Revision History

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20-Sep-13</td>
<td>Addition of C++ API</td>
</tr>
<tr>
<td>3</td>
<td>22-Feb-13</td>
<td>Revised Java jar definitions and minor API name changes.</td>
</tr>
<tr>
<td>2</td>
<td>10-Jan-13</td>
<td>Updated to include support for Ro and Rf interfaces.</td>
</tr>
<tr>
<td>1</td>
<td>09-Nov-12</td>
<td>Initial Release for use during Dialogic DSI Diameter Stack beta trial.</td>
</tr>
</tbody>
</table>

Note: The current version of this guide can be found at: http://www.dialogic.com/support/helpweb/signaling

Contents

Revision History ................................................................................................................. 3

1  Introduction .................................................................................................................. 5
   1.1 Applicability ........................................................................................................... 5
   1.2 Related Documentation .......................................................................................... 5

2  Stack Overview ............................................................................................................. 6

3  Component Overview .................................................................................................... 7
   3.1 Java Development Package Components .................................................................. 7
   3.2 C++ Development Package Components .................................................................. 8
   3.3 Message Passing Environment (GCTLIB) API ......................................................... 8
   3.4 DMR Access API ..................................................................................................... 8
       3.4.1 DmrContext ..................................................................................................... 8
       3.4.2 DMR UserApi and Message Encoders .............................................................. 9
   3.5 Diameter Command API .......................................................................................... 10

4  Functional API Illustration ........................................................................................ 12
   4.1 Generating a Diameter Session ............................................................................. 12
       4.1.1 Building the session/request object ............................................................... 13
   4.2 Handling a received Diameter Request ................................................................. 14
   4.3 AVP Handling ......................................................................................................... 15
       4.3.1 Adding and Retrieving an AVP ....................................................................... 15
       4.3.2 Adding an AVP list .......................................................................................... 15
       4.3.3 Retrieving an AVP list .................................................................................... 15
       4.3.4 Adding an unknown AVP ................................................................................ 16
       4.3.5 Retrieving an unknown AVP .......................................................................... 16
       4.3.6 Handling enum AVPs with unknown values ..................................................... 16
   4.4 Result Code ............................................................................................................ 17
   4.5 Error Command ..................................................................................................... 18
   4.6 Exception Handling ............................................................................................... 18

5  Command and Message Dictionaries .......................................................................... 20
   5.1 DMS Utility ............................................................................................................ 21
   5.2 Namespace ............................................................................................................. 22


Section 1 Introduction

6 Example Java Applications ................................................................. 23
6.1 DTU Diameter Test Utility .................................................................. 23
   6.1.1 Command line output .................................................................. 26
   6.1.2 Command line examples .............................................................. 27
6.2 DTR Diameter Test Responder ............................................................ 27
   6.2.1 Command line output .................................................................. 28
   6.2.2 Command line examples .............................................................. 28

7 Example C++ Applications .................................................................... 29
7.1 Installation ............................................................................................ 29
7.2 Building and running the example code .............................................. 29
7.3 C++ Example Overview ........................................................................ 30
   7.3.1 Generating a Diameter Session .................................................. 30
   7.3.2 Handling a received Diameter Request ....................................... 31

Figures
Figure 1. Diameter Module and Functional APIs ........................................ 6
Figure 2. Functional APIs .......................................................................... 7
Figure 3. DmrContext Static Class Hierarchy .......................................... 9
Figure 4. Diameter Command API Static Class Hierarchy ....................... 10
Figure 5. DMS Class generation and encoders ........................................ 20
Figure 6. DTU/DTR Message Sequence Chart (Mode 0 – Update Location) 23
Figure 7. DTU/DTR Message Sequence Chart (Mode 1 – Credit Control) ... 24

Tables
Table 1. Result Code Values ..................................................................... 17
1 Introduction

The Dialogic® DSI Diameter Stack is a software implementation of the IETF Diameter Base Protocol which is intended to facilitate development of user applications that interface to LTE and IMS networks for the implementation of services in the areas of: Mobility, Online Charging and Offline Charging.

The Dialogic® DSI Diameter Stack includes a message based binary Diameter Module, a Functional API Library and utility components and header files for use when developing a User Application.

Dialogic's Diameter Module (DMR) implements the Diameter Base Protocol offering a message based API to the User Application to control Diameter sessions. DMR is a member of the family of Dialogic® DSI Components and offers similar message-based interfaces and management capabilities to those offered for other SS7 and SIGTRAN protocol layers. DMR uses the services provided by the SCTP layer of the Dialogic® DSI SIGTRAN Stack for the transfer of messages between Diameter Peers.

Dialogic’s DMR Functional API is for use within user applications interfacing with the DMR module. The messages defined by DMR as part of its user interface contain network formatted Diameter commands. The functional API allows for the easy encoding and decoding of these DMR messages to support quick application development and maintenance.

This manual is intended for use by Application Developers who intend to write or maintain applications which use the Functional API for the Dialogic® DSI Diameter Stack. It provides an overview of the Functional API and other related software components. Users needing to configure and maintain the Diameter Module (DMR) should refer to the Diameter Programmer’s Manual.

1.1 Applicability

This manual is applicable to the following software:

Dialogic® DSI Development Package for Solaris – Release 5.3.1 or later (Java API only – check for availability of C++ API).

Dialogic® DSI Development Package for Linux – Release 6.6.1 or later

1.2 Related Documentation

Current software and documentation supporting Dialogic® DSI components is available at: http://www.dialogic.com/support/helpweb/signaling

Video tutorials are available at: http://www.dialogic.com/den/media

The following User Documentation relates to the use of the Dialogic® DSI Diameter Stack:

- Dialogic® DSI Diameter Stack - Diameter Functional API Manual
- Dialogic® DSI SIGTRAN Stack - SCTP Programmer’s Manual
2 Stack Overview

There are a number of components supplied as part of the Dialogic® DSI Diameter Stack. These include APIs for interfacing to the Dialogic message passing environment in general as well as APIs specific for use with the Diameter Module within the Dialogic® DSI Diameter Stack.

The User Application will be built using the supplied Functional APIs described in this manual. These APIs facilitate the generation and handling of messages sent to or received from the Diameter Module.

The Diameter Module (DMR) implements the Diameter Base Protocol offering a message based API to the User Application to control Diameter sessions and is discussed further in the DMR Programmer’s Manual.

The Diameter Module can be configured via messages using a Management Module. This Management Module may be the s7_mgt utility included within the DSI Development Package or optionally it may be replaced by a User generated module.

The Diameter Module interfaces to the network via the SCTPN module to send the appropriate payload messages and for control of SIGTRAN associations.

Figure 1. Diameter Module and Functional APIs
### 3 Component Overview

This section provides an overview of the functional APIs used by an Application to interface with the Diameter Module (DMR). The APIs are shown below in Figure 2.

The Message Passing API allows C++ and Java users to interface with the underlying DSI message-passing mechanism by providing access to the GCTLIB library. The DMR Access API provides the methods to control and manage the messages that access the functionality provided in the Diameter Module. The Diameter Command API provides functionality to build Diameter Command and AVP objects to permit them to be encoded using the DMR Access API.

![Diagram of functional APIs](image)

**Figure 2. Functional APIs**

### 3.1 Java Development Package Components

The required functionality and components for use in Java are stored in the following Java Jar files within the DSI Development Package.

<table>
<thead>
<tr>
<th>API</th>
<th>Package</th>
<th>Jar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Passing Library</td>
<td>com.dialogic.signaling.gct</td>
<td>gctApi.jar</td>
</tr>
<tr>
<td>DMR Access API</td>
<td>com.dialogic.signaling.dmr</td>
<td>dmrApr.jar</td>
</tr>
<tr>
<td>Diameter Command API</td>
<td>com.dialogic.signaling.diameter</td>
<td>dmrApr.jar</td>
</tr>
<tr>
<td></td>
<td>com.dialogic.signaling.diameter.*</td>
<td>dmrCmds.jar</td>
</tr>
</tbody>
</table>
### 3.2 C++ Development Package Components

The required functionality and components for use with C++ are stored in the following files within the DSI Development Package.

<table>
<thead>
<tr>
<th>API</th>
<th>Package/Namespace</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Passing Library</td>
<td>gctlib</td>
<td>libcctlib.so</td>
</tr>
<tr>
<td>DMR Access API</td>
<td>com_dialogic_signaling::dmr</td>
<td>dmrApr.lib</td>
</tr>
<tr>
<td></td>
<td>com_dialogic_signaling::dmr::user</td>
<td></td>
</tr>
<tr>
<td>Diameter Command API</td>
<td>com_dialogic_signaling::diameter</td>
<td>dmrApr.lib</td>
</tr>
<tr>
<td></td>
<td>com_dialogic_signaling::diameter::*</td>
<td>dmrCmds.lib</td>
</tr>
</tbody>
</table>

### 3.3 Message Passing Environment (GCTLIB) API

The functional API is supplied together with Java extensions to the DSI Signaling Products Development Package to support access to the 'C' based GCTLIB message-passing library. C++ users can make direct use of the existing GCTLIB library. The message passing environment is the same environment as used by the Diameter Module and most parts of the Development Package. This is shown in Figure 2.

### 3.4 DMR Access API

These are a set of classes which enable Diameter encoding/decoding, session handling and co-ordination independent of any individual Diameter message or specific encoding rules.

#### 3.4.1 DmrContext

The focus of the DmrContext is APIs to encode and decode messages sent to and from the Diameter Module. This includes messages defined within the DMR Programmer’s Manual and also Diameter requests and answers embedded within these messages. This class may be considered the top level class for your application interfacing with the Diameter module.

The DmrContext class is also used to set-up and define settings and parameters appropriate to a Diameter user. This means it can be created once and used multiple times to encode or decode Diameter Command Requests and Answers using the UserApi class it contains.

The DmrContext class includes a method to permit Diameter Command API classes to be registered with that context object. This then permits later calls to the UserAPI encode/decode methods of a DmrContet object to be made.
3.4.2 **DMR UserApi and Message Encoders**

This class provides access methods for the encoders and decoder classes that perform the conversion between instances of the message classes and a well formed GCT message.

**DMRSessionIndEncoder**

Provides encode, decode and length calculation for DMR Session Indication GCT messages.

**DMRSessionReqEncoder**

Provides encode, decode and length calculation for DMR Session Indication GCT messages.
3.5 **Diameter Command API**

This component provides the classes to encode and decode specific Diameter commands and AVPs. These are derived from a definition dictionary to permit extensions and modifications to be supported.

For the Java these are packaged in com.dialogic.signaling.diameter.<specname>.

For the C++ these use the namespace com_dialogic_signaling::diameter::<specname>.

For example com.dialogic.signaling.diameter.rfc3588 or com_dialogic_signaling::diameter::rfc3588.

Shown in Figure 4 is a simplified overview of the class hierarchy of the Diameter Command API with a single command (UpdateLocationRequest) shown. Each specific command request or answer required for an application should be registered against the DmrContext object being used.

![Diameter Command API Static Class Hierarchy](image)

**Figure 4. Diameter Command API Static Class Hierarchy**

**IDiameterCommand**

This is an interface representing a Diameter Command. This class includes attributes for the Diameter Command Header such as Command Code, Hop by Hop Id, it extends from IAvpList.
**IAvpList**
Abstract interface which stores a list one or more AVPs.

**IAvp**
Abstract class that models a single AVP object.

**GroupedAvp**
Models an AVP which contains one more other AVPs.

**OctetStringAvp**
Standard primitive AVP representing an array of octet data. Other primitive AVP types exist.

**Utf8StringAvp**
Pre-defined primitive AVP representing an array of UTF8 data – sub-classed from OctetStringAvp.

**ProxyInfoAvp**
An example GroupedAvp which has at least one sub-AVP.

**ProxyHostAvp**
An example member of the ProxyInfoAvp GroupedAvp.

**UpdateLocationRequest**
An example of a concrete Diameter Command class used to encode or decode an Update Location Request. Other classes exist for other commands. It provides the real methods for adding the correct AVPs to the command or retrieving AVPs on decode.
4 Functional API Illustration

This section provides a description of the various API calls appropriate to permit development of applications making use of the Functional API. Unless otherwise stated the source code examples are for Java. For C++ Users further example code can be found in Section 7 - Example C++ Applications.

4.1 Generating a Diameter Session

The following code samples and description show a simplified example of building and sending a Diameter Update Location Request. Exception handling and other error handling has been removed to show the core functionality more closely.

```java
//Step 1
dmrContext = new DmrContext();

//Step 2
dmrContext.registerDiameterCommand(UpdateLocationRequest.class);
dmrContext.registerDiameterCommand(UpdateLocationAnswer.class);
dmrContext.setDmrTaskId(config.DstMID);
dmrContext.setSrcTaskId(config.SrcMID);

//Step 3
DmrSessionReq dmrSsnReq = DmrSessionReqFactory.buildDmrSessionReq(dmrContext, Config);
dmrSsnReq.SessionId = dmrSsnReq.SessionId % Config.MaxSsnNum;

//Step 4
int len = dmrContext.userApi.dmrSessionReqEncoder.calculateLength(dmrSsnReq);
GctMsg gctMsg = GctLib.getm(len);

//Step 5
dmrContext.userApi.dmrSessionReqEncoder.encode(dmrSsnReq, gctMsg);

//Step 6
GctLib.send((short) Config.DstMID, gctMsg);
```

**Step 1: Generate a diameter context**
The diameter context object is used to store and initialize factory and encoder objects required by the user application.

**Step 2: Register a command with the context**
In order to define which commands are valid with the diameter context, the required requests and answer commands need to be registered into the context object. You can also set the module ids for the application and Diameter module here too.
Step 3: Create and populate the session/request object
This step builds up a structured form of the request object which is to be encoded and is covered in further detail in section 4.1.1 Building the session/request object. This uses methods from the Diameter Command Specific API.

Step 4: Allocate a GCT message
From the context object, a method can be used to calculate the encoded length for the message, allowing the GCT message structure to be allocated.

Step 5: Encode the command request into a GCT message
The context object also contains the encoder method to format the DmrSessionReq into a GCT message.

Step 6: Send the GCT Message
Takes the GCT message and sends it to the DSI Diameter Module

4.1.1 Building the session/request object
The diameter context object has factory methods that allow a session request object to be built. The request can then have the Network Context, Session Id and Primitive type values set as appropriate to the request as shown below.

```java
DmrSessionReq req = new DmrSessionReq();
req.primitiveType = req.primitiveType.OPEN;
req.diameterCommand = UpdateLocationRequestFactory.buildUpdateLocationRequest(dtuConfig);
req.sessionId = sessionId;
return req;
```

As shown above the example method UpdateLocationRequestFactory includes calls to create a new Update Location Request. This object has a number of methods one to add each of the AVPs included in the request. An example of this is shown below.
4.2 Handling a received Diameter Request

```java
//Step 1
dmrContext = new DmrContext();

//Step 2
dmrContext.registerDiameterCommand(UpdateLocationRequest.class);
dmrContext.registerDiameterCommand(UpdateLocationAnswer.class);
dmrContext.setDmrTaskId(config.DstMID);
dmrContext.setSrcTaskId(config.SrcMID);

//Step 3
GctMsg rxedMsg = GctLib.receive(Config.SrcMID);

//Step 4
if (rxedMsg.getType() ==
dmrContext.MsgType.DMR_MSG_SESSION_IND.getValue()) {
    DmrSessionInd decodedInd =
        dmrContext.UserApi.DmrSessionIndEncoder.decode(gctmsg);
}

//Step 5
if (ind.DiameterCommand.getCommandCode() ==
UpdateLocationRequest.StandardCommandCode) {
    UpdateLocationRequest ulr = (UpdateLocationRequest) ind.DiameterCommand;
    try {
        System.out.println("Command code matches");
        System.out.println(ulr.getOriginHostAvp().getString());
        System.out.println(ulr.getOriginRealmAvp().getString());
    } catch (UnsupportedEncodingException ex) {
        System.out.println("Failed to recover AVP" + ex.toString());
    }
}
```

**Step 1: Generate a diameter context**

The diameter context object is used to store and initialize factory and encoder/decoder objects required by the user application.

**Step 2: Register a command with the context**

In order to define which commands are valid with the diameter context the required requests and answer commands need to be registered into the Context Object.

**Step 3: Receive the GCT message**

This call will block until a message for the requested module ids is returned.
Step 4: Decode the Session Indication
As with encoding functionality, the diameter context object provides methods which will decode the GCT message into a structured object representing the indication.

Step 5: Decode the Command
The command code of the received indication can be compared against the expected command code and used to create a command object of the correct class. The various getXYZAvp() methods that are supplied can then be used.

4.3 AVP Handling

4.3.1 Adding and Retrieving an AVP
For AVPs at the top level of a Diameter command, there are a series of methods which allow AVPs to be added or retrieved. In the example, below a new OriginHostAvp object is created and added to an Update Location Request object.

```java
ulr.addOriginHostAvp(new OriginHostAvp("OriginHost"));
```

There is a matching getOriginHostAvp() method for retrieving the values – in this case returning an OriginHostAvp object.

4.3.2 Adding an AVP list
AVPs which are permitted to have multiple instances within a single command request or answer can be supported by calling the appropriate AVP addXYXAvp() multiple times.

```java
//First Subscriber Data
SubscriptionDataAvp sda = new SubscriptionDataAvp();
sda.addAccessRestrictionDataAvp(new AccessRestrictionDataAvp((long) 0x55555));
ula.addSubscriptionDataAvp(sda);

//Second Subscriber Data
SubscriptionDataAvp sda2 = new SubscriptionDataAvp();
sda2.addAccessRestrictionDataAvp(new AccessRestrictionDataAvp((long) 0x44444));
ula.addSubscriptionDataAvp(sda2);
```

4.3.3 Retrieving an AVP list
When retrieving an AVP which may have multiple instances, it is necessary to use an iterator. AVPs which may have multiple instances have two forms of the getters, one with and one without an iterator parameter. The form without the iterator will return the first instance of that AVP. The form with the iterator will return the first instance of the AVP the first time it is called and then the next each extra time it is called.
ListIterator<IAvp> li = ulr.listIterator();

while (li.hasNext()) {
    ProxyInfoAvp pia = ulr.getProxyInfoAvp(li);
    if ((pia != null) && (pia.getProxyHostAvp() != null)) {
        System.out.println( pia.getProxyHostAvp().getString());
    }
}

4.3.4 Adding an unknown AVP

To add an unknown AVP, a new AVP needs to be created of the appropriate AVP type. The unknown AVP can then be marked as unknown with the setIsUnknown() method and added using the add() method on the request or answer.

IAvp unknownAvp = new OctetStringAvp(999L, 0L, ByteBuffer.wrap(new byte[]{1, 2, 3, 4}) );
ulr.add(unknownAvp);

4.3.5 Retrieving an unknown AVP

When retrieving one or more unknown AVPs, it may be necessary to use an iterator. There may be one or more unknown AVPs; therefore, there are two forms of getters: one with and one without an iterator parameter. The form without the iterator will return the first instance of an unknown AVP. The form with the iterator will return the first instance of an unknown AVP the first time it is called and then the next each extra time it is called.

ListIterator<IAvp> li = ulr.listIterator();

while (li.hasNext()) {
    IAvp ua = ulr.getUnknownAvp(li);
    if (ua != null){
        System.out.println("Unknown Avp:" + ua.getCode());
    }
}

4.3.6 Handling enum AVPs with unknown values

If enumerated values are encountered that do not correspond to explicitly defined enumerated values, then their value can be returned by integer value instead.
System.out.println("Unknown enum val:"); ulr.getUESrvccCapabilityAvp().getInteger();

4.4 Result Code
Diameter Answers include the ResultCode AVP. This AVP is an Integer value indicating the success or failure code of the Diameter Command. The commands support `getResultCodeAvp()` and `addResultCodeAvp()` methods to allow the value to be retrieved from an Answer Indication.

In addition, there is an enumerated type which can be used to generate standard values.

```
ResultCode.DIAMETER_SUCCESS.getValue();
```

Values valid for this AVP are included in the table below.

Table 1. Result Code Values

<table>
<thead>
<tr>
<th>Result Code Mnemonic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAMETER_MULTI_ROUND_AUTH</td>
<td>1001</td>
</tr>
<tr>
<td>DIAMETER_SUCCESS</td>
<td>2001</td>
</tr>
<tr>
<td>DIAMETER_LIMITED_SUCCESS</td>
<td>2002</td>
</tr>
<tr>
<td>DIAMETER_COMMAND_UNSUPPORTED</td>
<td>3001</td>
</tr>
<tr>
<td>DIAMETER_UNABLE_TO_DELIVER</td>
<td>3002</td>
</tr>
<tr>
<td>DIAMETER_REALM_NOT_SERVED</td>
<td>3003</td>
</tr>
<tr>
<td>DIAMETER_TOO_BUSY</td>
<td>3004</td>
</tr>
<tr>
<td>DIAMETER_LOOP_DETECTED</td>
<td>3005</td>
</tr>
<tr>
<td>DIAMETER_REDIRECT_INDICATION</td>
<td>3006</td>
</tr>
<tr>
<td>DIAMETER_APPLICATION_UNSUPPORTED</td>
<td>3007</td>
</tr>
<tr>
<td>DIAMETER_INVALID_HDR_BITS</td>
<td>3008</td>
</tr>
<tr>
<td>DIAMETER_INVALID_AVP_BITS</td>
<td>3009</td>
</tr>
<tr>
<td>DIAMETER_UNKNOWN_PEER</td>
<td>3010</td>
</tr>
<tr>
<td>DIAMETER_AUTHENTICATION_REJECTED</td>
<td>4001</td>
</tr>
<tr>
<td>DIAMETER_OUT_OF_SPACE</td>
<td>4002</td>
</tr>
<tr>
<td>DIAMETER_ELECTION_LOST</td>
<td>4003</td>
</tr>
<tr>
<td>DIAMETER_AVP_UNSUPPORTED</td>
<td>5001</td>
</tr>
<tr>
<td>DIAMETER_UNKNOWN_SESSION_ID</td>
<td>5002</td>
</tr>
<tr>
<td>DIAMETER_AUTHORIZATION_REJECTED</td>
<td>5003</td>
</tr>
<tr>
<td>DIAMETER_INVALID_AVP_VALUE</td>
<td>5004</td>
</tr>
<tr>
<td>DIAMETER_MISSING_AVP</td>
<td>5005</td>
</tr>
<tr>
<td>DIAMETER_RESOURCES_EXCEEDED</td>
<td>5006</td>
</tr>
<tr>
<td>DIAMETER_CONTRADICTING_AVPVS</td>
<td>5007</td>
</tr>
</tbody>
</table>
4.5 Error Command

The Diameter Command API includes the ErrorCommand class to represent Error notifications sent in response to Command Requests. These are built, encoded and decoded in the same way as other command objects with the exception that they do not need to be registered against the DmrContext object.

The addresultCodeAvp() method can be used to set the result code on an Error Command to send. For received Error Commands, they can be checked to confirm they are really ErrorCommand objects and then processed appropriately.

```java
if (ind.diameterCommand.getCommandCode() == ErrorCommand.StandardCommandCode) {
    ErrorCommand ec = (ErrorCommand) ind.DiameterCommand;
    // Do something with the Error Command 'ec' object,
    // e.g. check the Result Code
}
```

4.6 Exception Handling

There are two main groups of exceptions that may need to be handled by application using the API.

GctException

Generated by the GctLib classes to indicate issues when allocating messages, sending messages, retrieving messages or releasing messages. The class includes a `getMessage()` method to allow further details to be returned as a String.
EncoderException
Generated by the dmrApi classes when encoding or decoding commands. The class includes a getMessage() method to allow further details to be returned as a String.

The example below shows both of these kinds of exceptions being handled for some sample Java:

```java
try {
    DmrSessionReq dmrSsnReq = DmrSessionReqFactory.BuildDmrSessionReq(dmrContext, Config);
    int len = dmrContext.userApi.dmrSessionReqEncoder.calculateLength(dmrSsnReq);
    gctMsg gctMsg = GctLib.getm(len);
    dmrContext.userApi.dmrSessionReqEncoder.encode(dmrSsnReq, gctMsg);
    if (Config.TraceOn) {
        DtuMsgUtil.traceMsg("DTU>>", gctMsg);
    }
    GctLib.send((short) Config.DstMID, gctMsg);
} catch (EncoderException eEx) {
    System.out.println("Problem with Encode/Decode" + eEx.getMessage());
} catch (GctException gctEx) {
    System.out.println("Problem with message handling: " + gctEx.getMessage());
} catch (Exception ex) {
    System.out.println(ex.toString());
}
```
5 Command and Message Dictionaries

The functional API provides a set of class files which are supplied to greatly simplify the generation and handling of specific Diameter commands and parameters. These class files are generated from a set of Dictionary files to allow extensions and modifications to the class sets as new services, interfaces or parameters are required.

Figure 5. DMS Class generation and encoders

In the figure above, the Diameter definition files are used by the DMS utility (part of the Dialogic® DSI Diameter Stack) to generate encoder libraries for use with the Functional API. This generation happens prior to compiling the application.

At application start-time, the application the Encoder Factory classes can be used to create an instance of the DmrContext class in advance of sessions needing to be handled.
After the initial start-up phase of run-time, the DmrContext User API methods can be used to encode Diameter request or answer objects into GCT Message. The same DmrContext object can also be used to decode messages from GCT Messages into Diameter request or answer objects.

5.1 DMS Utility
The DMS utility is a supporting component to the Dialogic® DSI Diameter Stack. It is used to convert the structured definitions of the Diameter commands and AVPs into class libraries for use with the Functional API.

Syntax

```
java -jar dms [-input <path>] [-output <path>] [-namespace <ns>]
```

Example

```
java -jar //opt/DSI/java/dms.jar -output "src" -l CPP
```

Parameters

- **-input <path>**
  Path to *.dms Diameter specification definition files. Short form is '-i'
- **-output <path>**
  Path to which generated files should be written. Short form is '-o'
- **-namespace <ns>**
  The root namespace for generated class files. The default namespace is com.dialogic.signaling.diameter. Short form is '-n'
- **-language <JAVA|CPP>**
  The required language, default is CPP. Short form is '-l'
- **-quiet**
  Suppress output and don’t ask for user prompts. Short form is '-q'
- **-verbose**
  Display verbose output. Short form is '-verb'
- **-version**
  Display DMS version information. Short form is '-v'
- **-help**
  Display DMS help information. Short form is '-h'
5.2 Namespace

The classes generated by DMS using the XML definition files make use of appropriate package names to differentiate the namespace of the classes. Typically each specification will produce a package with its name derived from the specification title (for example rfc3588).
6 Example Java Applications

6.1 DTU Diameter Test Utility

Description
The Diameter test Utility (DTU) is a simple example application that illustrates the use of the Functional API and can be configured to implement specific operations.

By default, the DTU example application simulates the sending of an Update Location Request Message and waits for the corresponding Update Location Answer. The utility can be used with the DTR example listed in section 6.1.2.

![DTU/DTR Message Sequence Chart](image-url)

Figure 6. DTU/DTR Message Sequence Chart (Mode 0 – Update Location)
DTU can also generate a message sequence flow for demonstration of a Credit Control message flow appropriate for the Ro Diameter interface.

![Diagram of a message sequence flow for Credit Control](image)

**Figure 7. DTU/DTR Message Sequence Chart (Mode 1 – Credit Control)**

**Syntax**

```bash
java -jar dtu [ -m<modid> ] [ -dmr<dmr modid> ] [ -dstr<dest host> ] [ -dstr<dest realm> ] [ -numreq<num requests> ] [ -maxin<max in flight> ] [ -maxssn<max session number> ] [ -basessn<base session id> ] [ -nc<Network Context> ] [ -mode<Mode> ] [ -traceoff ]
```

**Parameters**

- **-m<modid>**
  
  Set to define the module id used by DTU. If not set then DTU will default to 0x1d.

- **-dmr<dmr modid>**
  
  Set to define the module id used by the Diameter module (DMR). If not set then DTU will default to 0x74.
-dsth <dest host>
-dstr <dest realm>
Set the destination host and realm values to be used.
-numreq <num requests>
Defines the total number of requests to send.
-maxin <max in flight>
Set to limit the number of requests that will be outstanding at any one time.
-maxssn <max session number>
Set to define the maximum session number to use.
-base <base session number>
Set to define the base session number to use.
-nc <Network Context>
Set to define the Network Context value to use, defaults to 0.
-mode <Mode value>
Set to 0 for Update Location Request (S6a) example session message flow. This mode is also the default mode.
Set to 1 for Credit Control (Ro) example session message flow.
-traceoff
Disable message tracing
6.1.1 Command line output

Help Screen

DSI DTU Diameter Test Utility Release 1.01
Part of the Dialogic(R) DSI Development Package
Copyright (C) 2012 Dialogic Inc. All Rights Reserved.

Syntax: dtu.jar [-m<> -dmr<> -dsth<> -dstr<>
-umreq<> -maxin<> -maxssn<> -traceoff]
DTU Sends Diameter Session Requests

Example: java -jar dtu.jar -numreq1000

Options: -m[] DTU Module Id
Sets the module id for DTU

-dmr[] DMR Module Id
Sets the module id for the Diameter Module (DMR)

-dsth[] Dest Host
-dstr[] Dest Realm
Set the Destination Host and Realm addresses

-numreq[] Number of requests
Sets the total number of Update Location Requests to send

-maxin[] Max sessions in flight
Sets the maximum number of session active at once

-maxssn[] Max session
Sets the maximum number of session ids to use

-basesn[] Base session id
Sets the base session id to use (default 0)

-nc[] NetworkContext
Sets the Network Context value

-mode[] Mode
Sets the mode for the sessions to use:
Send ULR: Set mode to 0 (default)
Send CCR: Set mode to 1

-traceoff Turn Trace Off

Typical Output

DSI DTU Diameter Test Utility Release 1.01
Part of the Dialogic(R) DSI Development Package
Copyright (C) 2012 Dialogic Inc. All Rights Reserved.
6.1.2 Command line examples
If host based or realm based routing is used then the host and realm options should be selected as shown below.

```java
java -Djava.library.path=/opt/DSI/32 -jar /opt/DSI/JAVA/dtu.jar -dsthdmr02.lab.dialogic.com -dstrdialogic.com -numreq10
```

An example set of command line options for Credit Control session handling (Ro) is as follows.

```java
java -Djava.library.path=/opt/DSI/32 -jar /opt/DSI/JAVA/dtu.jar -dsthdmr02.lab.dialogic.com -dstrdialogic.com -numreq10 -mode1
```

6.2 DTR Diameter Test Responder

**Description**
The DTR example application simulates the response to the sending of an Update Location Request Message and sends an Update Location Answer in reply.

**Syntax**

```java
java -jar dtr [-m<modid> -dmr<dmr modid> -traceoff]
```

**Parameters**

- `-m<modid>`
  Set to define the module id used by DTR. If not set, then DTR will default to 0x2d.

- `-dmr<dmr modid>`
  Set to define the module id used by the Diameter module (DMR). If not set, then DTU will default to 0x74.

- `-traceoff`
  Disable message tracing
6.2.1 Command line output

Help Screen

DSI DTR Diameter Test Responder Release 1.01
Part of the Dialogic(R) DSI Development Package
Copyright (C) 2012 Dialogic Inc. All Rights Reserved.

Syntax: dtr.jar [-m<> -dmr<> -lossrate<> -traceoff]
DTR Receives Diameter Session Requests and Answers them

Example: java -jar dtr.jar -m0x2d

Options:
- m[] DTR Module Id
  Sets the module id for DTR

- dmr[] DMR Module Id
  Sets the module id for the Diameter Module (DMR)

- delay[] Delay Rate
  If set, the module will insert a 10 msec delay
  every 1 in n ULR received.

- traceoff Turn Trace Off

Typical Output

DSI DTR Diameter Test Responder Release 1.01
Part of the Dialogic(R) DSI Development Package
Copyright (C) 2012 Dialogic Inc. All Rights Reserved.

DTR module id: 0x2d
DMR module id: 0x74

6.2.2 Command line examples

The example DTR system.txt file included within the DSI Development Package starts the dtr example automatically with default options. This is likely appropriate for most normal systems and situations. The command line used is as shown below. Unlike DTU, it is not necessary select a specific mode to handle different session message flows.

```bash
java -Djava.library.path=/opt/DSI/JAVA -jar /opt/DSI/JAVA/dtr.jar
```
7 Example C++ Applications

The DSI Development Package includes an example application for the C++ Functional API. This section provides further details and explanation of the example.

7.1 Installation

The development package should be installed in the normal manner following the instructions in the Dialogic® DSI Software Environment Programmer’s Manual.

The package installs the following directories and files:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Files</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>dmrapi.lib</td>
<td>The libraries contains the DMR Access API and the Diameter command specific functionality appropriate for the supported interfaces and commands. It also includes the encoding support used by other parts of the library.</td>
</tr>
<tr>
<td>INC/dmtrcmd</td>
<td>(Supplied include files)</td>
<td>Include files to support building against the Diameter Commands.</td>
</tr>
<tr>
<td>INC/dmtrcmd/rfc3588</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC/dmtrcmd/rfc4006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC/etc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPD/SRC/DMRAPI</td>
<td>dmr_api_tests.cpp</td>
<td>Sample code example to demonstrate use of the API</td>
</tr>
<tr>
<td>UPD/RUN/DMRAPI</td>
<td>system.txt</td>
<td>Simple system configuration file for use with the source code example</td>
</tr>
</tbody>
</table>

7.2 Building and running the example code

The sample API code in the file dmr_api_tests.cpp provides a simple application that sends a single Diameter command request to the s7_log utility and then simulates a reply by sending a Diameter command answer to itself and displaying it.

The example test application can be built together with the other parts of the User Development Package by calling the makeall.sh script in the UPD/SRC sub-directory.

```
cd /opt/DSI/UPD/SRC
./makeall.sh
```

To run the example application the user can make use of the supplied simple application loop-back system configuration in the UPD/RUN/DMRAPI directory. Change to the appropriate directory and start the message passing environment.
In another console window start the test application.

```
../../BIN/dmrapitest
DMR API Test Application
Result Code = 2000
Origin Host = origin.host.avp
Origin Realm = origin.realm.avp
Accuracy Fulfilment Indicator = 0
Vendor ID = 67
Feature List = 45
Feature List ID = 56
```

### 7.3 C++ Example Overview

The following sections give a more detailed overview of the C++ Functional API as used in the example code supplied.

#### 7.3.1 Generating a Diameter Session

**Step 1: Generate a diameter context**

The diameter context object is used to store and initialize factory and encoder objects required by the user application. This is demonstrated in the `DMR_API_TEST_build_dmr_context()` function within the supplied example.

**Step 2: Register a command with the context**

In order to define which commands are valid with the diameter context, the required requests and answer commands need to be registered into the context object. You can also set the module ids for the application and Diameter module here too. The function `DMR_API_TEST_build_dmr_context()` mentioned in step 1 also shows the commands being registered.

**Step 3: Create and populate the session/request object**

This step builds up a structured form of the request object which is to be encoded. This uses methods from the Diameter Command Specific API.

The creation of a request object is shown in the `DMR_API_TEST_build_dmr_session_open_req()` function. An example of this then being used to build up a Provide Subscriber Location request command is covered in `DMR_API_TEST_build_provide_location_request()`. 

```bash
cd /opt/DSI/UPD/RUN/DMRAPI
/opt/DSI/gctload
```
Step 4: Allocate a GCT message
From the context object, a method can be used to calculate the encoded length for the message, allowing the GCT message structure to be allocated using the getm() function from the normal C based message passing API (GCTLIB shared object libgct). This allocation is shown in DMR_API_TEST_transmit_dmr_session_req().

Step 5: Encode the command request into a GCT message
The context object also contains the encoder method to format the DmrSessionReq into a GCT message. The function referenced in Step 4 also shows and example of encoding the request.

Step 6: Send the GCT Message
Takes the GCT message and sends it to the DSI Diameter Module using the GCT_send() function from the normal C based message passing API (GCTLIB shared object libgct). The function referenced in Step 4 also shows the GCT message being sent.

7.3.2 Handling a received Diameter Request

Step 1: Generate a diameter context
The diameter context object is used to store and initialize factory and encoder/decoder objects required by the user application. This step is common with the code required for the Request generation.

Step 2: Register a command with the context
In order to define which commands are valid with the diameter context the required requests and answer commands need to be registered into the Context Object. This step is common with the code required for the Request generation.

Step 3: Receive the GCT message
This uses the GCT_receive() call which will block until a message for the requested module ids is returned. This function is from the normal C based message passing API (GCTLIB shared object libgct).

The function DMR_API_TEST_receive_dmr_session_ind() in the example code shows call to receive the next incoming messages using the same mechanism as other modules in the stack.

Step 4: Decode the Session Indication
As with encoding functionality, the diameter context object provides methods which will decode the GCT message into a structured object representing the indication. This is also shown in the DMR_API_TEST_receive_dmr_session_ind() function.
Step 5: Release the GCT message
Once handling is complete a call to relm() can be used to free the message structure. As with the GCT_receive() this function is from the normal C based message passing API (GCTLIB shared object libgct).