Multicast Dissemination Protocol (MDP) Developer's Guide

Brian Adamson
Newlink Global Engineering Corporation
<adamson@itd.nrl.navy.mil>

Joe Macker
Naval Research Laboratory
<macker@itd.nrl.navy.mil>
Table of Contents

1.0 Introduction ..................................................... 5

2.0 MDP Application Programming Interface (API) Overview ........ 5

3.0 MDP API Initialization ............................................. 6
  3.1 Use of MdpInit() ................................................. 6
    Function MdpInit() .............................................. 7
    Function MdpGetLocalNodeId() .................................. 7
    Function MdpSetLocalNodeId() .................................. 8
  3.2 Protocol Engine Callback Functions ................................ 8
    Function MdpSetTimerInstallerCallback() ....................... 9
    TimerInstallCallback Description ................................ 9
    Function MdpDoTimers() .......................................... 11
    Function MdpSetSocketInstallerCallback() ...................... 11
    SocketInstallCallback Description ............................... 12
    Function MdpReadSocket() ....................................... 13
    Function MdpSetNotifyCallback() ............................... 14
    MdpNotifyCallback Description .................................. 14
    MdpNotifyCallback Guidelines .................................... 16
    MdpNotifyCallback Notification Events .......................... 17

4.0 MDP API Variable Types ........................................... 20
  4.1 MDP API Variable Type Definitions ............................ 20
  4.2 MDP Defaults and Constants ................................... 21

5.0 MdpSession Management Functions .................................. 22
  5.1 MdpSession Creation and Destruction ......................... 22
    Function MdpNewSession() ...................................... 22
    Function MdpDeleteSession() .................................... 23
  5.2 MdpSession General Control ................................... 23
    Function MdpSessionSetTxRate() ................................ 24
    Function MdpSessionSetStatusReporting() ....................... 25
    Function MdpSessionSetTOS() .................................... 25
    Function MdpSessionSetMulticastInterfaceAddress() ........... 27
    Function MdpSessionSetMulticastLoopback() .................... 27
    Function MdpSessionSetRobustFactor() .......................... 28
  5.3 MdpSession Server Operation .................................. 29
Function MdpSessionOpenServer() .................................. 30
Function MdpSessionCloseServer() .................................. 32
Function MdpSessionQueueTxFile() .................................. 33
Function MdpSessionQueueTxDataObject() .......................... 34
Function MdpSessionRemoveTxObject() .............................. 36
Function MdpSessionSetCongestionControl() ....................... 36
Function MdpSessionSetAutoParity() ................................. 37
Function MdpSessionSetServerEmcon() .............................. 37
Function MdpSessionAddAckingClient() ............................. 38
Function MdpSessionRemoveAckingClient() ......................... 39
Function MdpSessionSetTxCacheDepth() ............................ 39
Function MdpSessionServerSetGrttEstimate() ..................... 40
Function MdpSessionServerGetGrttEstimate() ..................... 41
Function MdpSessionServerSetGrttMax() ........................... 41
Function MdpSessionSetGrttProbeInterval() ....................... 41
Function MdpSessionSetGrttProbing() .............................. 42
Function MdpSessionSetBaseObjectTransportId() ................. 43

5.4 MdpSession Client Operation ...................................... 43
Function MdpSessionOpenClient() .................................. 45
Function MdpSessionCloseClient() .................................. 46
Function MdpSessionAbortRxObject() .............................. 46
Function MdpSessionGetNodeAddress() ............................ 47
Function MdpSessionGetNodeGrttEstimate() ....................... 48
Function MdpSessionGetNodeNextRxObject() ...................... 48
Function MdpSessionDeactivateNode() ............................. 49
Function MdpSessionDeleteNode() ................................. 49
Function MdpSessionSetClientAcking() ............................. 50
Function MdpSessionSetClientUnicastNacks() ..................... 50
Function MdpSessionSetClientNackingMode() ..................... 51
Function MdpSessionSetClientStreamIntegrity() ................. 52
Function MdpSessionSetClientMulticastAcks() ................... 53
Function MdpSessionSetArchiveDirectory() ...................... 53
Function MdpSessionSetArchiveMode() ............................ 54
Function MdpSessionSetClientEmcon() ............................ 55

6.0 MdpObject Management Functions ............................... 56
Function MdpObjectGetType() ....................................... 57
Function MdpObjectGetInfo() ...................................... 57
Function MdpObjectGetSize() ...................................... 58
Function MdpObjectGetRecvBytes() ................................ 58
Function MdpObjectGetTransportId() .......................... 59
Function MdpObjectGetSourceNodeId() .......................... 59
Function MdpObjectSetNackingMode() .......................... 60
Function MdpObjectSetData() ................................. 60
Function MdpObjectGetData() ................................. 61
Function MdpObjectGetFileName() ............................ 62

7.0 Miscellaneous Functions ........................................... 62
Function MdpSessionSetRecvDropRate() .......................... 62
Function MdpSessionSetSendDropRate() .......................... 63
1.0 Introduction
This document serves as a guide for developing reliable multicast applications using the MDP Software Development Kit (SDK). The SDK consists of object libraries and header files which allow developer's to create applications using the MDP reliable multicast protocol.

2.0 MDP Application Programming Interface (API) Overview
MDP is a transport protocol which allows data to be reliably transmitted from a source (server) to a set of receivers. Normally, as its name implies, MDP is intended to use network multicast transport, but it should be noted that the destination of data transmitted by an MDP server can also be a unicast (point-to-point) network address. In some cases, MDP's rate-controlled, selective negative acknowledgment protocol design or other special features may offer benefits for unicast transport of data. However, it is MDP's utilization of multicast service which allows the creation of very efficient one-to-many or many-to-many reliable data communication applications and this is how it is anticipated that MDP will usually be utilized. For more general information on multicast, please refer to [1].

To best understand how the MDP API is intended to be used, it is important to first establish some terminology which reflects the design of the MDP protocol engine and the corresponding API procedures. At the highest level, a call to the MdpInit() function returns a handle to an instance of the MDP API. Using this MdpInstanceHandle, the developer can then make other API calls to further init the API to operate within the context of the developer's application and go on to create and control instances of MDP reliable multicast transport sessions.

MDP is intended to be used as a transport protocol and so the semantics of the API design are centered around the notion of the transport of data from a server (source) node to one or more client (receiver) nodes. Note that a single node may act as both server and client, and that multiple servers may co-exist in the same MDP communications space allowing for the creation of many-to-many communication applications. The "communications space" in which the MDP protocol operates is defined by an IP network address (usually a Class D multicast address) and a system User Datagram Protocol (UDP) port number. It is assumed that the MDP developer has an a priori understanding of basic network communications programming, and for further description of the principles involved there, please refer to [2]. This "communications space" which defines a single instantiation of the MDP transport protocol is referred to in the MDP API as an MdpSession. Note the MDP API permits the existence of multiple MdpSession instances so that applications requiring data exchange on multiple, different multicast addresses or the creation of unicast/multicast or multicast/multicast gateway applications may be created.

Within the context of a single MdpSession, multiple participants, referred to as MdpNodes, exchange data using the reliability mechanisms of the underlying MDP protocol. Some (or all) MdpNodes within an MdpSession, may act as servers (sources) while others may act as clients (receivers). Data is transmitted from a server to the set of clients as a serialized stream of MdpObjects. Each MdpObject consists of a file or static block of memory-resident data which the server application has queued for transmission. Each MdpObject may also
optionally have a small amount of “out-of-band” MdpInfo associated with it to aid application operation. Potential uses of the MdpInfo attachment will be discussed in more detail later in this document.

The use of the API can be summarized with the following steps:

1) Initialize the MDP library and install appropriate callback functions for protocol engine timing, packet input/output, and protocol event notification. *(Note: The MDP library is designed to use whatever timing facilities and receive packet notification methodology the application designer specifies.)*

2) Create on or more MDP sessions with appropriate multicast (or unicast) destination addresses and port numbers.

3) Set any parameters for the newly created sessions which need to differ from default values and need to be set prior to session startup.

4) “Open” the session as an MdpClient and/or MdpServer. *(After opening a session as a server, objects may be queued for transmission at any time)*

5) Enter the application's primary event loop dispatching appropriate events to the MdpDoTimers() and MdpReadSockets() API calls *(These are explained in detail later)*.

6) Handle notifications received from the MDP protocol engine with regards to newly received objects or to queue additional objects for transmission. API calls are available to retrieve information concerning MdpObjects received or queued for transmission.

3.0 MDP API Initialization

The MDP library must be properly initialized before any of the other API functions can be used. Also, as part of initialization, appropriate callback functions must be installed to facilitate the operation of the MDP protocol engine. These callbacks include application-defined functions for system timer usage and receive packet notification.

3.1 Use of MdpInit()

The first step is to place a call to the function MdpInit() to create and partially initialize and instance of the MDP API. This function has two optional parameters to allow the application to set a local ASCII MdpNode name identifier and a 32-bit identification number (MdpNodeId) for the local station. The MdpNodeId should be unique among all nodes in an MdpSession. The default (NULL) values provided for these parameters will cause the MdpInit() function to attempt to learn the local default IP address to use as the MdpNodeId and resolve that address and determine the login name to establish a default local name in the form of “user@hostname”.
**Function MdpInit()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpInstanceHandle MdpInit(const char* localName = (const char*)NULL,
                          MdpNodeId localId = MDP_NULL_NODE,
                          MdpError* error = NULL);
```

**DESCRIPTION**

The MdpInit() function creates and partially initializes an instance of the MDP API. This function should be only called once and must be called before any other MDP API functions are used. The `localName` parameter should be a pointer to a NULL-terminated string with a maximum length of MDP_NODE_NAME_MAX characters. Using the MdpInit() function in its default form (no arguments) will automatically pick a default `MdpNodeId` (using the local host’s default IP address) and `MdpNode` name. The MDP library does this by making a call to `getlogin()` (or `cuserid()` in some cases) and `gethostname()` to retrieve the user's name and the machines name. The machine’s name is resolved to an IP address with a call to `gethostbyname()` and the 32-bit IP address (converted to host byte order) is used as the local `MdpNodeId`. This technique usually produces a suitable default `MdpNodeId` assignment method resulting in unique node identification. Note that in the case of the tkMdp/winMdp/mdp example applications, the use of a *a priori* client positive acknowledgment list depends upon the names/addresses in the list resolving to the address picked by the client as its MdpNodeId. The `localId` parameter can be used to override this default method of `MdpNodeId` assignment. The optional `error` parameter allows the application to receive and error code if the call to MdpInit() is unsuccessful.

**RETURN VALUES**

MdpInit() will return a handle to the instance of the MDP API created of type `MdpInstanceHandle`. The application-provided `error` parameter will contain a value of `MDP_ERROR_NONE` if successful. A return value of `MDP_NULL_INSTANCE` and other `error` codes (TBD) are passed in the `error` parameter upon failure.

**Function MdpGetLocalNodeId()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpNodeId MdpGetLocalNodeId(MdpInstanceHandle instance);
```

**DESCRIPTION**
The MdpGetLocalNodeId() function retrieves the local *MdpNodeId* for the MDP instantiation designated by the *instance* parameter.

**RETURN VALUES**

MdpGetLocalNodeId() will return the local MdpNodeId upon success. A value of MDP_NULL_NODE is returned if the *instance* parameter is NULL (invalid).

**Function MdpSetLocalNodeId()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpError MdpSetLocalNodeId(MdpInstanceHandle instance,
                          MdpNodeId nodeId);
```

**DESCRIPTION**

The MdpSetLocalNodeId() function sets the local *MdpNodeId* for the MDP instantiation designated by the *instance* parameter. The *nodeId* parameter indicates the value to be used as the local hosts node identifier. A value of MDP_NULL_NODE is may not be used as a node identifier.

**RETURN VALUES**

MdpGetLocalNodeId() will return a value of MDP_ERROR_NONE upon success. A value of MDP_ERROR_INSTANCE_INVALID or MDP_ERROR_NODE_INVALID may be returned upon error.

### 3.2 Protocol Engine Callback Functions

After a successful call to MdpInit(), the next step the application should take is installing appropriate callback functions for the returned *MdpInstanceHandle* to provide timing and receive data event notification for the MDP protocol engine to function and to allow the MDP protocol to alert the application of important events (e.g. received object completion, etc). Different callback functions can be installed for different MDP API instances. The use of callback functions for these core aspects of the current MDP SDK allows MDP to be very portable to different operating systems and environments. Wherever possible, the MDP library attempts to allow the application to control memory allocation and interface with the operating system. (It is possible this will be further expanded in the future). While this “open-ended” approach may seem complex, it allows for flexibility as the MDP API is further refined. At some point in the future, it is possible that a higher level (less complex) API may be created (and optimized) for specific operating systems and environments.

There are three primary callbacks which need to be installed. One is for the maintenance of a
single operating system timer which the MDP protocol engine will use for protocol timing purposes. Although the MDP protocol implements multiple timeout events for operation, it manages these timeouts with a single (application-provided) system timer. The second callback function is used by MDP to notify the application of network socket file descriptor(s) MDP has opened (or closed) so that the application can install (or remove) appropriate “data-ready-to-be-read” monitors (e.g. for use with a select() system call or other asynchronous I/O notification mechanism). The third callback is a function pointer to an MdpNotifyFunc which is a user-defined function which MDP will call to notify the application of significant protocol events or errors.

Function MdpSetTimerInstallerCallback()

SYNOPSIS
#include <mdpApi.h>

void MdpSetTimerInstallerCallback(
    MdpInstanceHandle instance,
    TimerInstallCallback* timerInstaller,
    void* userData);

DESCRIPTION
This function is used to install an application-provided TimerInstallCallback function for the MDP protocol engine to use. The TimerInstallCallback is used by MDP to request the application to install, modify, or remove the system timer which MDPv2 uses for protocol timing.

The instance parameter must be a value obtained by the application from a previous call to MdpInit().

The timerInstaller parameter is a pointer to the TimerInstallCallback the application wishes MDP to use.

The value passed by the userData parameter is stored by MDP and MDP will later pass a pointer to this stored value to the provided TimerInstallCallback for the application to use. This userData might be useful for an application to retrieve state associated with a specific timer which has been installed or for some other application-defined use. The usage is described below in more detail in the detailed description of the TimerInstallCallback function the application must provide.

RETURN VALUE
The MdpSetTimerInstallerCallback() function returns no value.

TimerInstallCallback Description

The prototype for this application-provided callback function is given by:

typedef bool (TimerInstallCallback) (
ProtocolTimerInstallCmd cmd,
void** timerId,
double delay,
void* timerData,
void** userData);

The **cmd** parameter instructs the callback function on what action is to be taken. Possible values for the **cmd** parameter include PROTOCOL_TIMER_INSTALL, PROTOCOL_TIMER_MODIFY, and PROTOCOL_TIMER_REMOVE. These values correspond to whether the system timer used by MDPv2 is to be newly installed, its timeout interval to be changed, or if the timer is to be removed (cancelled). *(Note: MDPv2 assumes the application-installed system timer is a "one-shot" timer which needs re-installation after it fires.)*

The **timerId** parameter is currently provided for convenience. One purpose is that the application may use it to store the value of a system-specific identifier for the timer which it has installed for MDP. On subsequent calls to **modify** or **remove** the timer, MDP will provide this parameter with the value previously set by the application.

**IMPORTANT NOTE:** It is recommended that new applications built with the MDP API do not use the **timerId** parameter. The **userData** parameter (described below) can be used for whatever purpose the **timerId** parameter might be used and future versions of the API will likely eliminate this parameter in this callback function.

The **delay** parameter is used by MDP to inform the application of the timeout interval value the application-installed system timer should use. The value is given in time units of seconds. This parameter is not applicable for the case of **cmd** = PROTOCOL_TIMER_REMOVE. Note that the value of **delay** can possibly equal zero.

The **timerData** parameter is a value used internally by the MDP protocol engine. This value is passed to the **TimerInstallCallback** because when the application-installed system timer fires, the application must call the MdpDoTimers() API function passing this **timerData** value as the parameter of the MdpDoTimers() function. In future versions of the API, it may be possible for multiple instances the MDP API to co-exist (primarily for simulation environments) so it is important to properly associate the **timerData** value provided in calls to the **TimerInstallCallback** function with the resultant system timer installed. The MdpDoTimers() function is used to dispatch MDP protocol timeout events to the underlying protocol engine.

The **userData** parameter of the **TimerInstallCallback** supplies a pointer to the value stored when the MdpSetTimerInstallCallback() function was called. Note that since a pointer to this value is provided, the application may currently change this value. An example use of **userData** is for the application to store a pointer to state that it needs to properly install and maintain system timer resources.

**IMPORTANT NOTE:** It is recommended that new applications built with the MDP API are
structured so as not to change the value of the userData originally provided by the application. For code safety reasons, it is highly likely that future versions of this callback will provide the userData value directly and the application will be unable to change it other than through a call to MdpSetTimerInstallCallback(). (Moral of the story: Providing pointers to the guts of the protocol engine was a bad idea.)

The TimerInstallCallback should return a value of true on success and a value of false on failure.

**Function MdpDoTimers()**

**SYNOPSIS**

```c
#include <mdpApi.h>

double MdpDoTimers(void* timerData);
```

**DESCRIPTION**

The MdpDoTimers() function is used by the application to dispatch timeout events to the MDP protocol engine. The timerData value passed to the MdpDoTimers() function should equal the value passed to the application when the corresponding timer was “installed” or “modified”. A call to MdpDoTimers() causes the MDP protocol engine to take whatever action is appropriate as a result of timeout events. These actions include sending data, installing other timeouts, and notifying the application (via the MdpNotifyCallback) of important protocol events or errors. Note that the MDP protocol engine assumes that the system timer which has fired (resulting in the current timeout event) was a one shot timer and MDP will install a new system timer as needed. If the application environment is not using one-shot timers, the application should take measures to remove or cancel the timer prior to placing the call to MdpDoTimers().

**RETURN VALUES**

The MdpDoTimers() call returns a floating point value equal to the time interval until the next MDP system timer timeout event. A value of -1.0 is returned if no timeout event is pending after the call to MdpDoTimers().

**Function MdpSetSocketInstallerCallback()**

**SYNOPSIS**

```c
#include <mdpApi.h>

void MdpSetSocketInstallerCallback(
    MdpInstanceHandle instance,
    SocketInstallCallback* socketInstaller,
    void* userData);
```
DESCRIPTION

This function installs a SocketInstallCallback which is used by MDP to request the host application to asynchronously "monitor" a socket file descriptor for received data to be read. This allows an application to use whatever socket/file descriptor monitoring/notification methodology that is appropriate for the given application's implementation architecture. The application-provided SocketInstallCallback is called whenever MDP "opens" or "closes" a UDP socket.

The instance parameter must be a value obtained by the application from a previous call to MdpInit(). The socketInstaller parameter is a pointer to the application-provided SocketInstallCallback function. The userData parameter allows the application to store an application-specific data value (e.g. pointer) which is passed back to the application when the SocketInstallCallback is called by MDP.

RETURN VALUE

The MdpSetSocketInstallCallback() function returns no value.

IMPORTANT NOTE: As of version 1.7a7 of the MDP code, the SocketInstallCallback is now required for Win32 operation. Win32 has a few different mechanisms which could be leveraged including a Berkeley Sockets-like "select()" call, Window messages (e.g. using the WSAAsyncSelect()" call - this is typical of many Win32 applications), or Events (e.g. where "WaitForMultipleEvents()" is used similar to a "select()" call). This API leaves all of these options open to the programmer. In the future, it is hoped we will be able to provide more tutorial information on use of the MDP API on different operating systems for different applications. For now, refer to the example application code we provide.

SocketInstallCallback Description

The SocketInstallCallback is defined by the following function prototype:

typedef bool (SocketInstallCallback)(
    SocketInstallCmd cmd,
    int fd,
    void* socketData,
    void** userData);

The cmd parameter is used by MDP to indicate what action the application-provided SocketInstallCallback should take. Possible cmd parameter values include SOCKET_INSTALL and SOCKET_REMOVE. The SocketInstallCallback is called with cmd equal to SOCKET_INSTALL after MDP has opened a socket for which it would like notification for reading received data. The callback function will be called and the cmd parameter will have value of SOCKET_REMOVE when the corresponding socket has been closed. The purpose is that the application begin monitoring the socket for possible received data when the SOCKET_INSTALL "command" is issued. The host application must call the MDP API MdpReadSocket() function when data is available for reading on a given socket file descriptor. This will cause the MDP protocol engine to read and process the received data,
taking whatever action is necessary.

The `fd` parameter passes the integer file descriptor value for the network socket for which MDP is requesting monitoring to the `SocketInstallCallback`. Example uses of the file descriptor include adding the file descriptor to a file descriptor set (FD_SET) for use with the `select()` call. Other application synchronous input/output (I/O) notification mechanisms can make similar use of socket file descriptors.

The value passed in the `socketData` parameter is for internal use by the MDP protocol engine. When the application’s synchronous I/O notification mechanism detects that there is received data ready to be read on one of the MDP-installed socket file descriptors, it should call the `MdpReadSocket()` API function passing the corresponding value of the `socketData`. Note that each socket opened by MDP will use its own unique value of `socketData` so it is important that an application opening multiple `MdpSessions` (which will in turn open multiple sockets) maintain the relationship between a given socket file descriptor and the corresponding MDP-provided `socketData` value.

The `userData` parameter passes the a pointer to the stored value the application specified as user data in its initialization call to `MdpSetSocketInstallCallback()`. Note that since a pointer to this value is provided, the application may currently change this value. An example use of `userData` is for the application to store a pointer to state that it needs in order to properly install and maintain system timer resources.

**IMPORTANT NOTE:** It is recommended that new applications built with the MDP API are structured so as to not change the value of the `userData` originally provided by the application. For code safety reasons, it is highly likely that future versions of this callback will provide the `userData` value directly and the application will be unable to change it other than through a call to `MdpSetSocketInstallCallback()`. (Moral of the story: Providing pointers to the guts of the protocol engine was a bad idea.)

The application-provided `SocketInstallCallback` should return a value of `true` upon success and a value of `false` upon failure.

**Function MdpReadSocket()**

**SYNOPSIS**

```c
#include <mdpApi.h>

void MdpReadSocket(void* socketData);
```

**DESCRIPTION**

The `MdpReadSocket()` function is used by the application to notify MDP of received data ready to be read on a socket which MDP has previously opened and, through a call to the application’s `SocketInstallCallback`, installed for input notification. The `socketData` value passed to the `MdpReadSocket()` function must equal the value passed to the application
when the corresponding socket was “installed”. A call to MdpReadSocket() causes the MDP protocol engine to read the received message from the socket and process it, taking whatever action is appropriate. These actions include sending data in response, installing timeouts, and notifying the application (via the MdpNotifyCallback) of important protocol events or errors.

RETURN VALUES
The MdpReadSocket() returns no value.

Function MdpSetNotifyCallback()

SYNOPSIS
#include <mdpApi.h>

void MdpSetNotifyCallback(
    MdpNotifyCallback* notifyFunc,
    void* userData);

DESCRIPTION
This function allows the application to install a MdpNotifyCallback function which will be called when the MDP protocol engine needs to inform the application of important protocol events or errors. The MdpNotifyCallback (described in detail elsewhere) is used by MDPv2 to “notify” the host application of protocol events such as updated MdpSession or MdpObject (transmit or receive) status. This allows the host application to update user interface displays as needed, kick off post-processing of received data, be prompted for additional data to transmit, and be notified of errors.

The notifyFunc parameter is a function pointer to an application-provided MdpNotifyCallback function.

The userData parameter is an application—provided value which is stored by MDP and later passed back to the application when the MdpNotifyCallback function is called. This parameter allows the user to retrieve information necessary to perform whatever actions the application requires upon notification of significant MDP protocol events.

RETURN VALUES
MdpSetNotifyCallback() returns no values.

MdpNotifyCallback Description

The MDP protocol engine calls the application-provided MdpNotifyCallback function when different important protocol events or errors occur. This allows the user to take different actions such as to allow the application to process a received file or data object, enqueue additional file or data objects for transmission, receive notification of transmit or receive progress, get updated status on other nodes within the MDP multicast session, etc. The
**MdpNotifyCallback** function is described with the following prototype:

```c
typedef bool (MdpNotifyCallback)(
    MdpNotifyCode notifyCode,
    MdpSessionHandle sessionHandle,
    MdpNodeId nodeId,
    MdpObjectHandle objectHandle,
    MdpError errorCode,
    void* userData);
```

The `notifyCode` parameter indicates the nature of the MDP protocol event which has occurred. There are a number of possible events which cause the **MdpNotifyCallback** to be invoked. Possible values of the `notifyCode` include:

- MDP_NOTIFY_ERROR,
- MDP_NOTIFY_TX_OBJECT_START,
- MDP_NOTIFY_TX_OBJECT_FIRST_PASS,
- MDP_NOTIFY_TX_OBJECT_ACK_COMPLETE,
- MDP_NOTIFY_TX_OBJECT_FINISHED,
- MDP_NOTIFY_TX_QUEUE_EMPTY,
- MDP_NOTIFY_SERVER_CLOSED,
- MDP_NOTIFY_RX_OBJECT_START,
- MDP_NOTIFY_RX_OBJECT_UPDATE,
- MDP_NOTIFY_RX_OBJECT_COMPLETE,
- MDP_NOTIFY_REMOTE_SERVER_INACTIVE,
- MDP_NOTIFY_OBJECT_DELETE.

Each of these possible **MdpNotifyCallback** events will be described in detail below.

The `sessionHandle` parameter identifies the **MdpSession** for which the event is applicable. This is significant for applications which may manage multiple **MdpSessions**. This `sessionHandle` parameter can be used in other MDP API calls to manipulate the state of the corresponding **MdpSession**.

The `nodeId` parameter identifies the **MdpNodeId** to which the notification is applicable. If the notification is with regards to an **MdpObject** being received, the `nodeId` identifies the source of the **MdpObject**. When the **MdpObject** is an object being transmitted, the value of `nodeId` will be MDP_NULL_NODE. Also, the `nodeId` will be a value of MDP_NULL_NODE for notifications not directly related to remote **MdpServer** status changes or **MdpObject** reception.

The `objectHandle` parameter indicates the transmit or receive file or data **MdpObject** for which the notification event is applicable. It is important to note that the value of the `objectHandle` parameter provided here is only valid during execution of the **MdpNotifyCallback** and the value should not be saved for future reference outside the specific instance of the current **MdpNotifyCallback** function call. **MdpObjectHandles** apply only to the identification of data or file objects during the course of their transport by the MDP protocol. However, note that **MdpObjectHandle** values returned with a call to MdpSessionQueueTxFile() or MdpSessionQueueTxData() are valid until the receipt of a MDP_NOTIFY_TX_OBJECT_FINISHED notification event (or a fatal error...
event) for that MdpObject. Specific examples of the use of the objectHandle parameter will be provided in the description of the individual MdpNotifyCallback event types.

The errorCode parameter is used to provide an indication of the error type when the notifyCode value is equal to MDP_NOTIFY_ERROR. As in other events the sessionHandle and objectHandle parameters will provide additional information as appropriate to the errorCode type. Possible values for the errorCode currently include:

MDP_ERROR_FILE_OPEN

This indicates that the MDP protocol engine was unable to open a file for transmission (or possibly reception).

MDP_ERROR_FILE_EMPTY

This indicates that a file enqueued for transmission had zero contents.

MDP_ERROR_FILE_LOCKED

This indicates file could not be opened because it was locked.

MDP_ERROR

This generic error condition occurs when an MdpDataObject enqueued for transmission has a NULL data pointer value.

MDP_ERROR_MEMORY

This indicates MDP was unable to allocate memory for maintaining state on a transmit or receive object.

The userData parameter contains the value the application specified when installing the MdpNotifyCallback with the MdpSetNotifyCallback() function. This userData value is for the application's private use. For example, a pointer value might be stored so the application can retrieve context information when the MdpNotifyCallback is called.

**MdpNotifyCallback Guidelines (Rules of Implementation)**

**IMPORTANT NOTE:** The user's MD Pv2 application should not spend much time in the MdpNotifyCallback function since it blocks the thread of execution of the MD Pv2 protocol. The user of this API should consider burying the MD Pv2 protocol engine in a separate thread or process if needed. Also note that the application-installed Notification Callback may be re-entrantly called from the result of other MDP API calls made within the Notification Callback routine.

A better long term approach will be to embed MD Pv2 protocol operation in its own thread and provide an API using an inter-thread (or inter-process) communications model. But meanwhile, to simplify cross-platform portability (and until we think of a better approach) we'll allow developers using MD Pv2 for applications (or simulations) to do that however they like. These callback mechanisms can currently be “buried” by developers however they see fit for the platforms for which they are concerned.
(This section is unfinished - RBA)

**MdpNotifyCallback Notification Events**

This section describes each of the different types of *MdpNotifyCallback* events. Each event is associated with a specific *notifyCode* value.

**MDP_NOTIFY_ERROR**

This notification event occurs when the MDP protocol engine encounters an error during the process of transmitting or receiving files and/or data. In the current implementation of the MDP API, there are only a few error events which result in this notification event. These primarily include indications when MDP is unable to open a file to read for transmission, an opened transmit file is found to have no content, or an MdpDataObject was enqueued for transmission with an invalid (NULL) pointer. Additionally, an errorCode equal to MDP_ERROR_MEMORY might be indicated when MDP is unable to allocate sufficient memory to maintain state on new transmit or receive objects. Currently, the application needs to take no action upon receipt of these notifications other than if the application wishes to provide logging of these events. In the future, the API error notification procedure may be further refined and additional error conditions added.

**MDP_NOTIFY_TX_OBJECT_START**

This notification event occurs when a file or data object which has previously been enqueued for transmission by the application (See the descriptions of MdpSessionQueueTxFile() or MdpSessionQueueTxData()) begins to be transmitted by the MDP protocol engine. Since the application is allowed to enqueue multiple objects for transmission and MDP's transmission rate is governed, this provides an indication to the application when an object's actual transport begins. The *sessionHandle* and *objectHandle* parameters identify the MdpSession and MdpObject for which the notification is relevant. Note that the *sessionHandle* and *objectHandle* values will be the same as those returned in the application's previous respective calls to MdpNewSession() and MdpSessionQueueTxFile() or MdpSessionQueueTxData().

**MDP_NOTIFY_TX_OBJECT_FIRST_PASS**

This notification event occurs after MDP has completed a “first pass” transmission of an entire MdpObject. This does not occur until all data (including an “auto parity” has been transmitted. This is intended primarily for use in applications where only silent clients (all clients are silent (“EMCON”) with no repair requests) are present. Since no clients will be requesting repair transmissions, it is not necessary for the MdpServer to maintain state for the transmitted MdpObject. Thus any resources used by that MdpObject can be freed after this notification occurs. Note that additional strategies can be implemented around this notification (e.g. the application may maintain a timeout to limit the amount of time that repairs are available for a transmitted object) or this notification may be useful in updating graphical user interfaces in applicable applications.

**MDP_NOTIFY_TX_OBJECT_ACK_COMPLETE**

The *MdpNotifyCallback* is called with this notification code when the server has completed the positive acknowledgement process for a given object. If no clients are providing positive
acknowledgement, this notification will occur immediately after the initial transmission of the indicated object. Otherwise, this notification occurs after the server has completed the positive acknowledgement cycle for an enqueued transmit object. This can mean either that the positive acknowledgement cycle fully succeeded, or that it timed out for one, some or all clients providing positive acknowledgement. The MDP API will be expanded in the future to allow the application to determine the positive acknowledgement status of individual clients.

**MDP_NOTIFY_TX_OBJECT_FINISHED**
This notification event occurs when the MDP server is completely finished with an object the application has previously enqueued for transmission. At this time, the client may wish to free resources associated with that object (e.g. close and/or delete an associated file or free associated memory). The “transmit cache depth” with which MDP has been configured determines how long MDP will keep state on transmit objects before MDP_NOTIFY_TX_OBJECT_FINISHED events occur. If a limited number of objects are enqueued for transmission, the MDP protocol engine will indefinitely keep state for the last objects queued within the constraints of the “transmit cache depth”. Note the application may prematurely de-queue (or abort) transmit objects with a call to the MdpSessionRemoveTxObject() API function. However, note that it is not safe to call this function from within the MdpNotifyCallback. Transmit objects may be safely removed at any other time. When the MDP_NOTIFY_TX_OBJECT_FINISHED notification event occurs, the indicated object will be automatically de-queued and the application should not attempt to refer to that object after this notification event has occurred. Note that the MDP_NOTIFY_OBJECT_DELETE notification will also occur after this MDP_NOTIFY_TX_OBJECT_FINISHED notification.

**MDP_NOTIFY_TX_QUEUE_EMPTY**
This notification event occurs once each time the MDP protocol engine determines it has no more objects pending transmission. The application can use this opportunity to enqueue additional objects for transmission. Otherwise, the application will need to provide its own mechanism for adding additional objects to the MdpSession transmit queue as needed. This notification is useful to allow the MDP protocol engine, which is transmitting data at a controlled rate, to control the flow of queuing objects for transmission.

**MDP_NOTIFY_SERVER_CLOSED**
This notification event occurs when local server operation has fully terminated after the application has initiated a “soft” close of MdpServer operation. The expected procedure is that the application will initiate server shutdown with the MdpSessionCloseServer() API function (with appropriate arguments). After this notification, is received, the application can free up state the MDP library has kept for the corresponding MdpSession with confidence that the protocol has robustly notified receivers of its intention to terminate server operation.

**MDP_NOTIFY_RX_OBJECT_START**
This notification event occurs when a new object is received at an MDP client node. This is useful for updating user interface displays or for the application to log information on receive status.

Additionally, in the case of MdpDataObjects, it is at this time that the application needs to provide memory space for the MDP protocol engine to store the received data. The amount
of memory required to store the new receive object can be obtained with a call to the
MdpObjectGetSize() API function. If the application does not provide the memory
space for the object (with a call to MdpObjectSetData()), the object will not be received.

**MDP_NOTIFY_RX_OBJECT_INFO**
This notification event occurs when the MDP_INFO message associated with the indicated
MdpObject is received. As described elsewhere, the MDP_INFO message can be used to
provide a small amount (less than or equal to the MDP server segment_size for the
MdpSession) timely context information for each object transmitted by an MDP server. Note
that the use of the MDP_INFO feature is optional so different applications may or may not
choose to utilize this protocol feature. The MDP_INFO received can be retrieved with a call
to the MdpObjectGetInfo() API function.

**MDP_NOTIFY_RX_OBJECT_UPDATE**
This notification event occurs whenever new data content is received for an MdpObject. This
allows the application to monitor the receive progress of individual objects. The
MdpObjectGetRecvBytes() API function can be used by the application to get an
indication of what portion of the object has been received up to that point in time.

**MDP_NOTIFY_RX_OBJECT_COMPLETE**
This notification event occurs when a object transmitted from a remote MDP server has been
completely received at the local node. This includes the optional MDP_INFO, if it is
available for the indicated object. At this point in time, the application may wish to process
the received object since it has been received in its entirety and/or potentially free resources
the application has allocated for that object. Note that the MDP_NOTIFY_OBJECT_DELETE
notification will also occur after this MDP_NOTIFY_RX_OBJECT_COMPLETE notification.

**MDP_NOTIFY_REMOTE_SERVER_INACTIVE**
This notification event occurs when an individual remote server MdpNode has been timed out
and determined to be inactive (i.e. no messages at all have been received from that server in
some sufficiently long period of time). The value of this inactivity timeout is
(2.0*ROBUST_FACTOR*GRTT) seconds. The default ROBUST_FACTOR is 20 (the API
contains a call to override this default) and the GRTT is the current estimate of group round
trip time advertised by the server in question. The application may use this notification as a
cue to make API calls which release resources (memory) used to maintain buffering and state
for the indicated MdpServer node. The applicable API calls include
MdpSessionDeactivateNode() which only frees buffering resources (the most significant
quantity of memory resources for servers) and MdpSessionDeleteNode() which frees all
resources and state associated with a remote server MdpNode. **Note that in the current MDP
implementation (version 1.7a9), the state for a remote MdpServer is dropped when it goes
inactive AND there are no pending receive objects requiring repair (This responsibility will
be placed in the application’s domain in a future release).**

**MDP_NOTIFY_OBJECT_DELETE**
This notification event occurs when the MDP protocol engine is “deleting” state for objects
which have either fully completed the reception or transmission process or have failed. This
notification allows the application to reclaim or free any resources (e.g. memory) which have
been allocated for the indicated object. Note that if the indicated object has not been
previously been associated with an MDP_TX_OBJECT_FINISHED or
MDP_RX_OBJECT_COMPLETE notification event, it is possible the transmission/reception
of the object has failed for some reason. Note that this can occur if the application
prematurely de-queues a transmit object or “closes” an MdpSession (as client, server, or both)
while objects are pending reception or transmission.

4.0 MDP API Variable Types
A number of different variable types and constants are defined for use with the MDP API
function calls. The constants should be referred to by name as the actual values may change
(usually for good reason) in future versions of the SDK.

4.1 MDP API Variable Type Definitions
The following variable types are defined for use with the MDP API function calls. These
types are defined in the “mdpApi.h” header file.

MdpSessionHandle

Values of type MdpSessionHandle are used to uniquely identify a specific MdpSession which
has been created. MdpSessions are associated with a specific destination IP address and port
number. There are a number of protocol parameters associated with an individual MdpSession
and the API provides functions to set these parameters. The MdpSessionHandle is used as an
argument to these API functions to identify the MdpSession to which the functions are
applied. A value of MDP_NULL_SESSION indicates an invalid (nonexistent) MdpSessionHandle.

MdpNodeId

Values of type MdpNodeId are used within the MDP API to uniquely identify nodes
participating within a given MdpSession. In general, all nodes participating within an
MdpSession should use unique MdpNodeId. The default behavior of the MDP protocol
ingine is to automatically assign the local MdpNodeId based on the host platform’s default IP
address. A value of MDP_NULL_NODE indicates an invalid (nonexistent) MdpSessionHandle.

MdpObjectHandle

Values of type MdpObjectHandle are used to uniquely identify a specific MdpObject which is
currently being transported (transmitted or received) by the MDP protocol engine. These
values. A value of MDP_NULL_OBJECT indicates an invalid (nonexistent) MdpSessionHandle.

MdpObjectTransportId

Values of type MdpObjectTransportId are used to identify the 32-bit number which uniquely
identifies an object associated with a server during transport (i.e. active transmission). It is
very important to note that these numbers are valid only during actual transport with respect
to a specific server node. In MDP, an object being actively transported within the context of
an *MdpSession* is uniquely identified among the session participants by concatenating the server’s (object source) *MdpNodeId* and the object’s *MdpObjectTransportId*. Within the MDP API, the intended use of the *MdpObjectTransportId* is for function calls which allow an application to establish sessions with alternative emission-controlled (EMCON) clients (silent client nodes) message transmission modes of operation. At the time of this writing, these API calls are not fully defined or documented. More information on this feature will be provided in a future iteration of this document.

**MdpObjectType**

Values of type MdpObjectType are used to identify the type of object being transported. The type information is generally used by the application at the receiver to provide proper handling of notification events related to a specific object. The application may wish to process files differently than received data or applications may restrict the type of objects they support. (E.g. the example “tkMdp” application is intended for file transfer/broadcast and only supports reception of objects of type MDP_OBJECT_FILE). The types of objects currently supported in MDP are MDP_OBJECT_FILE and MDP_OBJECT_DATA.

The type MDP_OBJECT_FILE is used for transmission of storage device based files where MDP servers can use the large capacity of the storage device as extended buffering for repair retransmissions. The MDP protocol engine currently handles transmit/receive file I/O directly for files. This is primarily because the API is currently designed for the protocol running in an “application space” environment (as opposed to within an operating system’s “kernel space”) and this allowed for efficient operation. In the future, it is possible that the MDP API may be more generalized and that application’s will be responsible for their own file-specific I/O and MDP will provide programming interfaces in a manner more analogous to that of a TCP socket.

The type MDP_OBJECT_DATA is used for “static” memory-resident objects. The API allows MDP applications (via pointers) to indicate blocks of memory for server transmission and client reception. The MDP protocol engine directly reads from and writes to the indicated blocks of memory which allows for selective repair across potentially large blocks of memory. This allows applications to transfer memory resident data with relatively little participation by the application outside of the MDP protocol. However, as with file I/O, it is possible that this API model may change in the future if the MDP protocol engine is embedded within an operating system kernel or in a separate thread or process. These application programming issues for reliable multicast are under continued investigation.

### 4.2 MDP Defaults and Constants

The following constants are defined for use in the MDP API. In some cases, these constants are default parameter values provided for information purposes.

1. **MDP_SESSION_NAME_MAX**: Maximum string length for an *MdpSession* name.
2. **MDP_NODE_NAME_MAX**: Maximum string length for an *MdpNode* name.
(3) MDP_DEFAULT_TTL
(4) MDP_DEFAULT_TX_RATE
(5) MDP_DEFAULT_SEGMENT_SIZE
(6) MDP_DEFAULT_BLOCK_SIZE
(7) MDP_DEFAULT_NPARITY
(8) MDP_DEFAULT_BUFFER_SIZE
(9) MDP_DEFAULT_TX_CACHE_COUNT_MIN
(10) MDP_DEFAULT_TX_CACHE_COUNT_MAX
(11) MDP_DEFAULT_TX_CACHE_SIZE_MAX

5.0 MdpSession Management Functions

The MDP API allows multiple MdpSessions to be created and opened to operate as server (sender) and/or client (receiver) participants. The general use of the API calls listed in this section (after the API has been properly initialized and appropriate callback functions installed as previously described) is to create one or more instances of an MdpSession (referenced by a variable of type MdpSessionHandle) and then open the session for protocol operation as a client and/or server node as required by the application.

5.1 MdpSession Creation and Destruction

MdpSession instances must be created before any protocol operation begins. Each MdpSession instance defines a specific destination address (including IP address and host port number) to which the local MdpNode will communicate.

**Function MdpNewSession()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpSessionHandle MdpNewSession(
    MdpInstance* instance,
    const char* address,
    unsigned short port,
    unsigned char ttl = MDP_DEFAULT_TTL,
    MdpError error = NULL);
```

**DESCRIPTION**

The MdpNewSession() function is used by the application to create a new MdpSession instance. Memory is allocated to maintain state for the session and default values are established for MdpSession parameters where applicable. A number of parameters for an MdpSession can be manipulated with function calls described below. The newly created MdpSession will not be active (no packets are sent or processed) until the MdpSession is
opened as a server and/or client participant through use of the MdpSessionOpenServer() or MdpSessionOpenClient() functions respectively.

The instance parameter must be value obtained by the application from a previous call to MdpInit(). The address parameter must be a pointer to a string containing a valid dotted-decimal IP address or resolvable host name. The port parameter is a valid system UDP port number to use for the session. The ttl (time-to-live) parameter is the network hop count which will scope (limit) the propagation of IP multicast packets sent. The optional error parameter allows the application to retrieve any error indication which occurs upon failure of this API call.

RETURN VALUES

An MdpSessionHandle value is returned for future use with other API calls to control the operation of the corresponding MdpSession. A value of MDP_NULL_SESSION is returned upon error and, if the application provided a pointer for the error parameter, an error indication will be passed here.

Function MdpDeleteSession()

SYNOPSIS

#include <mdpApi.h>

void MdpDeleteSession(MdpInstanceHandle instance,
                      MdpSessionHandle sessionHandle)

DESCRIPTION

This function is the complement of the MdpNewSession() function. Call this function to delete state for an existing MdpSession as identified by the sessionHandle parameter. The session will be closed and state for all pending objects will be deleted as the session shuts down. The instance parameter must be an appropriate value obtained from a previous call to MdpInit() and the sessionHandle must be a value returned from a a previous call to MdpNewSession() for the same given instance.

RETURN VALUES

The MdpDeleteSession() function returns no values.

5.2 MdpSession General Control

While a single MdpSession can play a role as a server (source) and/or a client (receiver), there are some parameters of operation which apply to the MdpNode independently to its mode(s) of participation within an MdpSession. For example, the data transmission rate of an MdpNode within an MdpSession will not exceed application-controlled limits regardless of its operation as a server and/or client. This section describes function calls which generally apply to an MdpSession (referenced by an MdpSessionHandle within the API).
Table Z - Summary of MdpSession General Parameter Control Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdpSessionSetTxRate()</td>
<td>Controls peak transmission rate for the specific MdpSession.</td>
</tr>
<tr>
<td>MdpSessionSetStatusReporting()</td>
<td>Enables/Disables transmission of periodic status/ statistic reports.</td>
</tr>
<tr>
<td>MdpSessionSetTOS()</td>
<td>Sets value for IP TOS field for transmitted packets.</td>
</tr>
<tr>
<td>MdpSessionSetMulticast-InterfaceAddress()</td>
<td>Determines which network interface will be used for multicast packet</td>
</tr>
<tr>
<td></td>
<td>transmission/reception.</td>
</tr>
<tr>
<td>MdpSessionSetMulticastLoopback()</td>
<td>Enables/disables loopback of multicast packets for session.</td>
</tr>
<tr>
<td>MdpSessionSetRobustFactor()</td>
<td>Set the “robustness” (number of repeat attempts) MDP uses in transmission</td>
</tr>
<tr>
<td></td>
<td>of MDP_CMD_FLUSH messages and requests for positive acknowledgement.</td>
</tr>
</tbody>
</table>

Function MdpSessionSetTxRate()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetTxRate(MdpSessionHandle sessionHandle, double txRate);

DESCRIPTION
The MdpSessionSetTxRate() function sets the maximum transmission rate in bits per second (bps) for the identified MDP session. This sessionHandle parameter identifies a valid MdpSession previously created with a call to MdpNewSession() and the txRate parameter establishes the transmission rate for that session. This function may be called at any time. Note that in cases where the previous transmission rate was very low, there might be a noticeable delay before transmission at the new data rate begins. When MDP’s congestion control algorithm is operating (see MdpSessionSetCongestionControl()), the rate automatically determined by the MDP protocol will change the rate if it is set by the application.

RETURN VALUES
The MdpSessionSetTxRate() function will return a value of MDP_ERROR_NONE if
successful. The value MDP_ERROR_SESSION_INVALID is returned upon failure.

**Function MdpSessionSetStatusReporting()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpError MdpSessionSetStatusReporting(
    MdpSessionHandle sessionHandle,
    bool state);
```

**DESCRIPTION**

The MdpSessionSetStatusReporting() function controls whether the local MdpNode will send out regular status reports as it participates in an MdpSession. The status report messages are special MDP protocol messages, which are periodically (once per 90 seconds) sent out by enabled MdpNodes. The default behavior, if this function is not called, is that the report messages are not sent. The primary purpose of these messages is to provide debugging and performance status information. In some cases, these reports might be useful in identifying multicast routing problems within a network. Currently, the content of reports received are not available to the application via the API. This feature may be provided in the future. The sessionHandle parameter identifies the MdpSession for which status reports should be sent and the state parameter controls the status of reporting. When the state parameter is set to a value of true, status reports are transmitted (after a 90 second interval) and when it is set to false, no reports are sent.

**RETURN VALUES**

The MdpSessionSetStatusReporting() function will return a value of MDP_ERROR_NONE if successful. The value MDP_ERROR_SESSION_INVALID is returned upon failure.

**Function MdpSessionSetTOS()**

**SYNOPSIS**

```c
#include <mdpApi.h>

bool MdpSessionSetTOS(MdpSessionHandle sessionHandle, unsigned char theTOS);
```

**DESCRIPTION**

This function allows the application to set the value of the IP type-of-service (TOS) field for any packets transmitted for the MdpSession indicated by the sessionHandle parameter. The tos parameter corresponds to what the TOS byte in the IP header will be set for transmitted packets. It should be noted that use of TOS is still considered experimental and different networks and devices may treat traffic differently. Therefore, the following discussion should be treated as a general guideline only.
The TOS byte in the IP header is divided into three sections: the PRECEDENCE field (the 3 high-order bits), the Type-of-Service (TOS) field (next 4 bits), and a reserved bit (the low order bit). Any one or none of the TOS bits can be set (5 different settings including the default setting of 0000), while the PRECEDENCE bits can be set to 8 different values including the default 000.

```
0 1 2 3 4 5 6 7
+----------------------------------+
| PRECEDENCE | TOS     | MBZ     |
+----------------------------------+
```

The table below gives examples of how the TOS field can be set to achieve different service. The listed “shifted integer value” represent the corresponding value that should be used in the call to MdpSessionSetTOS(). These values should be “OR’d” with the desired shifted integer value for the PRECEDENCE field.

### Table Q - TOS Field Values

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Definition</th>
<th>Shifted integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>IPTOS_LOWDELAY</td>
<td>16</td>
</tr>
<tr>
<td>0100</td>
<td>IPTOS_THROUGHPUT</td>
<td>8</td>
</tr>
<tr>
<td>0010</td>
<td>IPTOS_RELIABILITY</td>
<td>4</td>
</tr>
<tr>
<td>0001</td>
<td>IPTOS_LOWCOST</td>
<td>2</td>
</tr>
<tr>
<td>0000</td>
<td>normal service</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table R - PRECEDENCE Field Values

<table>
<thead>
<tr>
<th>Bit Pattern</th>
<th>Definition</th>
<th>Shifted Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>IPTOS_PREC_NETCONTROL</td>
<td>224</td>
</tr>
<tr>
<td>110</td>
<td>IPTOS_PREC_INTERNETCONTROL</td>
<td>192</td>
</tr>
<tr>
<td>101</td>
<td>IPTOS_PREC_CRITIC_ECP</td>
<td>160</td>
</tr>
<tr>
<td>100</td>
<td>IPTOS_PREC_FLASHOVERRIDE</td>
<td>128</td>
</tr>
<tr>
<td>011</td>
<td>IPTOS_PREC_FLASH</td>
<td>96</td>
</tr>
<tr>
<td>010</td>
<td>IPTOS_PREC_IMMEDIATE</td>
<td>64</td>
</tr>
<tr>
<td>001</td>
<td>IPTOS_PREC_PRIORITY</td>
<td>32</td>
</tr>
<tr>
<td>000</td>
<td>IPTOS_PREC_ROUTINE</td>
<td>0</td>
</tr>
</tbody>
</table>
Example:

If the value of `tos` parameter was set equal to 164, the PRECEDENCE would be IPTOS_PREC_CRITIC_ECP, the TOS would be IPTOS_RELIABILITY and the IP TOS byte would be set with the bit pattern: 10100100.

**RETURN VALUES**

A value of `true` is returned if the system call to set the TOS field value is successful. Note that if the MdpSession is not yet open when this call is made, this function will always return `true`. However, if the set TOS is invalid, the subsequent call to open the session for client or server operation will fail with an error. Some operating systems require super-user privileges in order to set certain TOS values (e.g. IPTOS_PREC_NETCONTROL).

**Function MdpSessionSetMulticastInterfaceAddress()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpError MdpSessionSetMulticastInterfaceAddress(
    MdpSessionHandle  sessionHandle,
    const char*      interfaceAddr);
```

**DESCRIPTION**

This function allows the application to select which network interface is used for multicast transmission and reception for the multicast group possibly associated with a given MdpSession. Internet Group Management Protocol (IGMP) messages for the group will use the specified interface. This allows MDP operation to be properly supported on multi-homed hosts. For hosts with a single network interface or configured to route multicast traffic on a designated interface, use of this API function is not necessary. The `sessionHandle` parameter identifies the applicable `MdpSession` and the `interfaceAddr` parameter is a pointer to a fixed string containing an IP address in dotted notation (e.g. “132.250.95.8”).

**RETURN VALUES**

The `MdpSessionSetStatusReporting()` function will return a value of `MDP_ERROR_NONE` if successful. The value `MDP_ERROR_SESSION_INVALID` is returned upon failure. While no specific errors are returned by this function, if the interface is improperly specified, multicast operation may not occur on the expected network interface.

**Function MdpSessionSetMulticastLoopback()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpError MdpSessionSetMulticastLoopback(
    MdpSessionHandle  sessionHandle,
    const char*      interfaceAddr);
```
bool state);

DESCRIPTION
This function enables \( \text{state} = \text{true} \) and disables \( \text{state} = \text{false} \) the loopback of multicast packets transmitted for the session identified by the \text{sessionHandle} parameter. This function is useful for debugging MDP applications (i.e. a single host can act as a client and server to itself) or for applications where it is necessary that the client receive local server transmissions.

RETURN VALUES
A value of MDP_ERROR_NONE is returned if successful. The value MDP_ERROR_SESSION_INVALID if the indicated session is invalid. Note that on some operating systems (e.g. Win32) it appears impossible to disable multicast loopback. Also note that on some operating systems, loopback must be enabled for multiple processes to receive multicast traffic from each other.

Function MdpSessionSetRobustFactor()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetRobustFactor(
    MdpSessionHandle sessionHandle,
    unsigned int value);

DESCRIPTION
This function controls the “robustness” with which certain MDP server commands are sent and timeout behavior of MDP clients. The \text{value} parameter determines the number of transmissions of MDP_CMD_FLUSH messages which occur at the end of server transmission and the number of attempts made to collect positive acknowledgement from ACKing clients. Note that this is very important to efficient protocol operation that all clients and servers operating in the same MdpSession use the same “robustness”\text{value}. Since the MDP protocol does not advertise server “robustness”, the application is responsible for assuring that all participated MdpNodes are set to the same “robustness”\text{value}.

If the \text{value} is set to zero, no MDP_CMD_FLUSH messages will be sent at the end of server transmission, but at least one attempt to collect positive acknowledgement from all ACKing receivers when positive acknowledgement is applicable. The default “robustness” \text{value} used by MDP is 20. This value is estimated to provide greater than 90% reliability even in cases of roughly 50% packet loss.

RETURN VALUES
A value of MDP_ERROR_NONE is returned if successful. The value MDP_ERROR_SESSION_INVALID if the indicated session is invalid.
5.3 **MdpSession Server Operation**

As indicated previously, the MDP API provides functions to open (start) and close (stop) an `MdpSession` for both server and client operation. Even for the same `MdpSession` within an application, the “server” and the “client” operations are somewhat independent of each other. This section describes function calls related to the `MdpSession`'s operation as a server (source of data transfers). For an application to participate as a server in a previously-created `MdpSession`, it must make a call to `MdpSessionOpenServer()`. However, there are a number of API calls controlling parameters applicable to server operation which may be called before server operation is activated with a call to `MdpSessionOpenServer()`. And there are some function calls which can be made only before server operation is started. After a session has been opened for server operation, the application may enqueue file and/or static data objects for transmission. Table X summarizes the API routines applicable to server operation. Detailed descriptions of each of these functions is provided below.

<table>
<thead>
<tr>
<th><strong>Table X - Summary of MdpSession Server Management Functions</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Server parameter control routines which can be called at any time:</strong></td>
</tr>
<tr>
<td>MdpSessionSetCongestionControl()</td>
</tr>
<tr>
<td>MdpSessionSetAutoParity()</td>
</tr>
<tr>
<td>MdpSessionSetServerEmcon()</td>
</tr>
<tr>
<td>MdpSessionServerAddAckingClient()</td>
</tr>
<tr>
<td>MdpSessionServerRemoveAckingClient()</td>
</tr>
<tr>
<td>MdpSessionSetTxCacheDepth()</td>
</tr>
<tr>
<td>MdpSessionGetServerGrttEstimate()</td>
</tr>
<tr>
<td>MdpSessionSetServerGrttEstimate()</td>
</tr>
<tr>
<td>MdpSessionSetServerGrttMax()</td>
</tr>
</tbody>
</table>
## Table X - Summary of MdpSession Server Management Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MdpSessionSetGrttProbeInterval()</code></td>
<td>Set the time interval ranges used by the MDP server for GRTT probing.</td>
</tr>
<tr>
<td><code>MdpSessionSetGrttProbing()</code></td>
<td>Enable/disable MDP GRTT probing. Disabling allows complete manual setting of server GRTT estimate.</td>
</tr>
</tbody>
</table>

**Server parameter control routines to be called only before starting server operation:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MdpSessionSetBaseObjectTransportId()</code></td>
<td>Set initial transport object identifier value.</td>
</tr>
</tbody>
</table>

**Server start/stop control:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MdpSessionOpenServer()</code></td>
<td>Start server operation.</td>
</tr>
<tr>
<td><code>MdpSessionCloseServer()</code></td>
<td>Stop server operation.</td>
</tr>
</tbody>
</table>
### Function MdpSessionOpenServer()

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpError MdpSessionOpenServer(
    MdpSessionHandle sessionHandle,
    int segmentSize = MDP_DEFAULT_SEGMENT_SIZE,
    int blockSize = MDP_DEFAULT_BLOCK_SIZE,
    int nparity = MDP_DEFAULT_NPARITY,
    unsigned long bufferSize = MDP_DEFAULT_BUFFER_SIZE);
```

**DESCRIPTION**

The `MdpSessionOpenServer()` function opens a session for operation as a server (sender) within an `MdpSession`. The `sessionHandle` identifies the MdpSession to be opened for server operation. Default values are provided for the other parameters which can be overridden at the application designer's discretion. These parameters include:

- **segmentSize**: Maximum payload size (in bytes) of MDP protocol messages. MDP protocol messages are encapsulated within UDP/IP packets and some messages contain as much as 25 bytes of MDP-specific overhead in addition to the payload content. The “normal” amount of MDP overhead in addition to the payload is 21 bytes. The `segmentSize` parameter allows the application additional control for protocol efficiency so that MDP operation can be properly matched to expected network conditions and maximum transmission unit (MTU) sizes.

- **blockSize**: Number of “data” payload packets per MDP forward error correction (FEC) coding block. This must be a value from 1 to 255. The product

---

### Summary of MdpSession Server Management Functions

**Table X - Summary of MdpSession Server Management Functions**

<table>
<thead>
<tr>
<th>Server management routines for use only after server start:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdpSessionQueueTxFile() Enqueue file object for transmission by the server.</td>
</tr>
<tr>
<td>MdpSessionQueueTxData() Enqueue static data object for transmission by the server.</td>
</tr>
<tr>
<td>MdpSessionRemoveTxObject() Abort transmission of currently active transmit object (or one being held for potential repair transmissions)</td>
</tr>
</tbody>
</table>
nparity

Number of “parity” packets the server calculates and maintains for potential repair transmission per MDP FEC coding block. Note that these packets are simply calculated and not actually transmitted except in response to repair requests by clients or to satisfy the current server “auto parity” (discussed later) setting.

bufferSize

Amount of memory the MDP protocol engine allocates as buffer space for holding parity for repair transmissions to clients. A large bufferSize setting allows an MDP server to operate most efficiently in terms of processing demand. However, MDP will re-calculate parity as needed for retransmission within the bounds of the transmission “cache depth” set with the MdpSessionSetTxCacheDepth() function.

During the call to MdpSessionOpenServer(), the application-installed TimerCallbackFunction will be called as timers critical to server operation are installed. Immediately after opening an MdpSession for server operation (and after returning to your application’s event loop), the MDP protocol engine will begin transmission of MDP protocol messages appropriate to server operation. After the MdpSession is opened as a server, the application may begin queuing objects for transmission with calls to MdpSessionQueueTxFile() and MdpSessionQueueTxData().

RETURN VALUES

On success, a value of MDP_ERROR_NONE is returned. Otherwise, an MDP error code indicative of the problem opening server operation is returned.

Function MdpSessionCloseServer()

SYNOPSIS

#include <mdpApi.h>

void MdpSessionCloseServer(
    MdpSessionHandle sessionHandle,
    bool hardShutdown = false);

DESCRIPTION

The function MdpSessionCloseServer() is used to close (stop) server operation for a session which was previously opened. The sessionHandle parameter identifies the session to close. Note that the same session may be re-opened with another call to MdpSessionOpenServer(). However, any objects previously queued for transmission are purged when server operation is closed. The hardShutdown parameter is used to control the server’s behavior as it exits the MdpSession. By default (hardShutdown = false), the server will “gracefully” exit the session by transmitting an MDP_CMD_FLUSH message with the End-of-Transmission flag set. This notifies clients within the group that the server is terminating operation and it will not process further repair requests for any pending objects. An MDP_NOTIFY_SERVER_CLOSED notification occurs when this end-of-transmission flushing is complete. The session should not be re-opened as a server until this

32
notification has occurred. The period of time for the graceful close process to complete is a function of the server's current estimate of GRTT and the “robust factor” set for the session. If the hardShutdown parameter is set to a value of true, the server immediately halts with no notification to clients. In the future, a function to temporarily halt or pause server operation might be provided.

RETURN VALUES

MdpSessionCloseServer() returns no value.

Function MdpSessionQueueTxFile()

SYNOPSIS

#include <mdpApi.h>

MdpObjectHandle MdpSessionQueueTxFile(
    MdpSessionHandle sessionHandle,
    const char* path,
    const char* name,
    MdpError* error = NULL);

DESCRIPTION

The function MdpSessionQueueTxFile() is used to enqueue an MdpObject of type MDP_OBJECT_FILE for transmission by the server. The current MDP protocol implementation supported by this API handles all file I/O directly. The sessionHandle parameter identifies the MdpSession to which the file object is transmitted. The server transmits the file from the local file system of the name corresponding to the concatenation of the path and name text provided in this call. Additionally, the name information is transmitted as a single MDP_INFO packet with a name length limited to the segment size set when the server was opened. The file data content is transmitted and repaired as MDP_DATA. In the future, an option may be provided to suppress the transmission of MDP_INFO for MDP_OBJECT_FILE objects or allow the application to override the behavior of encapsulating the file name information (e.g using MDP_INFO content for MIME-type information may be useful for some envisioned applications)

RETURN VALUES

If the file is successfully queued, the return value contains a unique MdpObjectHandle which may be used to reference the object in other API calls. For example, if the application needs to prematurely de-queue (abort) the file transmission before MDP is completely finished with it, the MdpObjectHandle returned can be used with the MdpSessionRemoveTxObject() function call. A value of MDP_NULL_OBJECT is returned if the attempt to queue the file for transmission fails. An MdpError value indicating the reason for failure is returned in the memory space indicated by the application-provided error pointer parameter. Possible errors include:

    MDP_ERROR_SESSION_INVALID  sessionHandle refers to an non-existent MdpSession
Unable to allocate sufficient memory to maintain transmit MdpObject state.

MdpSession transmit queue (cache) is full. See MdpSessionSetTxCacheDepth() description for information on controlling the transmit queue depth.

The file name specified by the path/name parameters is not valid.

Function MdpSessionQueueTxDataObject()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionQueueTxDataObject(
    MdpSessionHandle sessionHandle,
    const char* infoPtr,
    unsigned short infoSize,
    const char* dataPtr,
    unsigned long dataSize,
    MdpError* error);

DESCRIPTION
The MdpSessionQueueTxData() function enqueues an MdpObject of type MDP_OBJECT_DATA for transmission in the MdpSession indicated by the sessionHandle parameter. The data to be sent is a static (unchanging) block of data in the application’s memory space or a short (single segment/packet) message as described below. The sessionHandle referenced must correspond to an MdpSession which has previously been successfully opened for operation as a server. The remaining parameters allow the application to specify the content to be transmitted for enqueued MdpObject.

As described in the MDP protocol documentation, there are two possible components to MdpObjects transmitted. These include optional, “out-of-band” information which must be capable of being encapsulated in a payload size less than or equal to the segmentSize for the transmitting MDP server (see MdpSessionOpenServer() for a description of the segmentSize server parameter). This information is transmitted as a message of type MDP_INFO within the MDP protocol. MDP_INFO packets (related to the fact that there is no fragmentation of content) are repaired more rapidly by the protocol so some applications may have use for this capability in their context of operation. Additionally, it may be useful for applications to transmit data messages which are known to be one segmentSize in length or smaller simply as MdpObjects with only MDP_INFO content (i.e. dataPtr = NULL).
This can speed up the transmission/repair process for small application messages. Note that MDP_INFO message are not included in MDP’s FEC coding so MDP_INFO messages cannot utilize the benefits of the autoParity option for server transmissions. The infoPtr parameter points to an application memory location containing the data the application wishes to be transmitted as MDP_INFO for the newly queued object and the infoSize indicates the length of the info data. The infoSize parameter must be less than or equal to the segmentSize specified in the previous call to MdpSessionOpenServer(). Note that the underlying MDP protocol engine allocates storage for the indicate info data and copies the data into the newly allocated storage. Therefore, the application may free or reuse the memory referenced by the infoPtr without disrupting protocol operation after a call to this function. (As described below, his is not the case for the data referenced by the dataPtr parameter) No MDP_INFO message is sent for the transmit MdpObject if the infoPtr parameter is equal to NULL.

The dataPtr parameter must point to a valid memory location that the application wishes the MDP protocol to transmit as MDP static data content. The dataSize parameter specifies the length of the data to be transmitted. IMPORTANT: The MDP protocol engine may reference this data storage location for future repair transmissions so it is critical that the application not free or modify the memory indicated by the dataPtr parameter until the transmit MdpObject is removed from the transmission queue via a call to the MdpSessionRemoveTxObject() API function, or a notification event of type MDP_NOTIFY_TRANSMIT_OBJECT_FINISHED or MDP_NOTIFY_OBJECT_DELETE is received via the the application-installed MdpNotifyCallback.

RETURN VALUES

If the data object is successfully queued, the return value contains a unique MdpObjectHandle which may be used to reference the object in other API calls. For example, if the application needs to prematurely de-queue (abort) the data object transmission before MDP is completely finished with it, the MdpObjectHandle returned can be used with the MdpSessionRemoveTxObject() function call. A value of MDP_NULL_OBJECT is returned if the attempt to queue the data for transmission fails. An MdpError value indicating the reason for failure is returned in the memory space indicated by the application-provided error pointer parameter. Possible errors include:

MDP_ERROR_SESSION_INVALID sessionHandle refers to an non-existent MdpSession

MDP_ERROR_MEMORY Unable to allocate sufficient memory to maintain transmit MdpObject state. This error code may also occur if the infoSize parameter exceeds the segmentSize specified for server transmissions.
MdpSession transmit queue (cache) is full. See MdpSessionSetTxCacheDepth() description for information on controlling the transmit queue depth.

**Function MdpSessionRemoveTxObject()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpError MdpSessionRemoveTxObject(
    MdpSessionHandle sessionHandle,
    MdpObjectHandle objectHandle);
```

**DESCRIPTION**

This function is used to remove (abort transmission of) MdpObjects which have been previously queued for transmission from the MdpSession’s transmit queue. The sessionHandle indicates the applicable MdpSession and the objectHandle parameter specifies the object to be removed. Note that this function should not be called from with the application installed MDP MdpNotifyCallback function when the notification event pertains to the same object referenced by the objectHandle parameter.

**RETURN VALUES**

A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID or MDP_ERROR_OBJECT_INVALID may be returned upon error.

**Function MdpSessionSetCongestionControl()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpError MdpSessionSetCongestionControl(
    MdpSessionHandle sessionHandle,
    bool state);
```

**DESCRIPTION**

This function enables (and disables) the operation of an experimental congestion control algorithm built into the MDP protocol. The goal of this congestion control algorithm is to automatically control the transmission rate of an MDP server such that its transmissions maximally utilize available network throughput while fairly sharing the capacity with other MDP and TCP flows. This feature of MDP is still under development and is not yet recommended for widespread use. The sessionHandle parameter specifies the applicable MdpSession and the state parameter set to values of true and false will enable and disable
congestion control operation, respectively. Congestion control operation is controlled by individual server nodes. Client nodes will automatically appropriately respond to servers running with congestion control enabled.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error.

Function MdpSessionSetAutoParity()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetAutoParity(
    MdpSessionHandle   sessionHandle,
    unsigned char     autoParity);

DESCRIPTION
This function controls how many FEC parity repair packets are automatically sent with each MDP FEC coding block. The default is zero where parity repair packets are sent only on demand in response to requests from clients. Setting some non-zero amount of “auto” parity may yield performance benefits over long delay network connections where there is some a priori known amount of expected packet loss. The sessionHandle parameter identifies the applicable MdpSession and the autoParity parameter sets the number of automatic repair packets. The value of the autoParity parameter must be less than or equal to the nparity parameter set when the MdpSession was opened for server operation.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID or MDP_ERROR_PARAMETER_INVALID may be returned upon error.

Function MdpSessionSetServerEmcon()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetServerEmcon(
    MdpSessionHandle   sessionHandle,
    bool               state);

DESCRIPTION
This function enables (and disables) special protocol features to support emissions-controlled (EMCON) transmissions by an MdpSession operating as a server. EMCON modes of
operation for MDP are under continued development so the specific protocol details relative to
EMCON operation are not yet finalized. However, at the current time, use of EMCON
operation causes a server to redundantly transmit objects’ MDP_INFO messages at the end of
each transmitted coding block in addition to sending MDP_INFO at the start of object
transmission. This is because MDP_INFO messages do not benefit from MDP’s FEC coding
structure. In general, MDP applications wishing to provide EMCON operation should not use
the optional MDP_INFO messages for object transmission in order to achieve the full gain of
MDP’s FEC coding. The sessionHandle parameter specifies the applicable MdpSession
and the state parameter set to values of true and false will enable and disable EMCON
operation, respectively.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of
MDP_ERROR_SESSION_INVALID may be returned upon error.

Function MdpSessionAddAckingClient()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionAddAckingClient(
    MdpSessionHandle sessionHandle,
    const char* host);

DESCRIPTION
This function adds a specific client (identified by its host name or IP address) to the list of
clients from which the server expects to receive positive receipt acknowledgement during
protocol operation. If MDP status reporting is used, it is important that the clients in this list
be appropriately configured for positive acknowledgement operation so that the clients’ status
reports do not cause them to be removed from the server’s positive acknowledgement list.
Controls over positive acknowledgement operation will be further refined in the future. The
sessionHandle parameter identifies the applicable MdpSession and the host parameter
points to a string containing the client’s resolvable host name or IP address in dotted notation
(e.g. “132.250.95.7”).

IMPORTANT NOTE: Identifying clients by their host names or IP addresses will lead to
problems for MDP applications which use an alternate MdpNodeIds convention. And, even if
IP addresses are user as MdpNodeIds, multi-homed hosts can add to confusion here.
Therefore, in the future, this API call will be changed to use a type of MdpNodeIds for the
second parameter to this function to be more consistent with the general utility of the MDP
API.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of
MDP_ERROR_SESSION_INVALID or MDP_ERROR_DNS_FAIL may be returned upon
error.

Function MdpSessionRemoveAckingClient()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionServerRemoveAckingClient(
    MdpSessionHandle sessionHandle,
    const char* host);

DESCRIPTION
This function removes a specific client (identified by its host name or IP address) from the list of clients from which the server expects to receive positive receipt acknowledgement during protocol operation. See the description of MdpSessionAddAckingClient() for more information.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error.

Function MdpSessionSetTxCacheDepth()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetTxCacheDepth(
    MdpSessionHandle sessionHandle,
    unsigned long minCount,
    unsigned long maxCount,
    unsigned long maxSize);

DESCRIPTION
This function sets the properties of the MdpSession's server transmit cache. The transmit cache acts as a “hold queue” for repairing previously-transmitted MdpObjects. Larger transmit cache settings allow an MDP server to repair previously transmitted objects over a longer time/bandwidth window. This is particularly important when using MDP as a purely NACK-based protocol without any positive acknowledgements. The transmit cache does not directly use much memory since MDP will pull information from the application's disk storage or memory as needed to recover repair information. When new objects are queued for transmission, MDP will remove state for old objects in the transmit cache as needed to make room for state for the new objects. The following parameters govern the behavior of the transmit cache:

minCount  Smallest number of objects (files or data) for which state is kept (regardless of
the size of the individual objects.

**maxCount**  Maximum number of objects (files or data) for which state is kept although the maxCount may not be reached if the total size of the individual objects exceed the maxSize parameter value.

**maxSize**  Maximum total size (in bytes) of objects for which state is kept in the transmit cache. However, the total size can exceed the maxSize parameter value if there are minCount or less objects in the cache.

If minCount and maxCount are set to zero, state for the object will be deleted immediately after its transmission and no repair transmissions will be sent in response to requests from clients.

**RETURN VALUES**

A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error.

---

**Function MdpSessionServerSetGrttEstimate()**

**SYNOPSIS**

```c
#include <mdpApi.h>

MdpError MdpSessionServerSetGrttEstimate(
    MdpSessionHandle sessionHandle,
    double initialGrtt);
```

**DESCRIPTION**

This function sets the value of the estimate of group greatest roundtrip delay time (GRTT) maintained by the server. As an integral part of the MDP protocol, MDP servers collect a long term estimate of the worst case roundtrip packet transmission delay time among active members in the MdpSession. This function allows an application to override the current estimate at any time. If an application had a good a priori idea of expected GRTT at session startup (obtained from previous runs as a server, or from side information), it could use this function to initialize the server’s starting GRTT estimate to an appropriate value. The server will then update this value with its own estimation and averaging process. The sessionHandle parameter indicates the applicable MdpSession and the initialGrtt parameter sets the server’s GRTT estimate in units of seconds. Also note that the GRTT estimate maintained by an MDP server is currently limited to a range of MDP_GRTT_MIN to MDP_GRTT_MAX (currently 1 msec and 15 sec, respectively).

**RETURN VALUES**

A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error.
Function MdpSessionServerGetGrttEstimate()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetGrttEstimate(
    MdpSessionHandle sessionHandle,
    double* grttEstimate);

DESCRIPTION
This function returns the MDP server’s current estimate of GRTT for the MdpSession indicated by the sessionHandle parameter. The estimate is returned (in units of seconds) to the space pointed to by the grttEstimate parameter. Note that the estimate returned is subject to the limits imposed by the MDP protocol engine and might not reflect actual GRTT in extreme cases (See the description of the MdpSessionSetGrttMax() function).

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error. The MDP server’s current estimate of GRTT is passed back via the grttEstimate pointer.

Function MdpSessionServerSetGrttMax()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetGrttEstimate(MdpSessionHandle sessionHandle,
    double* grttMax);

DESCRIPTION
This function sets the maximum value the server will use and advertise as its estimate of group greatest roundtrip delay time (GRTT). The default value (i.e. don’t call this function) is generally recommended as it represents a trade-off in the protocol’s ability to adapt to extreme network conditions and maintain responsiveness. While the MDP protocol’s quantization and encapsulation of advertised GRTT can support values up to 1000 seconds, the code currently limits the maximum which can be set to 120 seconds. (The normal default maximum is 15 seconds).

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error.

Function MdpSessionSetGrttProbeInterval()
SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetGrttProbeInterval(MdpSessionHandle sessionHandle,
                          double intervalMin,
                          double intervalMax);

DESCRIPTION
This function sets the time intervals (in seconds) the MDP server uses in its algorithm for collecting an estimate of group greatest round trip time (GRTT) for the MdpSession indicated by the sessionHandle parameter. The intervalMin parameter is the probing interval used when MDP begins probing. Subsequently, the probe interval is exponentially increased (by doubling) until the steady-state interval specified by intervalMax is reached. If the values of intervalMin and intervalMax are equal, the probe interval is constant. The value of intervalMin must be greater than or equal to 0.10 seconds and the value of intervalMax must be greater than or equal to 5.0 seconds.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success or MDP_ERROR_PARAMETER_INVALID if the intervals are not in an acceptable range (or if intervalMax is less than intervalMin). Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error.

Function MdpSessionSetGrttProbing()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetGrttProbing(MdpSessionHandle sessionHandle,
                          bool state);

DESCRIPTION
This function enables or disables the MDP server's algorithm for collecting a GRTT estimate from the group. The sessionHandle parameter is used to specify the applicable MdpSession and the state parameter is set to a value of true or false to enable or disable GRTT probing, respectively. Normally (and by default) GRTT probing should be enabled for most efficient protocol operation. The ability to disable GRTT probing is only provide for cases where network capacity is extremely limited (and a reasonable estimate of actual GRTT is known a priori) or for cases when no clients are providing feedback (all clients configured for EMCON operation).

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error.
Function MdpSessionSetBaseObjectTransportId()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetBaseObjectTransportId(
    MdpSessionHandle  sessionHandle,
    MdpObjectTransportId  baseId);

DESCRIPTION
This function sets the starting transport object identifier for the series of objects to be transmitted by an MDP server. MDP servers transmit objects with monotonically increasing transport object identifiers during the course of an MdpSession. MDP client nodes “synchronize” to an MDP server based on the transport object identifier of the first received message from the specific MDP server. This is part of what allows multicast receivers to “come and go” as they please within the context of an MDP session. Because the clients use these transport object identifiers to uniquely identify an object from a server, it is very important that, when a server application is stopped and restarted, it begins the subsequent transmission with a base object transport identifier sufficiently out-of-range of the identifier of the last object transmitted. There are different methods applications may use to achieve this goal. The “tkMdp” example application randomly selects a base object transport identifier at startup. Because the probability of selecting a number too close in range (currently 256) of the previous transport identifier is very low, this was deemed an acceptable method for the demonstration application. More robust schemes where the application logs the most recent object transport identifier in non-volatile storage (e.g. on a disk file) and then picks an appropriate restart value after the server is exited (because of a system failure or other reason) can also be implemented. The API function call MdpObjectGetTransportId() can be used to retrieve the transport object identifier of a newly-queued transmit object to serve this purpose. While the current object transport identifier range “window” over which a client tracks synchronization with a given server is 256, this value may be controlled via the API in the future. More information on the “synchronization window” will be provided then.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. Otherwise a value of MDP_ERROR_SESSION_INVALID may be returned upon error.

5.4 MdpSession Client Operation
This section describes functions related to the MDP protocol engine’s capabilities to participate as a client within an MdpSession. The general order of operation is that an application may open an MdpSession for client operation. At this point, the MDP protocol is open to receiving transmissions from any server(s) transmitting to the MdpSession. The MdpNotifyCallback installed by the application will be called as MdpObjects are received from servers. Additional API calls to retrieve information about the MdpObjects being received are described later. Currently, there are no controls within the MDP API to restrict
reception of any data or from any specific server(s). In the near future, it is likely the API will be expanded to allow MDP client applications to “pick and choose” which portions of data transmitted to an MdpSession they wish to receive. Also, as a client receives transmissions from a new server within the session, it allocates memory to buffer reliable reception for each individual server. So while in principle an MdpSession may have an arbitrary number of sources of data, the practical number of sources is limited by the clients’ ability to allocate buffer space for multiple sources. In the future, the API will be extended to allow client applications to “timeout” or explicitly “close” specific sources (thus freeing allocated memory resources) in addition to the receive data object filtering ability previously described. This will allow applications to create more flexible paradigms to support “many-to-many” communications. Meanwhile, the basic capabilities of the API allow an application to participate as a receiver in an MdpSession with relatively simple code. The functions related to participate as a client simply consist of functions to open (start) client participation, and control a few parameters of client operation. After this, it is primarily the code within the MdpNotifyCallback which allows the application to receive transmissions from servers and process the received objects properly. Table Y summarizes the functions specifically related to controlling client operation.

<table>
<thead>
<tr>
<th>Table Y - Summary of MdpSession Client Management Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client parameter control routines which can be called at any time:</strong></td>
</tr>
<tr>
<td>MdpSessionSetClientAcing()</td>
</tr>
<tr>
<td>MdpSessionSetClientUnicastNacks()</td>
</tr>
<tr>
<td>MdpSessionSetClientNackingMode()</td>
</tr>
<tr>
<td>MdpSessionSetClientStreamIntegrity()</td>
</tr>
<tr>
<td>MdpSessionSetClientMulticastAcks()</td>
</tr>
</tbody>
</table>

44
Table Y - Summary of MdpSession Client Management Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdpSessionSetArchiveDirectory()</td>
<td>Specifies directory where client stores received objects of type MDP_OBJECT_FILE.</td>
</tr>
<tr>
<td>MdpSessionSetArchiveMode()</td>
<td>Specifies whether client should permanently archive or temporarily caches received file objects. (Affects naming of received files and overwrite policy).</td>
</tr>
</tbody>
</table>

Client parameter control routines which should be called before client start:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdpSessionSetClientEmcon()</td>
<td>Controls optional emission-controlled (silent client) mode of operation for client.</td>
</tr>
</tbody>
</table>

Client control

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MdpSessionOpenClient()</td>
<td>Start client operation.</td>
</tr>
<tr>
<td>MdpSessionCloseClient()</td>
<td>Stop client operation.</td>
</tr>
<tr>
<td>MdpSessionAbortRxObject()</td>
<td>Terminate reception of a specific MdpObject.</td>
</tr>
<tr>
<td>MdpSessionGetNodeAddress()</td>
<td>Get the current source IP address and port number for a remote server node in the MdpSession.</td>
</tr>
<tr>
<td>MdpSessionGetNodeGrttEstimate()</td>
<td>Get the current GRTT estimate for a remote server in the MdpSession.</td>
</tr>
<tr>
<td>MdpSessionGetNodeNextRxObject()</td>
<td>Allows caller to retrieve the list of pending receive MdpObjects being received from a remote server.</td>
</tr>
<tr>
<td>MdpSessionDeactivateNode()</td>
<td>Frees buffering resources allocated for a remote server, dropping state on pending receive MdpObjects.</td>
</tr>
<tr>
<td>MdpSessionDeleteNode()</td>
<td>Drops all state and free all resources for a remote server MdpNode.</td>
</tr>
</tbody>
</table>

Function MdpSessionOpenClient()

SYNOPSIS
#include <mdpApi.h>
MdpError MdpSessionOpenClient(
    MdpSessionHandle sessionHandle,
    unsigned long bufferSize = MDP_DEFAULT_BUFFER_SIZE);

DESCRIPTION
The MdpSessionOpenClient() function opens a session for operation as a client (receiver) within an MdpSession. The sessionHandle parameter specifies the applicable session and the bufferSize parameter indicates how much memory (in bytes) the client should allocate for each server which is detected transmitting to the session. A default value of approximately 1 Mbyte for the bufferSize parameter is provided. After the session is opened for client operation, the client will begin waiting for transmissions from server nodes within the MdpSession. The client will also generate MDP status reports once per 90 seconds if it has been configured to do so.

RETURN VALUES
On success a value of MDP_ERROR_NONE. A value of MDP_ERROR_SESSION_INVALID or MDP_ERROR_MEMORY will returned upon failure.

    Function MdpSessionCloseClient()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionCloseClient(MdpSessionHandle sessionHandle);

DESCRIPTION
The MdpSessionCloseClient() function closes (stops) client operation for the MdpSession identified by the sessionHandle parameter.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. MDP_ERROR_SESSION_INVALID is returned otherwise.

    Function MdpSessionAbortRxObject()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionAbortRxObject(MdpSessionHandle sessionHandle,
                                 MdpObjectHandle objectHandle);

DESCRIPTION
The MdpSessionAbortRxObject() terminates reception of an MdpObject being received by a client. This function may be called at any time during the interval during which a given MdpObjectHandle is valid (i.e. from the time of MDP_NOTIFY_RX_OBJECT_START until MDP_NOTIFY_OBJECT_DELETE). The sessionHandle and objectHandle parameters indicate the applicable MdpSession and MdpObject, respectively. Note that even though the application may abort reception of an object, if a subsequent positive acknowledgement request for that object is sent by the server, the aborting receiver will acknowledge (ACK) that reception was completed. The positive acknowledgement mechanism in MDP is designed with respect to the successful transport of the object. Application layer needs for positive acknowledgement need to be addressed separately by the application.

RETURN VALUES

A value of MDP_ERROR_NONE is returned upon success. MDP_ERROR_SESSION_INVALID is returned if the sessionHandle is invalid, or MDP_ERROR_OBJECT_INVALID is returned if the objectHandle value is equal to MDP_NULL_OBJECT. It is critical that only valid non-null MdpObjectHandle's are passed to this function call.

Function MdpSessionGetNodeAddress()

SYNOPSIS

#include <mdpApi.h>

MdpError MdpSessionGetNodeAddress(
    MdpSessionHandle sessionHandle,
    MdpNodeId  nodeId,
    unsigned long* address,
    unsigned short* port);

DESCRIPTION

This function retrieves the IP source address and port number for the MdpNode identified by the nodeId parameter. The nodeId value may have been previously obtained with a call to MdpObjectGetSourceNodeId(). The sessionHandle parameter identifies the applicable MdpSession and the address and port parameters point to storage locations for the respective retrieved values.

RETURN VALUES

This function will return a value of MDP_ERROR_NONE if successful. A value of MDP_ERROR_SESSION_INVALID is returned upon failure.

Function MdpSessionGetNodeGrttEstimate()
# SYNOPSIS

```c
#include <mdpApi.h>

MdpError MdpSessionGetNodeGrttEstimate(
    MdpSessionHandle sessionHandle,
    MdpNodeId nodeId,
    double* grttEstimate);
```

# DESCRIPTION

This function retrieves the current estimate of GRRT advertised by the remote server node indicated by the `nodeId` parameter in the `MdpSession` identified by the `sessionHandle` parameter. The estimate of GRRT is returned to the storage space indicated by the `grttEstimate` pointer parameter in units of seconds. A valid `MdpNodeId` can be retrieved using the `MdpObjectGetSourceNodeId()` function given a valid `MdpObjectHandle` for a current object being received. The retrieved `MdpNodeId` is guaranteed to be valid only during the period of time for which `MdpObjectIds` associated with the same source are valid.

## RETURN VALUES

A value of MDP_ERROR_NONE is returned upon success. If the `nodeId` parameter is not valid, a value of MDP_ERROR_NODE_INVALID will be returned.

## Function MdpSessionGetNodeNextRxObject()

# SYNOPSIS

```c
#include <mdpApi.h>

MdpObjectHandle MdpSessionGetNodeNextRxObject(
    MdpSessionHandle sessionHandle,
    MdpNodeId nodeId,
    MdpObjectHandle* previousObject,
    MdpError* error = NULL);
```

# DESCRIPTION

This function allows the application to retrieve the list of pending receive objects for a particular remote server `MdpNode` one at a time. The `sessionHandle` parameter identifies the applicable `MdpSession` and the `nodeId` parameter identifies the remote server of interest. The `previousObject` parameter is used to start the list iteration (using a value of MDP_NULL_OBJECT) and to prompt the API for the next pending receive object using previously returned values. Thus, this function can be used to iterated over the list of pending receive `MdpObjects`.

## RETURN VALUES

If the `previousObject` parameter is set to a value of MDP_NULL_OBJECT (or an...
invalid MdpObjectHandle), the identifying handle of the first receive-pending MdpObject is
returned. If the previousObject parameter is a valid MdpObjectHandle, the handle of
the next following receive-pending MdpObject is returned. A value of MDP_NULL_OBJECT
may also be returned upon error. The storage identified by the error parameter will contain a
value indicating the error status of MDP_ERROR_NONE,
MDP_ERROR_SESSION_INVALID, or MDP_ERROR_NODE_INVALID.

Function MdpSessionDeactivateNode()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionDeactivateNode(
    MdpSessionHandle sessionHandle,
    MdpNodeId nodeId);

DESCRIPTION
This function releases buffering resources allocated for a remote server MdpNode, dropping
state on receive pending MdpObjects for that server, but maintains the server’s state with
regards to which MdpObjects have been received and other factors. The bulk of memory
resources used by the MDP library for client operation is for receive buffering. This function
allows the application to manage receive buffering memory resources in a timely manner.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. A value of
MDP_ERROR_SESSION_INVALID or MDP_ERROR_NODE_INVALID may be returned if
inappropriate (e.g. MDP_NULL_SESSION) values are supplied.

Function MdpSessionDeleteNode()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionDeleteNode(
    MdpSessionHandle sessionHandle,
    MdpNodeId nodeId);

DESCRIPTION
This function releases all memory resources and state allocated by a MDP client for a remote
server MdpNode. No state on the remote server is retained and when the client receive new
messages from that server, it will re-allocate state for the server and begin reception as if it
were a newly detected server.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. A value of MDP_ERROR_SESSION_INVALID or MDP_ERROR_NODE_INVALID may be returned if inappropriate (e.g. MDP_NULL_SESSION) values are supplied.

**Function MdpSessionSetClientAcking()**

**SYNOPSIS**

```c
#include <mdpApi.h>
MdpError MdpSessionSetClientAcking(
    MdpSessionHandle sessionHandle, bool state);
```

**DESCRIPTION**

The MdpSessionSetClientAcking() function controls the client node's willingness to participate in positive acknowledgement of the receipt of MdpObjects transmitted by servers. When MDP status reporting is used (see MdpSessionSetStatusReporting()), a flag in the report indicates the client's ACKing status. It is important to note that this flag must be properly set when a server has included the client in question in its positive acknowledgement list. Even though the client is in the list at startup, the client will be removed from the list if the server receives a report from the client with its ACKing status flag unset. Controls of the positive acknowledgement process will be further refined in the future as the need for this functionality in different reliable multicast applications is better defined. The sessionHandle parameter indicates the applicable MdpSession and the state parameter set to a value of true or false, enables or disables client ACKing, respectively.

**RETURN VALUES**

A value of MDP_ERROR_NONE is returned upon success. MDP_ERROR_SESSION_INVALID is returned otherwise.

**Function MdpSessionSetClientUnicastNacks()**

**SYNOPSIS**

```c
#include <mdpApi.h>
MdpError MdpSessionSetClientUnicastNacks(
    MdpSessionHandle sessionHandle, bool state);
```

**DESCRIPTION**

This function controls how MDP NACK messages are transmitted by a client. Normally, it is anticipated that MDP NACK messages will be multicast to the session to promote NACK
suppression among MDP clients. However, in some asymmetric network topologies without reciprocal multicast routing, it may be necessary for some clients to unicast NACK messages directly to the server source address(es). The sessionHandle parameter indicates the applicable MdpSession and the state parameter set to a value of true causes the client to unicast NACK messages to the server source address(es), while a value of false causes the client to send NACK messages to the MdpSession destination address (which may be a multicast or unicast address).

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. MDP_ERROR_SESSION_INVALID is returned otherwise.

Function MdpSessionSetClientNackingMode()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetClientNackingMode(
    MdpSessionHandle sessionHandle,
    MdpNackingMode mode);

DESCRIPTION
This function controls the default client nacking behavior for the MdpSession indicated by the sessionHandle parameter. The possible values for the mode parameter include MDP_NACKING_NORMAL, MDP_NACKING_INFOONLY, and MDP_NACKING_NONE. This default behavior applies to newly-received objects and objects “missing” from the series of server object transmissions. When the client's MdpNackingMode is set to MDP_NACKING_NORMAL, the client will request repair retransmission of the entire content of any missing or newly received MdpObjects as necessary. When the client MdpNackingMode is set to MDP_NACKING_INFOONLY, the client will request retransmission only for missing MdpObject info content, if applicable. And when the client MdpNackingMode is set to MDP_NACKING_NONE, the client will not request any repair for missing data (Note that the client may possibly still receive the object in question as a result of other client's repair requests).

The MdpNackingMode may be set for individual MdpObjects (see the MdpObjectSetNackingMode() function) to allow a client to selectively choose which objects it wishes to reliably receive. For example, the info content of objects transmitted by an MdpServer application may contain information the client application can use as criteria to make a decision on whether to reliably receive (or receive at all) particular MdpObjects. The MdpObject sender MdpNodeId could also be used as criteria. Other MDP API calls including MdpObjectGetSourceNodeId(), MdpObjectSetNackingMode(), and MdpSessionAbortRxObject() may be used in conjunction with this function to achieve desired application behavior.
By default, MdpSessions will operate with a MdpNackingMode of MDP_NACKING_NORMAL, unless this function is called. The client MdpNackingMode may be changed at any time and will apply on future newly-received or entirely missing objects. Partially-received MdpObjects will retain the MdpNackingMode behavior prior to the call to MdpSessionSetClientNackingMode(). The application can manage the nacking behavior of these receive pending objects separately with the MdpObjectSetNackingMode() function.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. MDP_ERROR_SESSION_INVALID is returned otherwise.

Function MdpSessionSetClientStreamIntegrity()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetClientStreamIntegrity(
    MdpSessionHandle   sessionHandle,
    bool              state);

DESCRIPTION
This function controls the client’s behavior with regard to requesting repair for MdpObjects “missing” in the sequential series of MdpObjects transmitted by MdpServer nodes. When the value of state is true, the client will request retransmission of MdpObjects it detects “missing” in the series of objects received. This behavior guarantees that every MdpObject transmitted by an MdpServer will be received by the client. If the value of the state parameter is false, “stream integrity” is disabled such that the client will only request repairs for MdpObjects for which it has received at least some content (in the form of an MDP_INFO, MDP_DATA, or MDP_PARITY message). However, if the MdpServer subsequently requests positive acknowledgement for a missing object, the client will not acknowledge, even if “stream integrity” is disabled.

When client “stream integrity” is disabled, the probability of reception of MdpObjects of small size becomes very dependent on the packet loss characteristics of the network. By default, client “stream integrity” is enabled, and it is only recommended that it be disabled to meet very special application requirements.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. MDP_ERROR_SESSION_INVALID is returned otherwise.
Function  MdpSessionSetClientMulticastAcks()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetMulticastAcks(
    MdpSessionHandle sessionHandle,
    bool state);

DESCRIPTION
This function controls how MDP ACK messages are transmitted by a client. Normally, ACK
messages are unicast directly to the server node(s) since there is currently no benefit to group-
wide reception of ACK messages. However, in cases where unicast routing is unavailable
(and we have a network like that!) or where, because of asymmetry, it is important that ACKs
are transmitted to the MdpSession destination address instead of the server source address(es),
this function can be used to achieve that capability. The sessionHandle parameter
indicates the applicable MdpSession and the state parameter set to a value of true causes
the client to send ACK messages to the MdpSession destination address (which may be a
multicast or unicast address), while a value of false causes the client to send NACK messages
to the server source address(es).

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success.
MDP_ERROR_SESSION_INVALID is returned otherwise.

Function  MdpSessionSetArchiveDirectory()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetArchiveDirectory(
    MdpSessionHandle sessionHandle,
    const char* path);

DESCRIPTION
This function sets the file name “path” to indicate the directory to which the MDP client
should store received MdpObjects of type MDP_OBJECT_FILE. This “path” must be set for
an MDP client application to receive files. However, if the application is only handling
MdpObjects of type MDP_OBJECT_DATA, it is not necessary to set the archive directory
path. The sessionHandle parameter indicates the applicable MdpSession and the path
parameter is a pointer to a string (PATH_MAX maximum length) containing the name of the
directory in which to store received files. The behavior of how received files are names
and/or overwritten is governed by the “archive mode” (see
MdpSessionSetArchiveMode()).

Note: In the future, application-specific features such as file archiving/cacheing will be
removed from the MDP protocol engine core code base. As a transport protocol,
application-specific features such as this are better handled by applications and the MDP
protocol engine will provide more general handling of transport objects, likely requiring
applications to be responsible for file I/O. The current MDP code contains these functions as
a result of the history of its development.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. If the MDP protocol engine detects
it is unable to write to the indicated directory, an error code of MDP_ERROR_FILE_OPEN is
returned and MDP_ERROR_SESSION_INVALID is returned if the sessionHandle is not valid.

Function MdpSessionSetArchiveMode()

SYNOPSIS
#include <mdpApi.h>
MdpError MdpSessionSetArchiveMode(
    MdpSessionHandle    sessionHandle,
    bool                state);

DESCRIPTION
This function is used to determine whether the MDP will treat received files as those to be
permanently stored or as files to be temporarily cached. The sessionHandle parameter
identifies the applicable MdpSession and a value of true enables file archiving while false
disables those features. When file archiving is enabled, two things happen:

1) If the file name in the MDP_INFO packet associated with the file contains directory
   path delimiters, the directory structure will recreated within the local MDP archive
directory.

2) If a file with the same name is subsequently received, MDP will overwrite the first
   file.

When file archiving is disabled (default), the corresponding change in behavior is such that:

1) If the file name in the MDP_INFO packet associated with the file contains directory
   path delimiters, the file name is “flattened” by replacing the directory delimiters with
   an underscore character (’_’). Thus, files to be “cached” are all contained within the
   single, flat archive directory.

2) When a file with a matching name is received, the new file is given a different name
   (a concatenation of a temporary file name from the system and the files actual name,
   so that the file’s extension is preserved).
See the “Note” under the description of MdpSessionSetArchiveDirectory() concerning the longevity of this type of functionality within the MDP transport protocol code base.

**RETURN VALUES**

A value of MDP_ERROR_NONE is returned upon success or MDP_ERROR_SESSION_INVALID is returned if the sessionHandle is not valid.

**Function** MdpSessionSetClientEmcon()

**SYNOPSIS**

```
#include <mdpApi.h>

MdpError MdpSessionSetClientEmcon(
    MdpSessionHandle sessionHandle,
    bool state);
```

**DESCRIPTION**

This function enables or disables an MDP client from operating in an EMCON (emissions-controlled, or uni-directional server->client transmission) mode of operation. The sessionHandle parameter identifies the applicable MdpSession and the state parameter enables EMCON operation if set to a value of true or disables it if set to a value of false. When EMCON operation is enabled, the following changes in MDP client behavior occur:

1) The client will not make any packet transmissions. Thus in EMCON mode, the client is entirely dependent upon robust transmissions by the server (through use of “auto parity” or redundant object transmission) as the client is prevented from sending NACK messages, ACK messages or any other kind of report. Therefore, servers should be carefully configured when operating in MdpSessions where clients may be operating in EMCON mode. Note that a mixture of clients operating with EMCON operation enabled or disabled is acceptable.

2) The MDP client changes the way it manages memory for buffering received objects. In normal (NACK-based) operation, an MDP client will buffer older received data in favor of new data when buffer space is limited under the presumption that the server will eventually respond to its NACK messages with repair (or send a SQUELCH command if repair of the object in question is no longer supported). However, when EMCON operation is enabled, the client buffers new data in favor of old data under the presumption that the server will not be retransmitting the old object data (since the client is unable to request repairs).

These capabilities allow for rudimentary support of EMCON operation among MDP servers and clients within an MdpSession. As concepts of operation for EMCON mature, the API and functionality in support of this capability will evolve.

**RETURN VALUES**
A value of MDP_ERROR_NONE is returned upon success or MDP_ERROR_SESSION_INVALID is returned if the sessionHandle is not valid.

6.0 MdpObject Management Functions

This section describes functions related to controlling and retrieving information concerning MdpObjects which are being transmitted or received. Most of these functions are most applicable during the reception of MdpObjects and are intended for use within the MdpNotifyCallback which the application installs for the MDP protocol engine. In general, the state referenced by MdpObjectHandles within the API is only valid during the actual transport of an MdpObject. For transmitted objects, this is from the time the object is queued for transmission until the MDP_NOTIFY_TX_OBJECT_FINISHED (or MDP_NOTIFY_OBJECT_DELETE) for the corresponding MdpObjectHandle notification occurs. For received objects, the MdpObjectHandle is valid from the time the MDP_RX_OBJECT_START notification occurs until the MDP_NOTIFY_RX_OBJECT_COMPLETE (or MDP_NOTIFY_OBJECT_DELETE) occurs. Use of these MDP API calls at inappropriate times can lead to a system error.

<table>
<thead>
<tr>
<th>General routines:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MdpObjectGetType()</td>
<td>Retrieves MdpObject type (MDP_DATA_OBJECT or MDP_FILE_OBJECT)</td>
</tr>
<tr>
<td>MdpObjectGetInfo()</td>
<td>Copies payload of MDP_INFO data associated with a given object.</td>
</tr>
<tr>
<td>MdpObjectGetSize()</td>
<td>Retrieves size of object in bytes.</td>
</tr>
<tr>
<td>MdpObjectGetRecvBytes()</td>
<td>Retrieves number of bytes currently received for a given object.</td>
</tr>
<tr>
<td>MdpObjectGetTransportId()</td>
<td>Retrieves the 32-bit transport identifier for an MdpObject.</td>
</tr>
<tr>
<td>MdpObjectGetSourceNodeId()</td>
<td>Retrieves the MdpNodeId of the source of an MdpObject being received.</td>
</tr>
<tr>
<td>MdpObjectSetNackingMode()</td>
<td>Set the client nacking behavior for the indicated MdpObject.</td>
</tr>
</tbody>
</table>

| MDP_DATA_OBJECT routines:    |                                                                 |

Table Q - Summary of MdpObject Management Functions
Function MdpObjectGetType()

SYNOPSIS
#include <mdpApi.h>

MdpObjectType MdpObjectGetType(MdpObjectHandle objectHandle);

DESCRIPTION
This function returns the object type for MdpObject indicated by the objectHandle parameter. The types currently supported include:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDP_DATA_OBJECT</td>
<td>Static memory-resident block of data to be transported by the MDP protocol.</td>
</tr>
<tr>
<td>MDP_FILE_OBJECT</td>
<td>MdpObject representing a disk-based file to be transported by the MDP protocol.</td>
</tr>
<tr>
<td>MDP_SIM_OBJECT</td>
<td>Null object used for representation of transport objects of different sizes for simulations conducted with the ns-2 or OPNET network simulation models supported in the MDP code base.</td>
</tr>
</tbody>
</table>

RETURN VALUES
The type of MdpObject is returned. If an invalid MdpObjectHandle is used, a system error may occur. Thus, caution care must be taken to use this function at only the appropriate time as described in the introduction to this section.

Function MdpObjectGetInfo()

SYNOPSIS
#include <mdpApi.h>
bool MdpObjectGetInfo(MdpObjectHandle objectHandle, 
        char* buffer, 
        unsigned short* bufLen);

DESCRIPTION
This function copies the MDP_INFO (if available) associated with the MdpObject indicated by the objectHandle into the buffer provided by the caller. The value the bufLen parameter points to should be preset to contain the amount of space available (in bytes) in the buffer when the function is called. Upon return, the bufLen parameter will be adjusted to how many bytes were copied. Note: A function will be added to the API in the future so that the application may determine how many bytes of MDP_INFO are available for a given object.

RETURN VALUES
A value of true is return if MDP_INFO is available for the object and was copied. A value of false is returned if no MDP_INFO is available.

Function MdpObjectGetSize()

SYNOPSIS
#include <mdpApi.h>

unsigned long MdpObjectGetSize(MdpObjectHandle objectHandle);

DESCRIPTION
This function returns the total size (in bytes) of the data content of the MdpObject. Identified by the objectHandle parameter.

RETURN VALUES
The total size of the MdpObject in bytes is returned. MdpObjects consisting of solely MDP_INFO will return a data content of size zero.

Function MdpObjectGetRecvBytes()

SYNOPSIS
#include <mdpApi.h>

unsigned long MdpObjectGetSize(MdpObjectHandle objectHandle);

DESCRIPTION
This function returns the number of bytes of the data content currently received for the MdpObject. Identified by the objectHandle parameter. This function can be used by an application to monitor the reception progress of an object in transit. The returned value can
be compared with the value obtained using MdpObjectGetSize() to calculate the percentage of receive completion.

RETURN VALUES
The number of current received bytes for the MdpObject is returned.

Function MdpObjectGetTransportId()

SYNOPSIS
#include <mdpApi.h>

MdpObjectTransportId MdpObjectGetTransportId(MdpObjectHandle objectHandle);

DESCRIPTION
This function returns the MdpObjectTransportId for the MdpObject indicated by the objectHandle parameter. The intended use of this information is for use with the MdpSessionRequeueTxObject() function (this function is not yet implemented or documented) to allow redundant transmission of an MdpObject as part of a robust one-way, EMCON transmission scheme. This function may also be useful for application debugging when used in conjunction with the debug logging of the MDP protocol engine. More information will be provided as EMCON modes of operation for MDP are better defined in the future.

RETURN VALUES
The 32-bit MdpObjectTransportId for the MdpObject is returned.

Function MdpObjectGetSourceNodeId()

SYNOPSIS
#include <mdpApi.h>

MdpNodeId MdpObjectGetSourceNodeId(MdpObjectHandle objectHandle);

DESCRIPTION
This function returns the MdpNodeId of the source of the MdpObject indicated by the objectHandle parameter. The returned MdpNodeId is guaranteed to be valid only for the time during which MdpObjectHandles of MdpObjects being received from that source are valid (i.e. until MDP_NOTIFY_OBJECT_DELETE notifications are received for all MdpObjects pending reception from the MdpServer).

RETURN VALUES
The 32-bit MdpNodeID for the source of the MdpObject is returned. If the objectHandle value corresponds to an MdpObject being transmitted by the local MdpServer, a value of MDP_NULL_NODE is returned.

Function MdpObjectSetNackingMode()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpObjectSetNackingMode(MdpObjectHandle objectHandle, MdpNackingMode mode);

DESCRIPTION
This function controls the nacking behavior for a partially-received MdpObject indicated by the objectHandle parameter. Possible values for the MdpNackingMode include MDP_NACKING_NORMAL, MDP_NACKING_INFOONLY, and MDP_NACKING_NONE.

Setting the nacking behavior of individual receive MdpObjects allows the client application to customize its operation in an MDP multicast session. If receivers know they are not interested in reliably receiving some portion of the content of a transmitted object, they can reduce the overall level of traffic generated by the group by tuning their repair request behavior with this function call. If the client application is not at all interested (even unreliably) in receiving the indicated object, it should use the MdpSessionAbortRxObject() function.

The default MdpNackingMode for newly-received objects is determined from the client’s MdpNackingMode value. See the MdpSessionSetClientNackingMode() description for more discussion on usage of these function calls.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. MDP_ERROR_OBJECT_INVALID is returned when the objectHandle parameter is MDP_NULL_OBJECT, but the application should take great care not to pass invalid MdpObjectHandle’s to this (or other MdpObject) function calls.

Function MdpObjectSetData()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpObjectSetData(MdpObjectHandle objectHandle, char* dataPtr, unsigned long dataSize);
DESCRIPTION

This function assigns a memory buffer for reception of an MdpObject of type MDP_DATA. This function can only be used for receive objects of type MDP_DATA. The appropriate use of this function is to assign a buffer to use for data reception when the MDP_NOTIFY_RX_OBJECT_START notification occurs. If a buffer is not assigned at this time, the MDP protocol will assume that the application does not wish to receive the object in question and will stop attempting to receive the data (state for the object will be deleted and no request for repairs will be made). The dataPtr parameter points to the block of memory in which to store the received MdpObject data content and the dataSize parameter is used to indicate the size of the memory block. If the value of the dataSize parameter is not of sufficient size to store the MdpObject, an error will be returned. Therefore, the application should first use the MdpObjectGetSize() function to determine the size of the object’s data content. The application must not free this memory during object transport. Since the application is responsible for allocating and assigning the buffer space used for data reception, it is up to the application to manage this memory space after reception of the object has completed (or upon the MDP_NOTIFY_OBJECT_DELETE notification which occurs for both receive object completion or failure).

RETURN VALUES

A value of MDP_ERROR_NONE is returned upon success MDP_ERROR_OBJECT_INVALID is returned if the indicated object is not of type MDP_DATA_OBJECT and MDP_ERROR is returned if the dataSize is insufficient.

Function MdpObjectGetData()

SYNOPSIS

#include <mdpApi.h>

char* MdpObjectGetData (MdpObjectHandle objectHandle, 
        unsigned long* dataSize, 
        MdpError* error);

DESCRIPTION

This function returns a pointer to the location of data content storage for an MdpObject of type MDP_DATA identified by the objectHandle parameter. The length (in bytes) of the object is copied to the location indicated by the dataSize parameter. This function is useful for applications wishing to take over management of the memory space after an MDP_NOTIFY_OBJECT_DELETE notification occurs or to make use of a received data object after the MDP_NOTIFY_RX_OBJECT_COMPLETE notification occurs. Note that the management of the memory space for objects of MDP_DATA_OBJECT is up to the application much the way the use of files sent or received is up to the application after reliable transport is complete.
RETURN VALUES
A pointer to the object data content storage is returned upon success. Otherwise, a value of NULL is returned and the error parameter is filled in with an appropriate MdpError code. This function is only applicable to MdpObjects of type MDP_DATA_OBJECT.

Function MdpObjectGetFileName()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpObjectGetFileName(MdpObjectHandle objectHandle, char* name, int maxlen);

DESCRIPTION
This function copies the full file name for an object of type MDP_FILE_OBJECT. For received objects, this full name includes the path of the archive directory (see MdpSessionSetArchiveDirectory()) as well as the name received in the MDP_INFO content, if available. If no MDP_INFO was available for the object, the temporary name assigned by the MDP protocol engine is returned. This function may also be used for transmit objects if the application wishes. Again, the full name of the file is returned.

RETURN VALUES
A value of MDP_ERROR_NONE is returned upon success. This function is only applicable to MdpObjects of type MDP_FILE_OBJECT. A value of MDP_ERROR_OBJECT_INVALID is returned if this function is used on objects of other types.

7.0 Miscellaneous Functions
This section describes a number of functions which serve peripheral uses in the MDP API and do not clearly fit into the previous categories.

Function MdpSessionSetRecvDropRate()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetRecvDropRate(MdpSessionHandle sessionHandle, double percent);
The MdpSessionSetRecvDropRate() function allows the application to specify the percent of artificially generated received packet loss (0-100%) in the MdpSession identified by sessionHandle. This is useful for testing purposes to simulate uncorrelated packet loss among receiving nodes. The default is zero artificial packet loss if this function is never called.

RETURN VALUES
The MdpSessionSetRecvDropRate() function will return a value of MDP_ERROR_NONE if successful. The value MDP_ERROR_SESSION_INVALID is returned upon failure.

Function MdpSessionSetSendDropRate()

SYNOPSIS
#include <mdpApi.h>

MdpError MdpSessionSetSendDropRate(MdpSessionHandle sessionHandle,
                                double percent);

DESCRIPTION
The MdpSessionSetSendDropRate() function is used to specify the percent of artificially generated transmit packet drops (0-100%) to simulate correlated packet loss among receiving nodes for the MdpSession identified by the sessionHandle parameter. This is useful for protocol testing purposes. The default is zero artificial packet loss if this function is never called.

RETURN VALUE
The MdpSessionSetSendDropRate() function will return a value of MDP_ERROR_NONE if successful. The value MDP_ERROR_SESSION_INVALID is returned upon failure.