these functions as they will be removed in a future release:

NetSendApplicationMessage

NetSendMessage

Removed APIs

The following API's have been removed:

NetSendFilteredMessage

NetTypeMessageFilterCallBack

NetSendMessageByUDPtoIP

Changed APIs

ObjectFiltering

NetConnect (no more blocking calls)

NetAddressType

NetAddress

NetConnectStatus

NetErrorCode

NetMessageClass

NetConnectInParams

NetGameFind (now NetLANGameFind)

NetStreamMedia

New APIs

NetTokens

NetLANFind (replaces NetGameFind)

NetLANMessaging

NetBitmask

NetClientList

NetGetLatencyMetrics

NetTokens vs. NetObjects vs. NetStreamMedia vs. NetMessages

NetDisconnectInParams

NetSendAppMessage
NetRegisterObjectFilter

Typographic Conventions

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

Convention	Meaning
`courier`	Indicates literal program code.
italic	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)

Overview

The SCE-RT Distributed Memory Engine (DME) builds upon the RTIME Interactive Networking Engine. This document provides an overview of the SCE-RT DME, including examples.

The DME is an interface to the SCE-RT networking engine. It simplifies the development of multi-user interactive games and applications. Games can be developed in either peer-to-peer or client/server architectures.

The DME provides the following capabilities:

- Session Master – The DME provides the ability to have a client who can authoritatively control a game
- Session Master Migration – Games can continue even after the creator has left the game
- Client management
- Distributed object synchronization
- Bandwidth-controlled data stream broadcasting
- Each application can declare its own object and message structures in a simple manner through a series of registration functions
- Automated delta update broadcasting occurs immediately after creation of an object
- Developers can define filter functions for limiting the propagation of objects, tokens, DataStreams and messages
- Application callbacks are provided for remote client updates, object updates, messages and data stream updates
- Applications can utilize Stream Media support for video and voice-over IP functionality
- Applications can define various transport options for game-specific objects, tokens, and messages

PSP™ Overview

The 2.10 release supports PSP™ network development. SCE-RT PSP™ supports both ad hoc and infrastructure modes of communication. SCE-RT PSP™ enables developers to easily port existing PS2 SCE-RT applications to the PSP™.

Adhoc – wireless peer-to-peer only communication. PSP™ Medius connection not possible.

Infrastructure – wireless network infrastructure support. PSP™ Medius connection possible.

PSP™ Platform Requirements

- PSP™ Development kit
- SCE 1.0.3 Library/tool chain

PSP™ Network Stack Initialization

Similar to other DME platforms, it is up to the application programmer to initialize the appropriate PSP™ networking stack. For example if Adhoc is to be used the application programmer must initialize the SCEI Adhoc stack before using the DME. For more information on the SCEI stacks please consult the SCEI library documentation. Be sure to follow the documentation for TRC Compliance.

Linking the DME Libraries into your PS2 or PSP™ Application

The SCE-RT library components are available as static libraries and ERXs. Using the static libraries is as simple as building the application with the appropriate static libraries, and calling the appropriate NetUse functions.

NOTICE: It is critical to call the appropriate NetUse functions when initializing the SCE-RT libraries. Not doing so can result in undefined behavior (most often FATAL Errors). It is required that you call the NetUse functions before calling NetIntialize(). (see Table 1)

Applications can also use the ERX versions of the components. An ERX is a dynamic module created using Sony's ERX tools. An explanation of the ERX system is beyond the scope of this document. Developers interested in using the ERX components should read the ERX section in the EE Kernel Overview document.

For specific examples of the how the libraries are linked to together to produce an executable, please see the sample applications in the SCE-RT SDK release.

The DME components are:

- **librtbase:** Contains the core functionality of the DME. Every application must use this component.
- For the communication layer of the DME, every application must use one of these components.
  - **librtcomm** – inet comm library for PS2
  - **librtcommee** – libeenet comm library for PS2
  - **librtcommah** – Adhoc comm library for PSP™
  - **librtcommin** – Infrastructure/inet comm for PSP™

Note: For PSP™ it is okay to load both the **librtcommah** and **librtcommin** libraries.

- **librtinetb:** Contains functionality for use with the SCEI Inet network stack. **librtcomm** must be used with this component.
- **librtinetc:** Contains a more memory efficient Inet network stack as well as being more CPU efficient on the EE. **librtcomm** must be used with this component. If your application is limited in either IOP or EE memory or CPU usage, **librtinetc** can be used instead of **librtinetb**.
- **librtmsgcl:** Contains Client Server functionality. If your application uses TCP/IP to connect to a Medius or DME Server your application must use this component.
- **librtmsgsr:** Contains the functionality required for utilizing the integrated server component.
- **librtp2p:** Contains peer-to-peer functionality.
- **librtobject:** Contains distributed object functionality. Most applications will need to use **librtobject**.
- **librtcrypt:** Contains encryption functionality.
- **librtmedia:** Contains functionality for streaming audio/video. If your application uses audio or video streams this component must be used.
- **librtlpc:** Contains information for streaming audio using the LPC Codec. **librtmedia** must also be used.
- **librtlpc10:** Provides the LPC10 audio codec. **librtmedia** must also be used.
- **librtGSM:** Contains information for streaming audio using the GSM Codec. **librtmedia** must also be used.

Medius is made up of the following two components:

- **libmcl:** Contains Medius client functionality.
- **libmgcl:** Contains MGCL (Medius Game Communication Library) functionality.

NetUse Functions

There are approximately a dozen “NetUse” functions in the DME. These functions allow the application developer to choose which components are to be included in the application. Components cannot be used unless their NetUse function is called. The NetUse functions are defined as follows:

Table 1

Platform	DME Library	Required NetUse* Function
PS2, PSP™	**librtbase**	**NetUseDme()**
PS2, PSP™	**librtcomm**	**NetUseCommInet()**
PS2, PSP™	**librtcommee**	**NetUseCommEEnet()**
PSP™ only	**librtcommah**	**NetUseCommAdhoc()***
PSP™ only	**librtcommin**	**NetUseCommInet()***
PS2 only	**librtinetb**	None
PS2 only	**librtinetc**	None
PS2, PSP™	**librtmsgcl**	**NetUseClientServer()**
PS2, PSP™	**librtmsgsr**	None
PS2, PSP™	**librtp2p**	**NetUsePeerToPeer()**
PS2, PSP™	**librtobject**	**NetUseObjects()**
PS2, PSP™	**librtcrypt**	**NetUseCrypt()**
PS2, PSP™	**librtmedia**	**NetUseStreamMedia()**
PS2, PSP™	**librtlpc**	**NetUseACodecLPC()**
PS2, PSP™	**librtlpc10**	**NetUseACodecLPC10()**
PS2, PSP™	**librtGSM**	**NetUseACodecGSM()**

* There are two NetUse Functions specific to the PSP™. **NetUseCommAdhoc** is for enabling Adhoc communication, while **NetUseCommInet** is for Infrastructure communication. Applications can toggle between Adhoc and Infrastructure modes with these two calls.

For example, a peer-to-peer PS2 LAN application that uses objects without encryption could be initialized as follows:

```
NetUseDme();
NetUseClientServer();
NetUsePeerToPeer();
NetUseCrypt();
NetUseObjects();
NetUseCommInet();
NetSetDefaultInitializeParams(&InitInParams);
InitInParams.ApplicationID=KM_GetSoftwareID();
NetInitialize(&InitInParams, &InitOutParams);
```

And link statically the following components:

```
librtbasePS2.a,
librtp2pPS2.a,
librtobjectPS2.a
librtcommPS2.a
```

or dynamically link:

```
rtbase.erx
rtp2p.erx
rtobject.erx
rtcomm.erx
```

Any application that uses Medius would make the following calls:

```
stMediusInitInParams.ApplicationID = KM_GetSoftwareID();
NetUseDme();
NetUseObjects();
NetUsePeerToPeer();
NetUseClientServer();
MediusInitialize(&stMediusInitInParams,&stMediusInitOutParams);
```

And statically link the following components:

```
librtbasePS2.a,
librtp2pPS2.a,
librtobjectPS2.a,
librtmsgclPS2.a
librtcommPS2.a
```

or dynamically link:

```
rtbase.erx
rtp2p.erx
rtobject.erx
rtmsgcl.erx
rtcomm.erx
```

An example of using encryption with the DME follows:

```
RSA_KEYPAIR AppKeyPair;
RSA_KEYPAIR LocalKeyPair;

// Struct declarations for NetUse() functions
NetUsePeerToPeerInParams UsePeerToPeer;
NetUseObjectsInParams UseObjects;

NetSetDefaultInitializeParams(&InitInParams);

KM_GetSoftwareKeyPair(&AppKeyPair.publicKey,&AppKeyPair.privateKey);
KM_GenerateRSAKeyPair(&LocalKeyPair.publicKey,&LocalKeyPair.privateKey);
InitInParams.pApplicationKeyPair = &AppKeyPair;
InitInParams.pLocalKeyPair = &LocalKeyPair;
InitInParams.ApplicationID=KM_GetSoftwareID();

// Setup the necessary structs before calling NetUse
NetSetDefaultNetUsePeerToPeerInParams(&UsePeerToPeer);
NetSetDefaultNetUseObjectsInParams(&UseObjects);

// Call NetUse functions
NetUseDme();
NetUseObjects(&UseObjects);
NetUsePeerToPeer(&UsePeerToPeer);
NetUseClientServer();
NetUseCrypt();

NetInitialize(InitInParams,InitOutParams);
```

And statically link the following components:

```
librtbasePS2.a,
librtp2pPS2.a,
librtobjectPS2.a,
librtmsgclPS2.a
librtcryptPS2.a
librtcommPS2.a
```

or dynamically link:

```
rtbase.erx
rtp2p.erx
rtobject.erx
rtmsgcl.erx
rtcrypt.erx
rtcomm.erx
```

The DME Application Interface

The application interface provides a set of functions and callback mechanisms to enable an application to synchronize the World State of clients that are connected to an SCE-RT Server.

A typical application will call the client interface to:

- Initialize the DME
- Perform registration
- Connect to the Server
- Join a world
- Create objects and DataStreams
- Send messages
- Call **NetUpdate()** every frame

When the DME detects new clients, it updates remote objects or DataStreams, and notifies the application by calling the functions specified in the registration. The DME also allows the application to query the contents of objects, DataStreams, clients and other state variables by using the access functions.

The Session Master

The Session Master is a client with special properties. It performs executive functions for managing clients.

The Session Master can also provide application-specific functionality such as:

- Score keeping
- End-of-game detection
- Filtering of messages and objects

The Client and Session Master Application

These applications should have code for registration, connection, object creation/deletion and disconnection. The application should implement callback functions for receiving updates and messages.

Note: the registration code must be identical for both the client and the Session Master. In most cases the client application will be identical to the Session Master. This is required if the Session Master is determined at connection time, when it is shared.

The DME Server

The DME Server is the central hub for all game-related Internet communications. A client must connect to the Server before it can talk to other clients. The DME Server is inactive when the application is using the integrated server or PeerToPeer hosting. In both integrated Server and PeerToPeer hosting, only one world is supported. Each DME Server allows multiple concurrent applications to have separate communication channels (specified by the *WorldID*). In a non-PeerToPeer environment, all messages are sent first to the Server, then either sent on to the Session Master or passed on to the target clients.

Note: Throughout the remainder of this document, the term *Server* can be generically applied to a Host of a PeerToPeer game.

Network Connection/Client Arbitration

The DME can provide Session Master services either on a shared basis or in a dedicated fashion. This behavior is configured by the *ServerOwnership* and **NetOwnershipStatus** arguments in the call to **NetJoin**.

If ownership status is shared, the DME will provide service on an as-needed basis. For example, if the *SessionMasterStatus* status is shared, the first client to connect to the Server will become the Session Master. Later, if the Session Master client leaves the Server, the next available client will become the Session Master.

To assign a client explicit responsibility for a service, set the ownership to **OwnershipPrivate**, and set the ownership for all other clients to **OwnershipNone**. If an application expects to support more than two clients, the Session Master should have a high-speed connection to the Server.

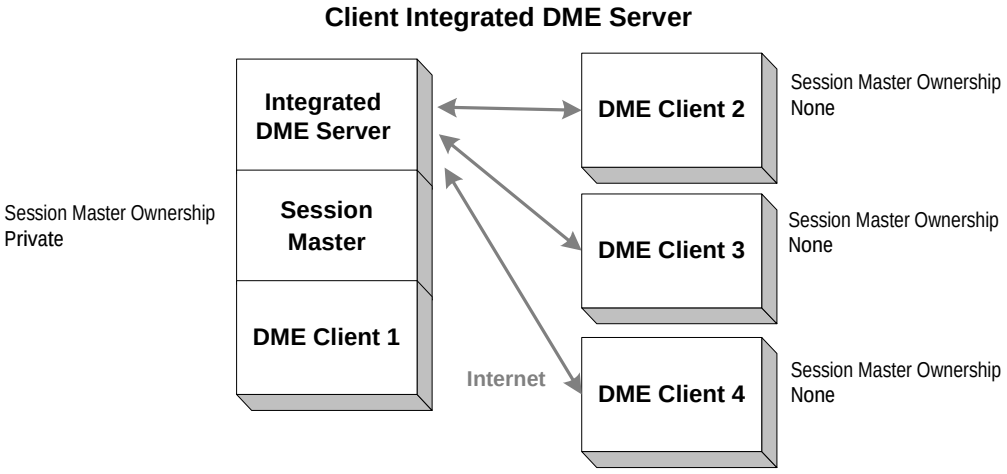
After connecting to the Server, each client will need a client index. If the client is not the Session Master, it must wait for the Session Master to grant a client index.

When a client joins a world, the client also requests creation rights for a block of objects and DataStreams. The Session Master then grants the client a block of objects and DataStreams. The client is now free to create objects and DataStreams. If the client is passive, it may request no objects or DataStreams and will only receive updates from other clients.

Client/Server Configurations

It is possible to configure the DME client, DME Server, and Session Master in many different topologies. The examples in Figures 1 through 3 show the unique capabilities, as well as the pros and cons, of some typical configurations. Each example shows the corresponding Session Master ownership.

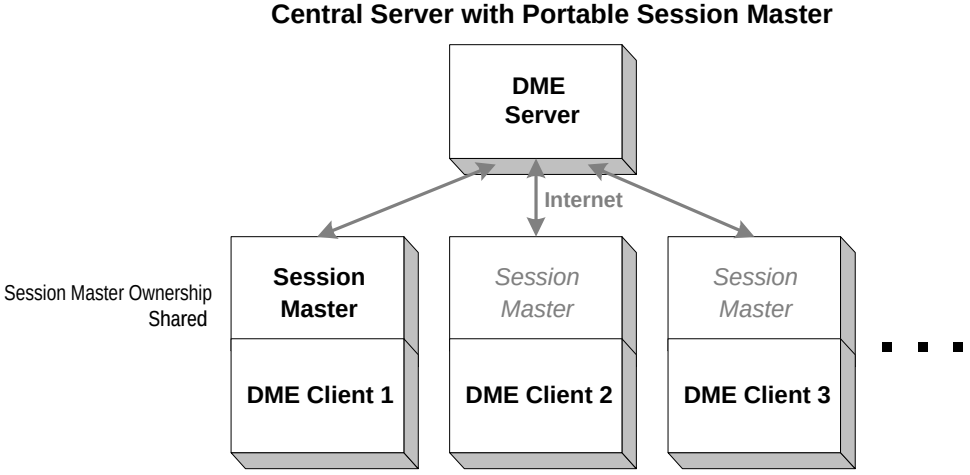
Figure 1



Pros: No external DME Server or Session Master required.

Cons: Number of clients limited by bandwidth capabilities of Host client; performance can be affected by application loading. If Host leaves, all clients lose connection.

Figure 2



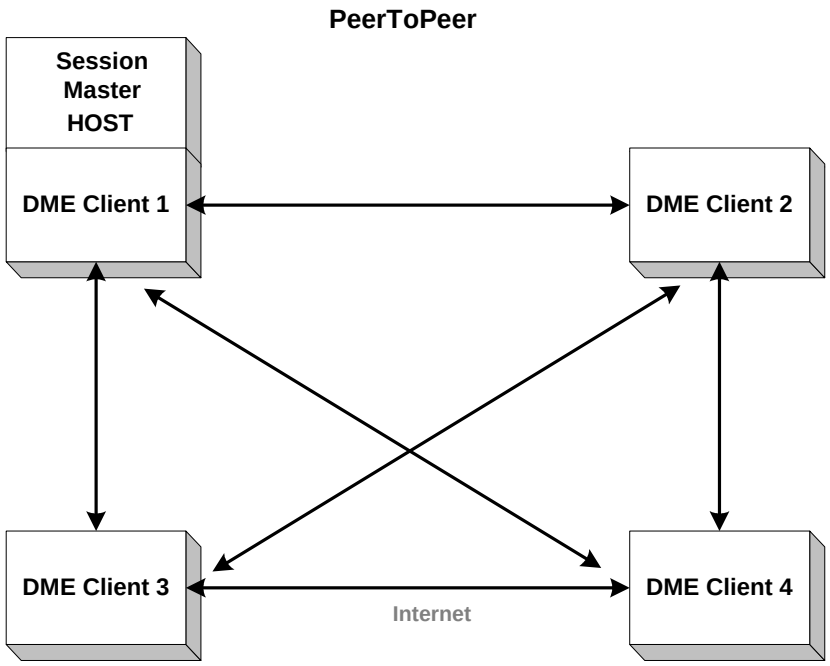
Pros: No external Session Master required, multiple games per Server, distributed Session Master loading.

Cons: Session Master not dedicated, transient. Not suitable for persistent worlds.

PeerToPeer Configurations

In Figure 3, there are four clients directly communicating with each other. Note that the Server has been replaced by an entity called a Host. In a peer-to-peer environment, the Host provides basic client management information. It notifies all clients when new clients connect and old ones disconnect. It does not perform the typical packet routing service of the DME Server. In a PeerToPeer configuration, each client communicates with the other via UDP; there is currently no client-to-client TCP support.

Figure 3



Pros: No backend Game Server required, smaller latencies in communication time between clients.

Cons: Not scalable beyond 8 or 16 players.

Network Protocol: TCP vs. Standard UDP vs. Reliable UDP

The DME Client API allows the application to set the protocol by which data is sent over the network. Any of the following three protocols can be used.

TCP

A program utilizing TCP can be guaranteed reliable in-order packet delivery. This means that any client sending out data is certain that the receiving client will get every packet in the same order in which it was sent out. One consequence of this protocol is that a client application may have to wait for network data if a packet gets dropped. TCP also performs other standoff conditions in sending and receiving that could unnecessarily delay an application from receiving data from a remote client.

Standard UDP

Standard UDP provides virtually no guarantees. An application cannot expect to receive every packet another client has sent out and it cannot expect in-order delivery.

Reliable UDP

Reliable UDP is the third option for data transport provided by the DME. The DME internally handles lost packets and retransmits them to the appropriate client. While Reliable UDP guarantees packet delivery, it does not explicitly guarantee in-order delivery. For an in-order Reliable UDP, you must set the transport flags on both messages and in the object's broadcast schedule to NET_ORDER_CRITICAL. Reliable UDP is only available in a PeerToPeer configuration.

Transport Flags: NET_DELIVERY_CRITICAL | NET_LATENCY_CRITICAL / NET_ORDER_CRITICAL

There are two areas in which transport flags are critical. From an application's point of view, these flags allow the DME client the ability to specify how a message should be sent out. You can specify a set of transport flags by passing (NET_DELIVERY_CRITICAL | NET_LATENCY_CRITICAL) (bitwise or) as a parameter.

Under normal circumstances, both the DME client and Server will aggregate data before sending it out over the network. NET_LATENCY_CRITICAL means that the DME should immediately send out the message. When data that is marked as latency critical is processed, all aggregated data is sent. After this occurs, the latency-critical message is immediately sent.

UDP Transport Modes

The DME allows for four different modes of UDP message delivery. Transport flags can be specified in **NetSendMessage**, **NetSendApplicationMessage**, **NetSendAppMessage** and in the *TransportFlags* field of **NetTypeBroadcastSchedule**. Two constants can be used separately or OR'ed together to create the delivery modes shown in Table 2.

Table 2

NET_DELIVERY_CRITICAL	NET_ORDER_CRITICAL	Behavior
0	0	Normal UDP, unreliable, out-of-order
1	0	Guaranteed delivery, out-of-order
0	1	In-order delivery, messages can be dropped
1	1	Guaranteed in-order delivery (like TCP)

Usage Examples:

For critical start game messages or object fields which have order dependency, set the transport flags to: NET_DELIVERY_CRITICAL | NET_ORDER_CRITICAL.

For object fields (i.e. position, velocity, angle) which are sent frequently, set the transport flags to: NET_ORDER_CRITICAL. This will prevent "back tracking jitter" and other visual artifacts associated with out-of-order updates.

For messages or object fields which are sent once or infrequently, set the transport flags to: NET_DELIVERY_CRITICAL.

Note: TCP is unaffected by these transport flags because it is always guaranteed in-order delivery. With Auxiliary UDP, if the NET_ORDER_CRITICAL or NET_DELIVERY_CRITICAL flag is specified the data will be transmitted over the TCP socket that is opened to the game server.

DME Initialization / Application Registration / NetConnect and NetJoin

Because of dependencies, application registration must proceed in the following order:

1. Call **NetInitialize()** (specify **NetConnectivityType**)
2. Register network object filter functions (if needed)
3. Register child *NetFields*
4. Register parent *NetFields*
5. Register *NetStructures* (used in the definition of a network object)
6. Register network object callback functions
7. Register DataStreams (if needed)
8. Register Packets (if needed)
9. Call **NetConnect()**
10. Call **NetJoin()**

NetConnect and NetJoin

(PeerToPeer)

```
/*Host of a PeerToPeer Game*/
NetHostPeerToPeerInParams  InParams;
NetHostPeerToPeerOutParams OutParams;
NetError = NetSetDefaultHostPeerToPeerParams(&InParams);
NetError = NetHostPeerToPeer(&InParams,&OutParams);

/*Clients wanting to connect to a PeerToPeer game (i.e., the host) will be
calling NetConnect. NetHostPeerToPeer replaces the need for the host to call
NetConnect.*/
```

(Client/Server)

```
NetErrorCode NetError;
NetConnectInParams  ConnectInParams; /*please see the library reference
                                      *for specifics on these structures*/
NetConnectOutParams ConnectOutParams;

NetError = NetSetDefaultConnectParams(&ConnectInParams,ConnectOutParams);
NetError = NetConnect(&ConnectInParams,ConnectOutParams);
```

NetConnect() is the primary means of connecting to a DME Server and/or a PeerToPeer host. This DME API function typically precedes all other DME API functions, except for **NetInitialize** and DME registration functions.

In **NetConnect()**, the field type for **NetTypeConnectCallback** – a parameter in the ConnectInParams structure – is set to a user-defined callback function that will be executed when the connection is completed. Until the connection is complete, the client must continue to call **NetUpdate()**. The client should not call **NetJoin()** until completing the connection. The connection callback will either succeed or fail. If the connection fails, the connection callback will be issued after a certain timeout period or upon an error.

DME Functionality After NetConnect

Once the DME client has successfully connected to a Server or a PeerToPeer host, but before it has joined (executed **NetJoin()**), the client may perform a few network-related operations. Most notably, the DME client can call **NetUpdate()** and **NetSetMyClientReceiveBroadcast()**, and send messages to other connected DME clients. However, it is recommended that an application always execute **NetJoin** before sending messages or enabling broadcasts (**NetSetMyClientReceiveBroadcast()**). One side effect of receiving messages before joining a world is that the DME client does not have client information regarding the source of the message.

NetJoin() performs several important operations that allow the DME client to send and receive state data related to network objects and DataStreams. After **NetJoin()** is executed, a Session Master is designated, client objects are created, network objects and DataStreams can be requested from the Session Master, global time base is initialized (synchronization generally takes about 60 seconds) and broadcast messages are enabled.

Once this initialization sequence is completed, the application can proceed to create objects and DataStreams for broadcast.

Broadcast Direction and Filtering

Use the *TargetClientIndex* field to direct messages and DataStreams either to an individual client or to all clients (for the latter, use *TargetClientIndex*=NET_SEND_TO_ALL_CLIENTS). Whenever a message, data stream or object update is broadcast to all clients, the scope of the broadcast can be limited by a filter function. Your application must provide the filter function, and return true if a client is allowed to receive the update.

Note: Each individual message type, data stream type and object can have its own filter function.

The DME API allows a client to enable broadcast via **NetSetMyClientReceiveBroadcast()**. This allows messages and updates that have a target client index of NET_SEND_TO_ALL_CLIENTS to receive data. This function is called after **NetJoin()** is executed. You can also call this function if your application needs to receive broadcasts before **NetJoin()** is executed. Likewise, an application can also disable broadcasts with this function.

Note: Enabling and disabling broadcasts can cause problems for clients, as they may miss critical DME messages, which are broadcast after **NetJoin()** is executed. It is recommended that you not call **NetSetMyClientReceiveBroadcast()**. Instead, if the application needs to send a broadcast message it can iterate through all of the clients by retaining the client indexes that are returned in the remote client connect callbacks registered in **NetConnect()**.

Network Fields and Structures

Network structures allow an application to define its own data object formats. These structures can be derived from ANSI C typedefs or C++ objects. These structures are constructed of recursively defined **NetTypeField**s (which allows for nested structures). A **NetTypeField** contains information about the type, the size, the offset and the element count (for arrays) of a data item. A **NetTypeField** also has a broadcast schedule that contains an error threshold and a minimum update interval.

Network Field Broadcast Scheduling

```
typedef struct
{
    unsigned int MinUpdateInterval; /* maximum rate limit for sending updates*/
    NetThresholdMethod ErrorThresholdType;
    union
    {
       float ErrorThresholdMagnitude; /*minimum delta for broadcast*/
       NetTypeErrorThresholdCallback pfThresholdCallback; /*callback function to
                                                            *executed*/
    }
    char TransportFlags;            /*used when sending field update*/
} NetTypeBroadcastSchedule;
```

The DME supports several types of broadcast schedules: rate-limited, error-limited and application-based.

Rate-limited broadcast schedules are good for fields that change very rapidly (for example, mouse-controlled rotation). The *MinUpdateInterval* is specified in milliseconds. It sets the maximum rate at which an update can be sent. For example, if the *MinUpdateInterval* is 100, then the maximum update frequency is 10 Hertz.

Error-limited broadcast schedules allow the application to only send updates when the values change beyond a specified tolerance.

Application-based broadcast schedules are performed by registering the **NetTypeErrorThresholdCallback**. When this callback is registered, the DME will invoke this callback every time the field changes, and allows the application to determine whether or not the field update actually needs to be broadcast.

As shown in Table 3, the error tolerance has six different modes:

Table 3

Mode	Action
NoThreshold	Never changed after Initialization (constant value)
ThresholdEquality	Any change will cause an update
ThresholdAbsoluteMagnitude	If change is greater than specified threshold, send an update
ThresholdRatioMagnitude	If change is greater than (value * ratio), send an update
ThresholdAnchorDelta	Currently not implemented
ThresholdCallback	Invoke a callback to determine whether the update should be sent out.

Rate-limited and error-limited methods can be used together to produce an optimal update broadcast schedule.

The data field *TransportFlags* in **NetTypeBroadcastSchedule()** allows the application to specify the transport mode in which an object update should be broadcast: NET_LATENCY_CRITICAL, NET_DELIVERY_CRITICAL and NET_ORDER_CRITICAL.

Example code: Register fields and structures to be used during network object creation

```
typedef struct                  // Application-defined child structure
{
   float   x,y,z;
} TypeMyVector;

typedef struct                  // Application-defined object structure
{
   float   Mass;
   float   Radius;
```

```
    TypeMyVector   Position;
    TypeMyVector   Velocity;
    TypeMyVector   Orientation;
unsigned char        FrameNumber;
} TypeMyObject;

// Code body
int               LastNetError;                        // Return type for DME functions.
NetTypeBroadcastSchedule  MyBroadcastSchedule;         // Broadcast Schedule for object
fields.
TypeMyVector              DummyVector;
// Needed only for field and
TypeMyObject              DummyObject;   // structure registration.
int                 NetFieldIndexTemp;                       // Field Index returned
(this sample ignores).
int                 NetStructIndexMyVector;     // Structure Index returned for
DummyVector.
int                 NetStructIndexMyObject;     // Structure Index returned for
DummyObject.

// Initialize a broadcast schedule to be used during field registration.  We may have a
variety of different
// broadcast schedules defined based on how we need to propagate field information.
MyBroadcastSchedule.MinUpdateInterval    = 100;               // Maximum update frequency
-> 10Hz.
MyBroadcastSchedule.ErrorThresholdType   = ThresholdEquality;  // Update on any change.
MyBroadcastSchedule.TransportFlags = NET_ORDER_CRITICAL;  //require updates to be received
in order

MyBroadcastSchedule.ThresholdData.ErrorThresholdMagnitude = 0;  // Used only if
ErrorThresholdType is
// not ThresholdEquality (thus, 0 ignored
// here).
// 1. Register child structures

// a. Register TypeMyVector's fields
NetRegisterObjectStart();
NetRegisterObjectField(   NetFieldIndexTemp,      // Registering float x
     DummyVector,
x,
FieldTypeFloat,
1,
MyBroadcastSchedule);                            // ← with this BroadcastSchedule.
NetRegisterObjectField(   NetFieldIndexTemp,      // Registering float y
     DummyVector,
y,
FieldTypeFloat,
1,
MyBroadcastSchedule);                       // ← with this BroadcastSchedule.
NetRegisterObjectField(   NetFieldIndexTemp, // Registering float z
      DummyVector,
      z,
      FieldTypeFloat,
      1,
MyBroadcastSchedule);                     // ← with this BroadcastSchedule.

// b. Register the TypeMyVector structure (this is a child to TypeMyObject)
LastErrorCode = NetRegisterStructure(&NetStructIndexMyVector, "MyVector",
sizeof(DummyVector));
if (LastNetError != NetErrorNone)
{
         printf("Error:NetRegisterStructure failed...\n");
    printf("Error Code:%d\n", LastNetError);
    return -1;
}
```

```
// 2. Register parent structures (these are used during network object creation)
// a. Register TypeMyObject's fields
NetRegisterObjectStart();
NetRegisterObjectField(   NetFieldIndexTemp,                    // Registering float Mass
      DummyObject,
Mass,
FieldTypeFloat,
1,
MyBroadcastSchedule);                      // ← with this BroadcastSchedule
NetRegisterObjectField(   NetFieldIndexTemp, // Registering float Radius
      DummyObject,
      Radius,
      FieldTypeFloat,
      1,
      MyBroadcastSchedule);                        // ← with this BroadcastSchedule.

NetRegisterObjectField(   NetFieldIndexTemp, // Registering child structure Position
      DummyObject,
Position,
FieldTypeChildStructure + NetStructIndexMyVector,
      1,
      MyBroadcastSchedule);                        // ← with this BroadcastSchedule.

NetRegisterObjectField(   NetFieldIndexTemp, // Registering child structure Velocity
      DummyObject,
      Velocity,
FieldTypeChildStructure + NetStructIndexMyVector,
1,
MyBroadcastSchedule);                      // ← with this BroadcastSchedule.

NetRegisterObjectField(   NetFieldIndexTemp, // Registering child structure Orientation
      DummyObject,
      Orientation,
 FieldTypeChildStructure + NetStructIndexMyVector,
      1,
      MyBroadcastSchedule);                          // ← with this BroadcastSchedule.

NetRegisterObjectField(   NetFieldIndexTemp, // Registering unsigned char FrameNumber
      DummyObject,
 FrameNumber,
 FieldTypeUnsignedChar,
      1,
      MyBroadcastSchedule);                      // ← with this BroadcastSchedule.

// b. Register the TypeMyObject structure
LastNetError = NetRegisterStructure(&NetStructIndexMyObject, "MyObject",
sizeof(DummyObject));
if (LastNetError != NetErrorNone)
{
         printf("Error:NetRegisterStructure failed...\n");
    printf("Error Code:%d\n", LastNetError);
    return -1;
}
```

Network Object Callbacks

After network object fields and structures have been registered with the DME, you can (optionally) define a set of callback functions. These callback functions are passed along as parameters for the **NetRegisterRemoteObjectCallback()** function, in order to handle the three events shown in Table 4.

Table 4

Callback/Event	Description
ObjectCreationCallback	Called when a remote client creates a new network object
ObjectUpdateCallback	Called when any registered network object field has detected a change (**Note:** This callback function will only be called once during **NetUpdate()**, even if more than one field has triggered this callback)
ObjectDeletionCallback	Called when a remote client has deleted a network object

Note: When client A creates a new network object on their machine, a network object creation message will be sent to, for example, clients B & C. The DME automatically manages the creation of this network object on the other machines and automatically initializes registered network object fields, this network object creation message can be used, for example, to initialize network object fields that were NOT registered with the DME.

ObjectUpdateCallback

The **ObjectUpdateCallback** has a special parameter called *FieldIndex*. This parameter is used to indicate several important events, as follows:

- Notify the client that a single field within an object has been updated
- Notify the client that the entire object has been updated (this notification occurs when *FieldIndex* is equal to NET_FULL_OBJECT_UPDATE)
- Notify the client of an object ownership update (this notification occurs when *FieldIndex* is set to NET_OBJECT_OWNERSHIP_UPDATE), allowing the client to determine whether it has gained private ownership of the object (that is, whether a **NetRequestObjectPrivateOwnership()** call has succeeded).

When triggered, these network object callback functions will be executed during the next **NetUpdate()** function call.

Example code: Register callback functions for a specific object type (structure index)

```
// Code body

// Function:        RemoteMyObjectCreationCallback
// Notes:   Registered fields will already have been initialized.
void RemoteMyObjectCreationCallback (int ClientIndex, int ObjectIndex)
{
        NetTypeObject*      pNetObject;
        TypeMyObject*       pMyObject;
        int         LastNetError;

printf("\n(Callback) A remote object has been created by:\n\tClientIndex'%d'\n\tObjectIndex'%d'\n", ClientIndex,
ObjectIndex);

        LastNetError = NetGetObject(&pNetObject, ObjectIndex);
        if (LastNetError != NetErrorNone)
        {
            printf("Error: NetGetObject failed...\n");
           printf("Error Code:%d\n", LastNetError);
           return;
        }
        // Gain access to the Object Data.
        pMyObject = (TypeMyObject*)(pNetObject->CurrentObjectData);
```

```
        // Do any special initialization here…  (For example initializing unregistered fields…)
}

// Function:        RemoteMyObjectUpdateCallback
// Notes:   This function can possibly be called many times per second; therefore, use caution if there is a
//          need to use a callback function such as this.
void RemoteMyObjectUpdateCallback (int ClientIndex, int ObjectIndex, int FieldIndex)
{
        NetTypeObject*      pNetObject;
        TypeMyObject*       pMyObject;
        int         LastNetError;

printf("\n(Callback) A remote object has triggered an update callback: \n\tClientIndex'%d'\n\tObjectIndex'%d'\n",
ClientIndex, ObjectIndex);


        if(FieldIndex == NET_OBJECT_OWNERSHIP_UPDATE)
        {
           LastNetError = NetGetObject(&pNetObject,ObjectIndex);
           if(LastNetError != NetErrorNone)
           {
              printf("App:[%s][%d]Error:%d\n",__FILE__,__LINE__, LastNetError);
              return;
           }
           if(pNetObject->OwnerClientIndex == g_MyClientIndex)
           {/*A PREVIOUS REQUEST OF PRIVATE OWNERSHIP HAS BEEN GRANTED (NetRequestObjectPrivateOwnership)*/

               // Assign values to the current network object's data fields.
               pMyObject = (TypeMyObject*)(pNetObject->CurrentObjectData);
               pMyObject->Mass       = pMyObject->Mass * 2.0f;// With us gaining access to this object
                                                              // we are making the Object's Mass double for
                                                              // whatever reason.
               return;
           }
        }

        LastNetError = NetGetObject(&pNetObject, ObjectIndex);
        if (LastNetError != NetErrorNone)
        {
           printf("Error: NetGetObject failed...\n");
           printf("Error Code:%d\n", LastNetError);
           return;
        }

}

// Function:        RemoteMyObjectDeletionCallback
// Notes:
void RemoteMyObjectDeletionCallback (int ClientIndex, int ObjectIndex)
{
printf("\n(Callback) A remote object has been deleted:\n\tClientIndex'%d'\n\tObjectIndex'%d'\n", ClientIndex,
ObjectIndex);

        // Do any special garbage collection work here…
}

// Associate callback functions to this NetObject via the NetStructIndexMyObject.
LastNetError = NetRegisterRemoteObjectCallback(NetStructIndexMyObject,          // (From example above)
                                               RemoteMyObjectCreationCallback,  // Create Callback.
                                               RemoteMyObjectUpdateCallback,    // Update Callback.
                                               RemoteMyObjectDeletionCallback); // Delete Callback.
```

NetTokens

Tokens are numbers that represent ownership of game entities and are different from objects that could contain state information such as position, orientation or velocity. If a client owns the ID number of a token, it owns that token. It is recommended to use **NetTokens** when an entity in the game needs to track/transfer its ownership (see Table 6). The advantage of using **NetTokens** are simplicity and bandwidth savings.

Initialization

To use **NetTokens**, **NetConnect**/**NetHostPeerToPeer** must explicitly set *pInParams*.TokenParams.*bUseToken* to 1. To receive NetToken callbacks the *pInParams*.TokenParams.*pfTokenOwnershipNotifyCallback* needs to be set – this is the only DME callback that is related to **NetTokens**.

The NetToken system must be ready before an application can start to use it. An application can use **NetTokenSystemQuery** to query the token system status or preferably setup a system callback at *pInParams*.*pfSystemStatusCallback*, which will be called when the token system becomes ready at the beginning of the game.

Ownership

The host (a Peer2Peer game) or server (a Client/Server game) maintains all tokens in the game.

NetTokenRequest()/**NetTokenListRequest()** are used to obtain ownership of a free token/token list from the host/server.

NetTokenRelease()/**NetTokenListRelease()** are used to release ownership of a token/token list that a client previously owned to the host/server.

NetTokenQuery() is used to query the ownership of a token.

Once a NetToken is granted, it will remain under the same ownership until its owner releases it or its owner is disconnected from the game.

Lists

When requesting/releasing ownership of a large number of tokens, it is recommended to use the list function calls in the API, i.e., **NetTokenListRequest()**/**NetTokenListRelease()**, this offers bandwidth savings in the form of data aggregation. The NetToken callback will still be fired once for each individual NetToken even with the list function calls.

NetTokenLists use NetBitmasks that are explained further in this document.

Host Migration

When a host migration occurs in a peer to peer game, the token system will not be ready until the host migration process is complete (A call to **NetTokenSystemQuery()** will result in *pSystemReady* = 0). During this time any **NetTokenRequest()**/**Release()** calls will fail. The application must wait for **NetTokenSystemQuery()** to return *pSystemReady* = 1.

Network Objects

Network objects contain the state data of the world. For example, a network object could be used to store the position, velocity, and orientation of a car.

Network objects are based on a user-defined **NetTypeObject** (also referred to as NetObjects). When a network object is created, two buffers are allocated: *CurrentObjectData* and *LastGlobalObjectDataUpdate*. The client is allowed to modify the contents of the CurrentObjectData buffer (if the client currently owns it).

The *LastGlobalObjectDataUpdate* buffer is used by the DME to determine when changes have been made to the *CurrentObjectData* buffer. Once the network object has been created, updates will be sent automatically.

When a network object is created, one of the arguments specified is the objects' life span. Table 5 lists network objects and associated levels of persistence.

Table 5

Object Life Span Type	Object Persistence
ObjectLifespanOneUpdate	For one **NetUpdate()**
ObjectLifespanTimeout	Until timeout
ObjectLifespanClient	As long as creating client remains connected to Server
ObjectLifespanSession	As long as Session Master remains connected to Server
ObjectLifespanPermanent	Until Session Master explicitly deletes it

Example code: Create a network object.

```
// Code body
int                 NetStructIndexMyVector;                 // (From example above).
int                 NetStructIndexMyObject;                 // (From example above).
int                 MyNetObjectIndex;                       // Object Index returned from
        // CreateNetworkObject().
NetTypeObject               *pNewNetObject;                         // Pointer to a network object.
TypeMyObject                *pMyObjectData;                         // Pointer to TypeMyObject.

// Create local object.
// Note:    This will also trigger an ObjectCreationCallback if a callback is registered for this NetObject's
// NetStructIndexMyObject.
LastNetError = NetCreateObject(&MyNetObjectIndex, // Object Index returned.
                               OwnershipPrivate,  // Ownership status (currently set to client
                                                  // ownership).
                               ObjectLifespanClient,// Life span type (currently set to persist as
                                                    // long as creating client remains connected
                                                    // to the server).
                               0,                   // Life span in milliseconds (used only if applicable).
                               NET_OBJECT_NOT_FILTERED, // No filter currently being used.
                               500,                     // MaxUpdateInterval (currently set to 500ms;
                                                        // therefore, this will give us 2 full network
                                                        // object updates/sec regardless of how
                                                         // individual network object field Broadcast
                                                         // Schedules were set.
                              0,                        // Latency critical flag (true/false).
                                                         // Note: If set to true, then updates will be sent
                                                         // immediately while bypassing all possible
                                                         // aggregation algorithms and etc… .

                              NULL, // Copy of initial data (currently set to NULL for
                                    // this sample, but this is a way we can
                                    // initialize network objects based on the current
                                    // NetStructure being applied for creation.
```

```
                        "MyFirstObjectName", // Object name.
                        NetStructIndexMyObject);
        // NetStructureIndex that was returned to us
        // during structure registration of a network
        // object via the NetRegisterStructure()
        // function call.
        // Thus, we are creating a network object with
        // this structure's properties.
// Was the object creation successful?
if (LastNetError != NetErrorNone)
{
        printf("Error: NetCreateObject failed...\n");
   printf("Error Code:%d\n", LastNetError);
   return -1;
}

// Did we get a valid object index?
if (MyNetObjectIndex != NET_INVALID_INDEX)
{
// Define the client's network object (avatar) association. This function is used to set an object that
// the client created as its primary object.  Though not required to do so, it can be beneficial to have
// this information available. (For example: In such areas as network object filtering).
        LastNetError = NetSetMyClientObject(MyNetObjectIndex);
}

// Get a pointer to the newly created network object via the Index returned from NetCreateObject.
LastNetError = NetGetObject(&pNewNetObject, MyNetObjectIndex);
if (LastNetError != NetErrorNone)
{
    printf("Error: NetGetObject failed...\n");
    printf("Error Code:%d\n", LastNetError);
    return;
}
if (pNewNetObject == NULL)
{
        printf("Error: There was no NetObject with index %d...\n", MyNetObjectIndex);
        return -1;
}
// Assign values to the current network object's data fields.  Since we created and have private ownership of
this
// network object, this data will be automatically propagated to all other connected clients (for the first time)
// during the next call to NetUpdate().
pMyObjectData = (TypeMyObject*)(pNewNetObject->CurrentObjectData);
pMyObjectData->Mass          = 5.0f;
pMyObjectData->Radius                = 1.0f;
pMyObjectData->Position.x            = 0.0f;
pMyObjectData->Position.y            = 0.0f;
pMyObjectData->Position.z            = 0.0f;
pMyObjectData->Velocity.x            = 0.0f;
pMyObjectData->Velocity.y            = 0.0f;
pMyObjectData->Velocity.z            = 0.0f;
pMyObjectData->Orientation.x = 0.0f;
pMyObjectData->Orientation.y = 0.0f;
pMyObjectData->Orientation.z = 0.0f;
pMyObjectData->FrameNumber   = 1;

// Network object and network object's data fields have been created and initialized…
```

Network Object Ownership

Network objects have a separate *CreatorClientIndex* and *OwnerClientIndex* field within the **NetTypeObject** structure. This allows object ownership to be decoupled from creation. If the **OwnerClientIndex** is equal to NET_OBJECT_OWNERSHIP_SHARED, then any client can request private ownership of that object. There are two ways this can be done; the first is to allow the DME to determine whether a client can have ownership and the second is to allow the owner of the object to define a callback to handle this process.

Allowing the DME to control ownership.

- When **NetJoin** is called, *NetJoinInParams*.***pfOwnershipUpdateCallback*** should be set to NULL.
- Request private ownership from the client who created the object
- The DME will automatically send back a response granting ownership if the object is shared.
- The person who requested ownership can modify the object.
- Private ownership is released

Note: An object can only be modified when the *OwnerClientIndex* is equal to **NetGetMyClientIndex()**.

Example code: Request private ownership, release private ownership

```
int WhateverObjectIndex      = 0;     // Whatever shared object we are requesting private ownership for.

// Request write access to a shared object. (Thus setting shared state to private).
LastNetError = NetRequestObjectPrivateOwnership( WhateverObjectIndex);
if (LastNetError == NetErrorObjectNotShared)
{
   printf("Warning: Could not gain private ownership of ObjectIndex %d because it was currently not in a
           shared state.\n", WhateverObjectIndex);
}
else if (LastNetError != NetErrorNone)
{
         printf("Error: NetRequestObjectPrivateOwnership failed...\n");
   printf("Error Code:%d\n", LastNetError);
}

 /* Call  NetUpdate(); until  the Object Update Callback is kicked
  * and you have been granted private ownership of that object
  */
/*modify the object*/
// Release ownership. (Thus setting private ownership back to shared ownership).
LastNetError = NetReleaseObjectPrivateOwnership(WhateverObjectIndex);
```

Allowing the application to control ownership.

- When **NetJoin** is called, **NetJoinInParams**.*pfOwnershipUpdateCallback* should be set to your user-defined function.
- Request private ownership from the client who created the object
- The creator of the object has the *pfOwnershipUpdateCallback* invoked and this function returns whether the ownership is granted or not. The application can use other criteria to determine whether or not the client should be granted ownership.
- The object can be modified
- Private ownership is released

```
//1)  Client Requesting Data
// Request write access to a shared object. (Thus setting shared state to private).
LastNetError = NetRequestObjectPrivateOwnership( WhateverObjectIndex);
if (LastNetError == NetErrorObjectNotShared)
{
   printf("Warning: Could not gain private ownership of ObjectIndex %d because it was currently not in a
           shared state.\n", WhateverObjectIndex);
}
else if (LastNetError != NetErrorNone)
{
         printf("Error: NetRequestObjectPrivateOwnership failed...\n");
   printf("Error Code:%d\n", LastNetError);
}

//2)OWNER OF THE OBJECT
NetJoinInParams.pfOwnershipUpdateCallback=AppNetObjectOnwershipUpdateCallback;
..
..
NetObjectOwnershipType AppNetObjectOwnershipUpdateCallback(HDME ConnectionHandle,int ClientIndex, int
ObjectIndex, void *pUserData)
{
      //Application specific code to process the request
      //The application can return an enumerated value from NetObjectOwnershipType

    return NetObjectOwnershipGranted;
}
//3 Client requesting ownership

/* Call  NetUpdate(); until  the Object Update Callback is kicked
  * and you have been granted private ownership of that object
  * A response may not come back if the client who owns the object is leaving or has ungracefully left the
game.
  * The application may have to request ownership of the object again.
  */
/*modify the object*/
//Release ownership (Thus setting private ownership back to shared ownership).
LastNetError = NetReleaseObjectPrivateOwnership(WhateverObjectIndex);
```

Remote Object Creation

Once the Session Master creates a shared object, the object can be modified by requesting private ownership.

All remotely spawned objects must have a unique name. If the object already exists, the call to **NetRequestCreateRemoteNamedObject()** will return the corresponding object index.

Example code: Request remote object creation

```
HDME g_MyConnectionHandle;                    // This is assigned during the NetConnect() call.
char        g_RemoteObjectName[] = "SoccerBall_01";    // Name to be submitted for a Remote Object to be
created.

// Function:        RemoteObjectCreationCallback
// Notes:   Remote object creation callback notification function.
//          To be registered with the NetRegisterRemoteObjectCallback() function for whatever NetStructure
//          we are associating this with.
void RemoteNamedObjectCreationCallback(int ClientIndex, int ObjectIndex)
{
   int SessionMasterClientIndex;
   NetGetSessionMasterClientIndex(&SessionMasterClientIndex, g_MyConnectionHandle);

   // Are we the Session Master at this point receiving this ObjectCreationCallback?  If so, we are going to
   // be requesting private ownership of this network object…
   if (ClientIndex == SessionMasterClientIndex)
   {
            TypeNetObject    *pThisNetObject;
            NetGetObject(&pThisNetObject, ObjectIndex);                // Should check NetErrorCode.
            if (pThisNetObject != NULL)
            {
                // Is this the object I requested?  (ie. The one with the g_RemoteObjectName?)
                if (strcmp(pThisNetObject->Name, g_RemoteObjectName) == 0)
                    {
                            // Request write access (private ownership) to a shared object.
                            //After this function returns call NetUpdate until the ObjectUpdate
                            //calls triggers and let's you know you have private ownership.
                            //DO NOT CALL NetUpdate within the context of a callback.
                            LastNetError = NetRequestObjectPrivateOwnership(    ObjectIndex);
                            if (LastNetError == NetErrorObjectNotShared)
                            {
                                    printf("Warning: Could not gain private ownership of ObjectIndex %d
                                    because it was currently not in a shared state.\n", ObjectIndex);
                            }
                            else if (LastNetError != NetErrorNone)
                            {
                                    printf("Error: NetRequestObjectPrivateOwnership failed...\n");
                                    printf("Error Code:%d\n", LastNetError);
                            }
            }
            }
   }
}

// Make a request for the Session Master to spawn a named object; whereas, we could be any client (A,B,C,etc)
// including the Session Master at this time making this request…
int SessionMasterClientIndex;
NetGetSessionMasterClientIndex(&SessionMasterClientIndex, g_MyConnectionHandle);
```

```
LastNetError = NetRequestCreateRemoteNamedObject(&ObjectIndex,                    // Object Index returned.
                                        OwnershipShared,                 // Ownership status (currently set to
                                                                         // client ownership).
                                        ObjectLifespanPermanent,         // Life span type (currently set to
                                                                         // persists until Session Master
                                                                         // explicitly deletes).
                                        0,                               // Life span in milliseconds (used
                                                                         // only if applicable).
                                        NET_OBJECT_NOT_FILTERED,         // Filter callback function.
                                        5000,                            // MaxUpdateInterval in milliseconds
                                                                         // (currently set to send full
                                                                         // object updates every 5 seconds).
                                        0,                               // Latency critical flag (true/false).
                                                                         // Note: If set to true, then updates
                                                                         // will be sent immediately while
                                                                         // bypassing all possible aggregation
                                                                         // algorithms and etc… .
                                        NULL,                            // Copy of initial data (currently set
                                                                         // to NULL for this sample, but this
                                                                         // is a way we can initialize network
                                                                         // objects based on the current
                                                                         // NetStructure being applied for
                                                                         // creation.
                                        g_RemoteObjectName,              // Object name.
                                        NetStructIndexMyObject,          // NetStructure index (using the
                                                                         // NetStructureIndex that was returned
                                                                         // to us during structure registration
                                                                         // via the NetRegisterStructure()
                                                                         // function call.  Thus, we are
                                                                         // creating a network object with this
                                                                         // structure's properties.
                                        SessionMasterClientIndex);       // → We designate the Session
// Master to spawn this object here.
if (LastNetError != NetErrorNone)
{
        printf("Error: NetRequestCreateRemoteNamedObject failed...\n");
        printf("Error Code:%d\n", LastNetError);
}

if (ObjectIndex != NET_INVALID_INDEX)
{
        printf(  "Requested object %s already exists on client %d with index %d\n",
                g_RemoteObjectName, SessionMasterClientIndex, ObjectIndex);

}
```

Network Object Filters

Filter functions can be used to control the propagation of network objects. These filter functions are give a pointer to a **NetObjectFilterData** structure. This structure will be filled out with a variety of information regarding the pending object update. One field of the structure is a **NetClientList** (see **NetBitMask** and **NetClientList**) which may be manipulated by the filter function to define a destination client subset.

To enable object filtering, the application must first register the filter function by calling **NetRegisterObjectFilter()**. The object is then created using the object filter index returned from the call to **NetRegisterObjectFilter()**.

A client can be associated with a network object by calling the **NetSetMyClientObject()** function. This allows the filter function to use any of the fields in the associated network object to determine if the client can receive the object.

Note: To create an unfiltered object, use the constant NET_OBJECT_NOT_FILTERED for the object filter index argument of **NetCreateObject()**.

Example code: Object filter callback function and registration

```
// Filter function based on the client's team.
int FilterObjectsByTeam(int TargetClientIndex, int SourceNetObjectIndex)
{
    MyClass*        pClass = (MyClass*)pFilterData->pUserData;
    unsigned int    i;
    int             b_set;

    // override the default NET_SEND_TO_ALL_CLIENTS
    pFilterData->ClientList.TargetClient = NET_SEND_TO_CLIENT_MASK;

    // the default bitmask setting specifies all clients that are currently NetJoined
    for (i = 0; i < pFilterData->ClientList.ClietnMask.max_id; i++)
    {
        // is this client in the bitmask?
        NetBitMastIsSet(&pFilterData->ClientList.ClientMask, i, &b_set);
        // yes! So filter!
        if (b_set)
        {
            // does the client need to receive the update?
            if (!ShouldClientReceiveUpdate(pClass, pFilterData->ObjectIndex, i))
            {
                // no, so clear the bit
                NetBitMaskUnSet(&pFilterData->ClientList.ClientMask, i);
            }
            // else the client will receive the update
        }
    }
}

// register object filter functions
NetRegisterObjectFilterInParams in;
NetRegisterObjectFilterOutParams out;
```

```
LastNetError = NetSetDefaultRegisterObjectFilterParams(&in);
// handle error...
in.ObjectFilterCallback = FilterObjectsByTeam;
in.pUserData = this;

LastNetError = NetRegisterObjectFilter(&in, &out);
// handle error...

// save off the filter index for later use
m_TeamFilterIndex = out.ObjectFilterType;
```

DataStreams

DataStreams are well suited for sending very large objects (such as texture maps or polygon meshes). The primary advantage of using **DataStreams** is precise control over the outgoing bandwidth.

Each data stream type needs three handlers: an update callback, a stream end callback, and a filter function. These handlers are called when remote **DataStreams** are being received. If the remote buffer flag is set when a data stream is opened, the DME will buffer the entire data stream object on the remote client machines. Once a data stream has been registered, sending the data takes three calls:

- **NetOpenDataStream()**
- **NetAddToDataStream()** (this can be called any number of times)
- **NetEndDataStream()**

The client can add to the data stream at any rate without affecting the outgoing bandwidth. To ensure that the data stream buffer never overflows, the application should not add more bytes than the value returned by the **NetDataStreamBytesFree()** function.

Note: To ensure that the broadcast scheduler will never fall behind, the actual outgoing bandwidth is a few percentage points higher than the value requested. Also note that the data stream does not close until all pending data has been sent.

If the client has registered a filter function for this data stream, the client DME will call the registered filter function to determine which clients should receive the data stream.

Example code: register, open, add to and end a data stream

```
#define Squared(arg) ((arg) * (arg))
#define PointWithinRadiusOfPoint(arg1, arg2, radius)                          \
                                    ( (Squared((arg1)->x - (arg2)->x) +       \
                                       Squared((arg1)->y - (arg2)->y) +       \
                                       Squared((arg1)->z - (arg2)->z)) <= Squared(radius))

#define     AudioStreamChunkSize 100
#define     MaxAudioReceiveDistance 20000 // 20 meters
int DataStreamTypeAudio; // to save the results of registration
int MyDataStreamChannel;

// my data stream callback function
void AudioDataStreamUpdateCallback(int ClientIndex, int Channel, char *NewData, int ByteCount)
{
   AddWaveDataToAudioPlaybackBuffer(NewData, ByteCount);       // play the sound
}

// audio data stream position filter function
int AudioDataStreamFilterCallback(int TargetClientIndex, int SourceClientIndex, int StreamChannel)
{
  int NetError = NetErrorNone;

  NetTypeClient ** TargetClient;             //dme type
  NetTypeClient ** SourceClient;
  NetTypeObject ** TargetClientObject;       //dme type
  NetTypeObject ** SourceClientObject;
  MyNetObject * TargetClientObjectData;      //user defined type
  MyNetObject * SourceClientObjectData;      //user defined type
```

```
  //error checking omitted for brevity.
  NetError = NetGetClient(SourceClient, SouceClientIndex);
  NetError = NetGetClient(TargetClient, TargetClientIndex);

  NetError = NetGetObject(SourceClientObject,((NetTypeClient *)SourceClient)->ClientObjectIndex);
  NetError = NetGetObject(TargetClientObject,((NetTypeClient *)TargetClient)->ClientObjectIndex);

  TargetClientObjectData =
    (MyNetObject *)(((NetTypeClient *)TargetClientNetObject)->CurrentObjectData);
  SourceClientObjectData =
     (MyNetObject *)(((NetTypeClient *)SourceClientNetObject)->CurrentObjectData);

  if(PointWithinRadiusOfPoint( &TargetClientObjectData->Position,
                        &SourceClientObjectData->Position,
                              MaxAudioReceiveDistance))
  {
    return 1;
  }

  return 0;
}

//separate functional block
int NetError;
int DataStreamTypeAudio,MyDataStreamChannel;
int DataStreamBytesFree;
.
.
.
// register remote data stream update handler
NetError = NetRegisterDataStream(&DataStreamTypeAudio,  //return value, the stream type
                              AudioDataStreamUpdateCallback, // called on creation and updates
                              NULL, // called on completion of the data stream
                              AudioDataStreamFilterCallback);  // my filter function

// open my outgoing data stream channel
NetError = NetOpenDataStream(&MyDataStreamChannel, //return value
                          512, //buffer size,
                          500, //output bandwidth in bytes per second
                          DataStreamTypeAudio, //stream type comes from registration
                          0, //remote buffer flag (true/false)
                          NET_SEND_TO_ALL_CLIENTS, //target client, (global broadcast)
                          1, //circular buffer flag (true/false)
                          1, //minimum packet size (for aggregation)
                          MaxStreamPacketSize); //maximum packet size (for granularity)

if (NetError !=NetErrorNone) //was the channel creation successful ?
{
  printf("open data stream failed. \n");
  return –1;
}
// check for available space.
NetError = NetDataStreamBytesFree(&DataStreamBytesFree,MyDataStreamChannel);

if(DataStreamBytesFree >= AudioStreamChunkSize)
{
  NetAddToDataStream(MyDataStreamChannel, (char*)AudioDataSource, AudioStreamChunkSize);
}// end the data stream
NetError = NetEndDataStream(MyDataStreamChannel);
```

Network Messages

Network messages are intended to communicate events that are transient in nature (such as explosions or chat messages). To use a network message, register your packet structure by calling **NetRegisterMessage()**. This function returns the packet type for the new message. To send a message, fill out all of the fields of your message structure and call **NetSendAppMessage()**. The message can be directed to an individual client, or to all clients by using NET_SEND_TO_ALL_CLIENTS, or to a list of clients using **NetClientList** (see **NetClientList** and **NetBitMask**).

Message filtering can be achieved by specifying an appropriate subset of clients via the NetClientList field passed into **NetSendAppMessage()**. The **NetSendFilteredMessage** API has been deprecated in favor of this functionality.

The optional legacy APIs **NetSendMessage()** and **NetSendApplicationMessage()** are still available, but we recommend that developers switch to using **NetSendAppMessage()** exclusively as the legacy functionality will be deprecated in a future release.

Note: When an application receives a message callback, it must return the appropriate message size. Failure to properly return the correct size of the message will result in a *NetErrorBadPacketReceived* being returned from **NetUpdate()**. In addition, this will cause all data contained in the buffers to be lost, as the DME will not be able to parse the rest of the messages.

Example code: Register and send an application-specific message

```
// user defined message
typedef struct
{
  char  ChatBuffer[128];     // large text buffer for typing messages
} TypeChatMessage;

unsigned int  MessageType; // the message type returned by NetRegisterMessage()
TypeChatMessage  MyChatMessage; // declare an instance of MyChatMessage

// user defined message parser
int ChatMessageHandler (HDME ConnectionHandle, int WorldID, int SourceClientIndex, void *MessageData);
{
  int NetError;
  NetTypeClient *SourceClient;
  TypeChatMessage *ThisMessage = (TypeChatMessage *)MessageData;

  printf("message received from client %d: %s \n", SourceClient->Name, ThisMessage->ChatBuffer);
   return sizeof(TypeChatMessage);   // NOTE: must always return the total size of the packet
}

// register user defined message and message parser
NetRegisterApplicationMessage(&(MessageType), ChatMessageHandler); //generic message class
                    .
                    .
                    .
        // send a user defined message to all clients
NetSendAppMessageInParams in;
NetSendAppMessageOutParams out;

sprintf(MyChatPacket.ChatBuffer, "Hello world");

// setup a default in params structure
NetSetDefaultAppMessageParams(&in);
```

```
// fill out the necessary bits
in.ConnectionHandle = hDME;
in.TransportFlags = NET_LATENCY_CRITICAL;
in.MessageType = MessageType;
in.MessageLength = sizeof(MyChatMessage);
in.MessageData = (unsigned char*)&MyChatMessage;

// define the destination clients (all connected clients)
NetGenerateClientList(hDME, &in.DestClient);

NetSendAppMessage(&in, &out);
```

Stream Media

Stream Media provides the ability for an application to perform audio and video data transfer over the network. The DME supports several types of Audio codecs, which are used for compression and encoding of a raw stream of data. The DME manages the transport and distribution of the data for the application. The application simply enables Stream Media functionality within the DME and passes audio or video data to the DME for transport.

Note: Stream Media is a completely separate feature from DataStreams in the DME.

Enabling Stream Media

- Before connecting, the application must set up the **NetStreamMediaParams** in **NetConnectInParams**. Two critical callbacks must be registered, the record callback and the play callback. These callbacks give the application the ability to transport both video and audio data.
- Connect to a server or PeerToPeer host. Stream Media must be enabled at connection time. Once the application connects, there is no way to enable Stream Media without disconnecting and reconnecting.
- Join a Stream Media Channel. A stream media channel gives the application the ability to have clients segregated based on a channel ID. For example, if all clients are connected together in a game and two channels have been enabled at connection time, Team A can be placed in channel 0 and Team B can be placed in channel 1. Clients in different channels will not be able to hear each other. Each channel must have at least two people in it before recording can be enabled.
- Call **NetUpdate()**

The play callback will be invoked by the DME whenever there is data to be played. It will return the data to the application, and the application must play it through the TV or headset. The record callback is invoked every time **NetUpdate** is called. If the application has data to pass to the client, then it should feed the DME the record data in the record callback itself. If there is no data, then the record callback will do nothing. With an audio media stream, once the application has completed recording it must call **NetStreamMediaEndRecording**.

Stream Media Channel

In order to record and playback data, the application must join a channel. The number of channels created is specified at connection time in the **NetStreamMediaParams** structure. If there is only one client in a channel, the DME will not allow the application to record and send data. There must be at least two clients connected for recording to work.

Controlling Who Can Talk

With only two players in a channel, it's possible for both clients to be able to speak at the same time. If there are more than two people in a channel, it is up to the application to control who can speak. If multiple people attempt to talk at the same time, only one client will be heard; the other clients' data will not be sent out.

Note: It is recommended that **NetTokens** be used if you wish for only one person to talk at a time.

Distribution of Stream Media Data

Stream Media data is distributed in a PeerToPeer manner and is aggregated together with game data in the same connected session. Data distribution must be a consideration as client numbers increase, as the transport capacity of the specified mode may be exceeded. Two modes of transport are supported: **NetStreamMediaGridTypeRelay** and **NetStreamMediaGridTypeDirect**. An application will benefit from

NetStreamMediaGridTypeRelay if there are more than four players connected to the same StreamMedia channel.

In **NetStreamMediaGridTypeRelay** mode, streaming data is relayed to only a small fraction of the clients connected and then relayed again to the remaining clients. With this method higher client counts can be supported. A drawback to this method is that the data stream is delayed in reaching all of the clients, as it must be passed to multiple clients before reaching everyone (higher latency). Also it has greater potential to be dropped, since there is are extra points through which the data must be sent before all clients finally receive it.

The second method is called **NetStreamMediaGridTypeDirect**. In this model, when a client sends StreamMedia data, that client will send the data directly to every other connected client. In **NetStreamMediaGridTypeDirect**, clients do not relay data they receive on to other clients. **NetStreamMediaGridTypeDirect** is the most efficient choice for applications with four or fewer connected clients per StreamMedia channel.

As of SCE-RT Release 2.9, it is now possible to specify the *TransportFlags* that will be used when transmitting StreamMedia data (i.e. NET_DELIVERY_CRITICAL, NET_LATENCY_CRITICAL, etc.).

Supported Audio Codecs

NetStreamMediaAudioTypeRaw – The application is sending data in the native format of the device. Selecting the Raw codec will generally result in higher bandwidth and is only recommended for testing. The raw codec roughly sends at a rate of 20.0 KB/second per client.

NetStreamMediaAudioTypeCustom –The application is using a custom codec and is handling all of the required encoding and decoding. The distinction between Raw and Custom is necessary since the DME will notify the application of the codec being used in the play callback.

NetStreamMediaAudiotTypeLPC –This codec roughly sends at a rate of 2.6KB/second per client.

NetStreamMediaAudioTypeLPC10 –This codec roughly sends at a rate of 2.0 KB/second per client

NetStreamMediaAudiotTypeGSM – The bandwidth requirement for sending a GSM stream is roughly 3.6KB/second. This codec generally has the best speech quality as compared with LPC and LPC10.

Peripheral Management – Headset or Camera

The application must manage all aspects of the peripheral. The DME will simply take data and distribute it to the other clients. If the peripheral is unplugged, the application must deal with that by throwing away the data received in the play callback and not passing data to the record callback.

NetTokens vs. NetObjects vs. NetStreamMedia vs. NetMessages

There are pro's and con's for using **NetTokens**, **NetObjects**, **NetStreamMedia**, and **NetMessages** for different features in your game. The following would be ideal usage for these particular DME API calls given the scenario of a First Person Shooter (FPS) game.

Table 6

Action	DME API call	Reason:
Player got flag race condition (in a Capture the Flag game)	**NetToken**	Ownership is the only thing that changed.
Player entered a vehicle and is driving it	**NetObject**	Positional data required. Game needs to know where vehicle is at all times.
Player fired machine gun	**NetSendAppMessage** (upon start and stop of machine gun fire)	Bullets lifespan too small. Too much overhead to create a NetObject.
Video of player reaction (if using EyeToy)	**NetStreamMedia**	Constant Video Stream. Can also use DataStreams but NetStreamMedia is specifically designed for audio and video streams.

Masking Functionality

The **NetBitMask** structure defines the format for the DME's generic bitmasking functionality. This is used as a simple abstraction to define a subset or list of members based on an integer index.

An array of 256 bits is used as an array of true/false values. These true/false values will be used by various functions to define a list of clients/tokens/objects/etc that should be operated upon.

NetBitMask.*base_id* defines the base index of the bitmask array. If *base_id* is set to 256, then bit 0 is mapped to identifier 256 and bit 255 is mapped to identifier 511.

NetBitMask.*max_id* defines the maximum count that will be used by the bitmask array. This is useful for internal loop processing optimizations. **Note:** *max_id* is 1 based instead of zero based, thus the maximum allowed value for *max_id* is 256.

Bitmask Manipulation

The DME client also defines the following APIs as a set of generic bitmask manipulation helper functions:

- **NetSetDefaultBitMask** – zeros out bitmask structure and sets *max_id* to 256
- **NetBitMaskIsSet** – Takes a pointer to a **NetBitMask** structure and an index within the bitmask range, and sets *b_set* to 1 if the bit is set or 0 if not. **Note:** The *mask_index* parameter must be contained within the range of *base_id* and *max_id*. Thus: *pBitMask->base_id* <= *mask_index* < *pBitMask->max_id*
- **NetBitMaskSet** – Takes a pointer to a **NetBitMask** structure and sets an index within the bitmask range.
- **NetBitMaskUnSet** – Takes a pointer to a **NetBitMask** structure and zeros an index within the bitmask range.

Client List Functionality

Building on the generic **NetBitMask** functionality, SCE-RT has also defined a method for specifically defining a subset or list of clients. The **NetClientList** structure is the base datatype for this implementation.

Along with the generic **NetBitMask** manipulation functions the developer may also use the following helper APIs when working with the **NetClientList** structure. (See Masking Functionality for **NetBitMask** description).

Client Lists API

- **NetGenerateClientList()** – Fills out a **NetBitMask** structure that maps to the current clients connected to the game.
- **NetGenerateJoinedClientList()** – Fills out a **NetBitMask** structure that maps to the current clients NetJoined to the game.

Sending a Message Using Client Lists

To send a message which takes advantage of the **NetClientList** definition you must use **NetSendAppMessage()**. It will send a message according to the parameters defined by the **NetSendAppMessageData** structure.

WARNING: **NetSendAppMessage()** deprecates the functionality provided by **NetSendMessage()** and **NetSendApplicationMessage()**. These functions will be removed in a future release in favor of **NetSendAppMessage()**.

Transmission Latency Issues

There are many factors that add delays to the transmission of packets to the Session Master and remote clients. In the following example a 100-byte TCP packet (including header) is sent to the Session Master for filtering and is rebroadcast to a remote client. Table 7 lists the major causes and magnitude of transmission latency (in milliseconds), in order of effect.

Table 7

Cause of Delay	Best Case	Average Case	Worst Case
Sending client calling **NetUpdate** @ 30Hz	0	16	33
Modem hardware compression	50	70	100
Transmission delay @ 5 K bytes / second	20	20	20
Internet router delays	5	20	500
Reception delay @ 5 K bytes / second	20	20	20
Receiving client calling **NetUpdate** @ 30Hz	0	16	33
Total	95	162	706

Note: In the average case 48 milliseconds are spent waiting for NetUpdate to be called. This can be reduced to 24 milliseconds if NetUpdate is called 60 times per second.

Determining Network Latency: NetPing and NetPingIP

To determine the latency of another DME client, an application can utilize these two functions. The term "Ping" does not mean that an ICMP packet was sent out. Rather it means that the DME is sending out an internal echo message that will be returned by the receiving end. The latency in milliseconds is returned in the registered callback. If the Ping is not echoed back, it will eventually timeout and the callback will be issued with an error code indicating failure. The echo message is tagged as NET_LATENCY_CRITICAL and this overrides the aggregation intervals in both the DME client and Server. Specifying the target of **NetPing** to be SEND_TO_SERVER will measures the round trip time from the client to server and server back to the client.

NetPing has two separate and distinct behaviors depending upon the transport mode. If the transport mode is set to PeerToPeer UDP, then the echo will go directly to the client. If the transport mode is Client/Server TCP, the echo message will go through the Server to the destination client.

NetPingIP performs the same basic echo mechanism, however it requires an IP Address and the echo is sent via UDP. UDP must be enabled before calling this function.

For overall Latency metrics see Client Latency Metrics.

Client Latency Metrics

Overview

NetGetLatencyMetrics() can be used to get overall latency metrics about a particular client. This is different from **NetPing** and **NetPingIP** in that they are limited to metrics about just one trip (send and receive of that particular ping). Once a client issues a **NetConnect()**, **NetGetLatencyMetrics** will start to keep overall statistics of latency metrics.

Client/Server

For client/server game, the latency is defined as the round-trip time between that client and the server. To keep track of latency the server is internally periodically polling the clients latency with a specific message. When a call to **NetGetLatencyMetrics()** is made the client will send a specific message to the server and the server will reply with the metric data for that particular client.

Peer To Peer

For a peer to peer game, the latency is defined as the round-trip time between that peer and the querying peer. Each client is internally recording the latency for each DELIVERY_CRITCAL and LATENCY_CRITICAL message. When a call to **NetGetLatencyMetrics()** is made it will fetch the data from the clients local metrics data.

Initialization

A client must have issued a **NetConnect()** before a call to **NetGetLatencyMetrics()** can be made. **NetGetLatencyMetrics()** has **NetLatencyMetricsParams** as parameter. **NetSetDefaultLatencyMetricsParams()** must be used to initialize a **NetLatencyMetricsParams** to a set of default values.

The application will then need to setup a callback function in *pLatencyMetricsParams*.*pfLatencyMetricsCallback* to get notified by the arrival of the result data. Then the call to **NetGetLatencyMetrics()** can be made.

Latency Metrics

The host in a peer-to-peer game will always have zero latency about itself.

The latency metrics include:

- longest latency – longest recorded latency (*LatencyMax*)
- shortest latency – smallest recorded latency (*LatencyMin*)
- average latency – average recorded latency (*LatencyAvg*)

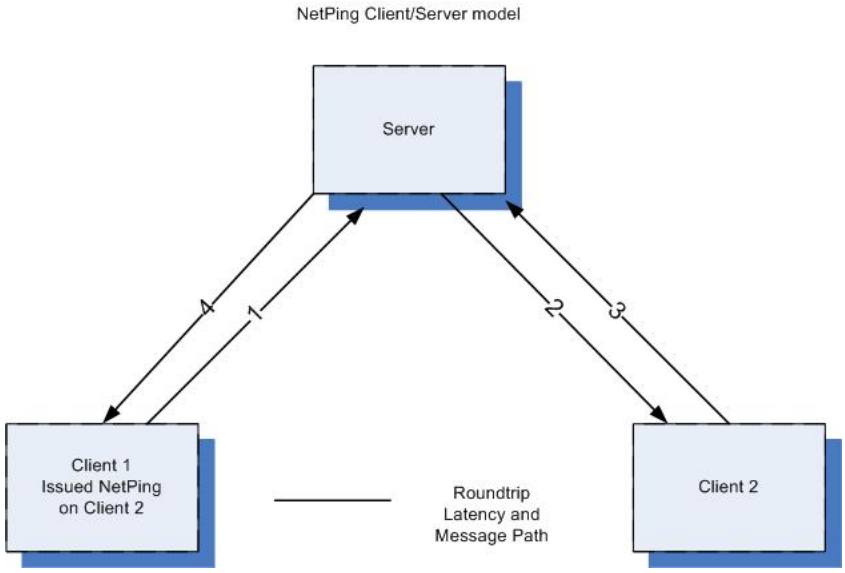
The application may setup a *pLatencyMetricsParams*.*pUserData* if it wants to use user data in the callback function.

NetPing vs. NetGetLatencyMetrics

Client/Server

NetPing

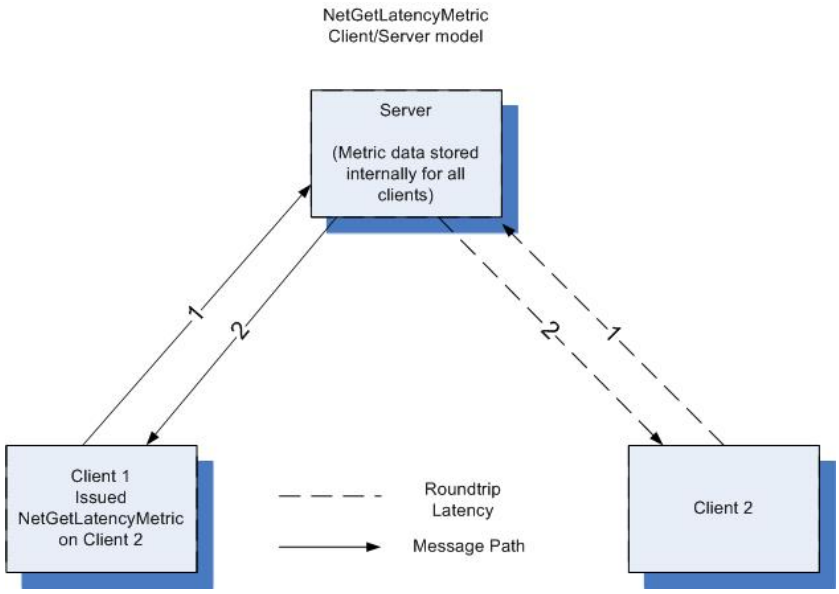
Figure 4



In the Client/Server mode, **NetPing** is used to ping the round trip between the two clients and the Server (see Figure 4). For a **NetPing** request issued from Client 1 about Client 2, the roundtrip latency and message path would go from Client 1 to the Server, Server to Client 2, Client 2 to the server, and the server back to Client 1.

NetGetLatencyMetrics

Figure 5



NetGetLatencyMetrics can be used for Client 1 to get the round trip time from Client 2 to Server, and Server back to Client 2. **NetGetLatencyMetrics** will offer improved bandwidth savings, as well as the overall latency metrics of clients.

Peer To Peer

For peer to peer game, **NetPing** and **NetGetLatencyMetrics** measure the round-trip time between that peer and the querying peer. The difference is **NetPing** measures one instance, while **NetGetLatencyMetrics** measures the overall statistics.

Data Structures

Table 8 lists the three atomic data structures used by the DME.

Table 8

NetTypeClient	Contains client info (i.e., *ClientObjectIndex*, *Name*, Binary encoded IP Address)
NetTypeObject	Contains object info (i.e., *Name*, *LifeSpanType*, *CurrentObjectData*)
NetTypeDataStream	Contains stream info (i.e., *BufferStart*, *RemoteBuffer*, *DataRate*, *Status*)

Reentrance in the DME Client

The DME will prevent multiple threads from calling into the DME. For example, if one thread is calling **NetUpdate** and then another thread attempts to call a function, say **NetGetTime**, the second thread will block until the first thread leaves **NetUpdate**. If the same thread is making multiple calls into the DME it will not be prevented from making other calls. A typical example of this would be to call **NetUpdate** and then call a function like **NetGetLocalTime** in registered DME callback all within the context of the same thread.

With this scheme, there are two things to keep in mind when developing. The first is to note that there is no protection on any of the memory that the DME maintains. For example, if the application stores a pointer to a DME Object in multiple threads, each thread could directly modify that memory location. An application should take some care to prevent this sort of access.

The second thing to keep in mind is deadlocks. Consider the case where an application has two threads and one thread must wait in a DME callback for the other thread to fill in information from another DME API call. In this situation, the application will block the first thread in the callback and therefore allow the second thread to run. However, the second thread will not be allowed to make its DME call, since the DME is still within the context of **NetUpdate()**. This will cause a deadlock as thread 1 requires information from thread 2, but thread 2 can not continue until thread 1 exits from the callback.

Likewise, there are some other restrictions about reentrance, which a DME client should not violate. When the flow of execution is within a DME registered callback event, the DME client must not attempt to perform the following operations:

- Call *NetUpdate*
- Attempt a connection, disconnection, “joins” or “leaves”
- Freeing DME Objects, DataStreams or other internal DME Structures

Simultaneous DME Client Connections

The DME client allows simultaneous connections to multiple servers. There is one key restriction with multiple connections; an application cannot control how often each connection is updated. NetUpdate performs updates for all DME connections. As a result, the DME client treats all connections equally, even if a connection only needs to send messages periodically. Any application that is utilizing multiple simultaneous connections should be prepared to handle all callbacks for all of the connections at any time.

DME Client and DME Server Aggregation

The term *aggregation* refers to how the DME client and the Server manage data to be sent out over the network. The simplest way of managing network data is to send it out as soon as it is received from the application above. This method has several problems, most notably high packet header overhead.

For example, suppose a field (a 4-byte value) within an object was changing once every 1msec. With aggregation (at 20msecs), the DME client would only send out data 1 packet after 20msecs. The benefits are huge when you consider that the size of the IP and TCP header is at least 40 bytes. Without aggregation over a 20msec interval, 20 packets at roughly 44 bytes per packet would be sent out.

Both the DME client and Server have a default aggregation interval of 30msecs. Both components can be individually configured based on the application's requirements. These aggregation intervals can make a big difference in how the DME client and Server application performs. If an application needs to override the aggregation interval for a particular message, then the application should set the transport flag to NET_LATENCY_CRITICAL.

NetLANFind: Locating Integrated Servers and PeerToPeer Hosts

The **NetLANFind** API is a subset of the DME API. It allows a developer to locate other DME enabled devices in the same Local Area Network (LAN). To use the API, you must include 'rt_lanfind.h' and link with librtbase. The **NetLANFind** API is only available when **NetInitialize** is called with **NetConnectivityType** set to **NetConnectivityLAN**. If the **NetLANFind** APIs are called when **NetConnectivityType** is set to anything other than **NetConnectivityLAN**, an error will occur. **NetConnectivityLAN** also applies and is used with the PlayStation Portable's (PSP™) Adhoc wireless mode.

This API replaces the **NetGameFind** API from SCE-RT versions prior to 2.9. **NetLANFind** will allow the following:

- The discovery of peers and game hosts;
- Discovering the **NetAddress** of other machines in your local area network;
- Ability to hide from other clients.

Initialization

NetUseP2P() and **NetUseCommInet()**, **NetUseCommEEnet()** or **NetUseCommAdhoc()** (see the NetUse section) *must* called before using **NetLANFind**. The **NetInitializeInParams**.*ConnectivityType* must also be set to **NetConnectivityLAN** when calling **NetInitialize()** in order to use **NetLANFind**. **NetLANFind** cannot be used with **NetConnectivityInternet** (Online).

Initiating the Search on the LAN

NetSetDefaultLANFindParams() must be called prior to **NetLANFind()**. The default parameter settings, when used unmodified, will instruct **NetLANFind** to locate only games that are running on the same port with the same *PlatformID* and *ApplicationID*. The developer may change any of the settings established by **NetSetDefaultLANFindParams()** by directly modifying the **NetLANFindInParams** structure before passing it to **NetLANFind()**. The system will use the *Username* set in **NetLANSetUserName()**.

Upon calling the **NetLANFind()** function, the DME broadcasts a pre-defined message onto the LAN which other DME devices receive. The developer is allowed to define a callback notification function (via **NetLANFindEnableExchange()**) at which point the receiver will be notified that another client is searching for peers and/or games. The application can then choose to either allow a reply to be sent, or to be "invisible" and prohibit a reply from being sent. If a peer allows replies to be sent, the replies are received via the *pfnLANFindCallback* that was specified using the **NetLANFindInParams** structure.

The common information that is exchanged between the sender and receiver (and visa versa) includes the **NetLANPeerDesc** structure. This information contains usernames, Network address, etc. Additional developer specific information can be exchanged between the sender and receiver (and visa versa) by utilizing the 'Details' structure members of the LANFind API. **Note:** Since the 'Details' is application-specific, make sure you validate that the message is being sent by applications you know how to interpret.

Session Types

- To search for Games on the LAN:
  The **NetLANFindInParams**.*SessionType* needs to be set to **NetSessionTypeGame**.
- To search for other peers on the LAN:
  The **NetLANFindInParams**.*SessionType* needs to be set to **NetSessionTypePeer**.
  Note: The recipients must of called **NetInitialize** for them to be located.
- To search for integrated servers on the LAN:
  The **NetLANFindInParams**.*SessionType* needs to be set to **NetSessionTypeIntegratedServer**.

NetLANFind Notification

By default, the DME will automatically respond to *all* **NetLANFind** requests. This may or may not suit your needs. To enable notification that clients are searching for your game, you must call **NetLANFindEnableExchange()**. This API allows you to specify a callback function (**NetLANFindExchangeCallback**) that the DME will call when a client is issuing a **NetLANFind** search.

When calling your **NetLANFindExchangeCallback** function, the DME will pass to you a complete description of the peer performing a search as well as an optional 300-byte user data blob. If you elect to respond, you may also set up to 300 bytes of user data in your response.

Data from the NetLANFind Request Source

Incoming data (**NetLANFindExchangeCallbackInArgs**) describes data regarding application that is issuing the **NetLANFind()**. It will contain the **NetLANPeerDesc** struct that can identify the applications *DmeVersion*, *NetPlatformID*, *ApplicationID*, *NetAddress*, *Localization*, *ApplicationName*, and *UserName*. This information can then be used to determine if your application will want to respond to the **NetLANFind** (see the next section).

Data Going to the NetLANFind Request Source

Replying to a **NetLANFind**:

Outgoing data (**NetLANFindExchangeCallbackOutArgs**) is used to tell the DME to respond and optionally send application specific data to the application that issued the **NetLANFind** request. **NetLANFindExchangeCallbackOutArgs**.*Details* is used to send application specific data. The application that issued the **NetLANFind** request will receive this information in **NetLANFindCallbackDataArgs**.*Details*. **Note:** It is very important to realize that the context of the information contained in

NetLANFindCallbackDataArgs.*Details* is application specific. You must be sure that you know how to parse this data before attempting to read it.

Hiding from **NetLANFind**:

An application must use **NetLANFindEnableExchange()** if it wants to be notified as incoming **NetLANFind** requests are received. This is what provides the capability for an application to decide whether a response will be sent or not. When the **NetLANFindExchangeCallback** function is called by the DME, the developer can indicate whether a response should be sent or not by setting **NetLANFindExchangeCallbackOutArgs**.*bRespondToSender* to TRUE or FALSE.

Canceling a NetLANFind

To abort any pending **NetLANFind()** requests you must call **NetLANFindCancel()**.

Debugging When NetLANFind Replies are NOT Occurring

It is possible not to get replies due to the following reasons:

1. Network interference. Since the data is transmitted unreliable, there always exists the possibility that the data was lost. This is especially true on a wireless LAN environment. The application may need to call **NetLANFind()** multiple times if no replies occur.
2. Both sender and recipient must have the **NetSetDefaultLANFindParams**.*ConnectivityType* set to **NetConnectivityLAN**.
3. No session type exists on the LAN at the moment.
4. The recipient didn't want the reply to occur.
5. The filter specified same *ApplicationID* and same platform – however one differs.
6. Once **NetLANFind** has been called an application needs to call **NetUpdate**.
7. The **LANFind** request was canceled.

Sample

Please refer to the samples/LanGameFind for sample source code using **NetLANFind**.

LAN Messaging

LAN Messaging allows the application developer to quickly send messages to other peers on the same LAN. Messages are sent connectionless and unreliably, since the intent of the API is to allow users to quickly send messages to other peers on the LAN. Messages can either be sent in text format or as Raw data.

The Steps for Implementing LAN Messaging

1. Enable LAN Messaging and set LAN Messaging callbacks used for receiving LAN messages.
2. Find Peers using LAN Find API. See NetLANFind: Locating Integrated Servers and PeerToPeer Hosts section for more details.
3. Extract **NetAddress** from **NetLANFind**.
4. **NetSendAppMessage()** to **NetAddress**.

API Description

Structures

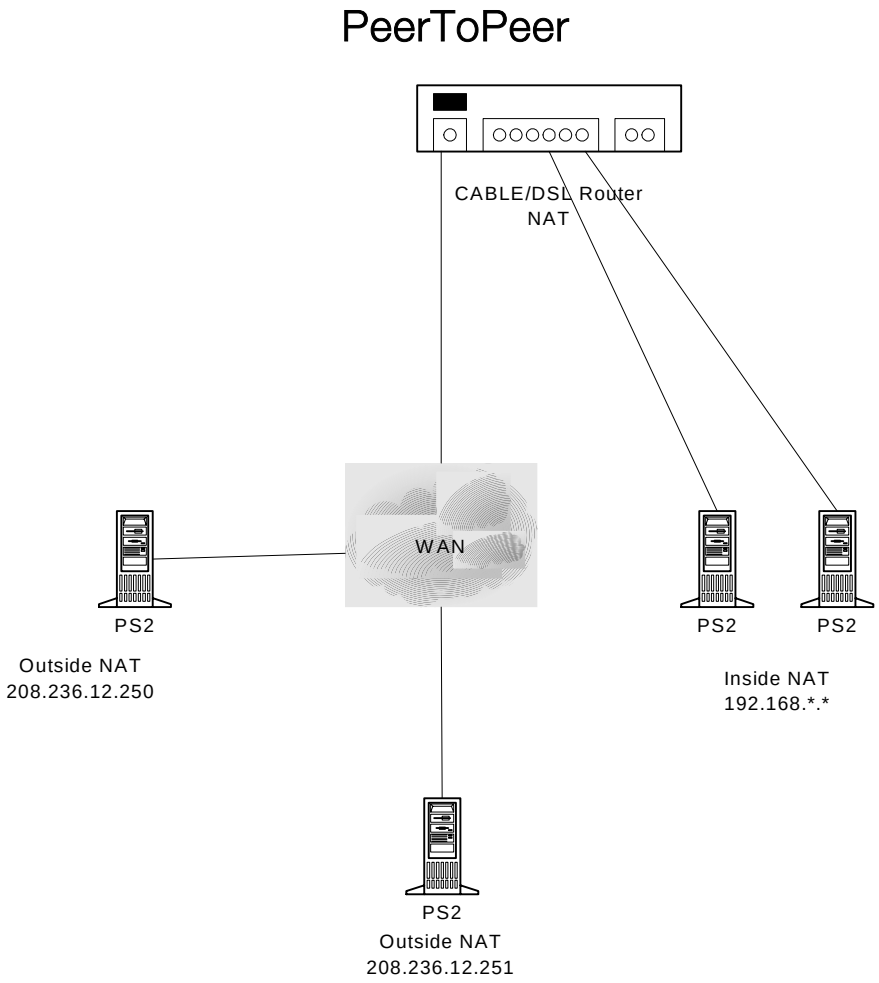
- **NetLANSendTextMessageInParams**/**NetLANSendRawMessageInParams** – input parameters for Send message functionality.
- **NetLanTextMessageCallbackParams**/ **NetLanRawMessageCallbackParams** – structure containing information received during a LAN message callback.
- **NetEnableLanMessagingInParams** – structure the user fills out specifying callback functions to be called when messages are received.

Functions

- **NetEnableLanMessaging()** – Enables LAN Messaging
- **NetDisableLanMessaging()** – Disables LAN Messaging
- **NetLANSetDefaultSendTextMessageInParams()**, **NetLANSetDefaultSendRawMessageInParams()**, **NetLanSetDefaultEnableLanMessagingInParams()** – sets defaults for parameters of LAN messaging API.
- **NetLANSendTextMessage()**, **NetLANSendRawMessage()** – Sends a message to an unconnected peer.

Network Address Translation Routers (NATs) and Connectionless UDP

Figure 6



Under normal circumstances, a connectionless UDP will not function behind Cable/DSL routers commonly used at homes. With the SCE-RT SDK, client applications can negotiate communications with multiple clients behind different NATs. This functionality is handled transparently within the SCE-RT SDK and the application will not have to worry about connecting to clients behind these types of routers. In order to support PeerToPeer games behind a NAT, an application must integrate with the following SCE-RT SDK Components: DME Client, Medius Client and MGCL.

Memory Management With the DME

Memory management within the DME utilizes the standard systems calls of malloc and free. If an application is performing its own type of memory management, then it should register the memory allocation callbacks within the DME. All memory allocation and de-allocation within the DME will use these application-specific callbacks. See **NetSetDefaultMemoryCallbackParams()** and **NetRegisterMemoryCallbacks()**.

Global Timebase

The Session Master is responsible for maintaining the global timebase and keeps all clients connected to a world in sync with its own world time. **NetGetTime** can be used to determine the global time. Global time is only valid for a client after it has completed **NetJoin**. As a result, the global time will be set to the machine's local time before joining (its time since DME initialization).

After **NetJoin** is complete the global timebase can fluctuate for up to thirty (30) seconds. The most notable jump in the global timebase will occur immediately after the Session Master accepts a client into the game. Synchronization of the timebase can be delayed if clients are in the process of loading data and the clients are not servicing **NetUpdate()** every frame.

The clients periodically receive updates to the global timebase throughout the game. Typically, once the global timebase has settled down, it doesn't fluctuate. Any updates after synchronization that will cause drastic modifications are not used to modify the global timebase. If these types of updates are continually received, it generally indicates increased latency in the communication between client and Server and may cause the global timebase to fluctuate.

NetGetTime will always return the last global time as set by **NetUpdate**. Subsequent calls to **NetUpdate** will keep the global time base up to date.

Hostname Resolution via the DME

The DME provides the ability to lookup hostnames and convert those names to an IP Address. You can only make one DNS request a time. Once a DNS request is made, the application needs to wait for a response or wait until the callback is invoked with an error.

```
unsigned int gbWaitingForDNSResponse=1;

//callback invoked upon a dns response

void myDNSResponseCallback(NetTypeResponse *pLookupResponse)

{

   gbWaitingForDNSResponse=0;

   if(pLookupResponse.ErrorCode != NetErrorNone)

   { //query was successful, save off IPAddress for use in NetConnect,MediusConnect

     ..

     return;

   }

    else

    {

        //Possible Error Codes are NetErrorServerError, NetErrorHostnameError, NetErrorLookupError

       printf("myDNSResponseCallback: Hostname lookup failed, result:%d\n",pLookupResponse.ErrorCode);

    }

}


{

..

..

// hostname to lookup

char hostname[]="www.scea.com";

NetTypeLookupParams stNetLookupParams;

NetErrorCode eNetError;

//DME must be initialized

..

..

eNetError = NetSetDefaultLookupParams(&stNetLookupParams);

if(eNetError != NetErrorNone){

   printf("NetSetDefaultLookupParams failed with result:%d\n",eNetError);

}

strncpy( stNetLookupParams.szHostName,hostname,NET_MAX_HOSTNAME_LENGTH);

//szDNSServerIP should be available to the application once the network interface has been brought up.

strncpy( stNetLookupParams.szServerIP,szDNSServerIP, NET_MAX_IP_LENGTH);

stNetLookupParams.pfLookupResponse = myDnsResponseCallback;

eNetError = NetGetHostByName(&stNetLookupParams);

eNetError = NetSetDefaultLookupParams(&stNetLookupParams);
```

```
    if(eNetError != NetErrorNone){
       printf("NetGetHostByName failed with result:%d\n",eNetError);
    }
    //Call NetUpdate until the DNS query comes backs.
    while(gbWaitingForDNSResponse)
    {
       eNetError= NetUpdate();
       if(eNetError != NetErrorNone)
        {
                printf("NetUpdate failed, result:%d\n",eNetError);
                return;
        }
}
```

This page intentionally left blank.