Java Encode/Decode API for 3GPP NAS Layer

Add-on and Standalone Kits for ASN1C

Table of Contents

Introduction

The NAS Encode/Decode API supports encoding and decoding 3GPP Layer 3 messages. These messages are described in the following 3GPP technical specifications:

- TS 24.007 - Mobile radio interface signaling layer 3; General Aspects
- TS 24.008 - Mobile radio interface Layer 3 specification; Core network protocols; Stage 3
- TS 24.301 - Universal Mobile Telecommunications System (UMTS); LTE; 5G; Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3
- TS 24.501 - 5G; Non-Access Stratum (NAS) protocpol for 5G System (5GS); Stage 3

This document explains how to use this API using the add-on package available for the ASN1C Compiler, or the standalone packages.

Methodology

This API has been developed in the Java programming language, using Objective Systems' ASN1C compiler to generate the structures and encode/decode functions. In order to be able to generate code for 3GPP specifications not using ASN.1, we added to ASN1C the ability to parse CSN.1, along with the ability to use configuration directives and custom code to refine the encoding/decoding of messages and information elements that we approximated using ASN.1 notation. The configuration directives are made effective by using the '-3gl3' command-line option with ASN1C. Our white paper, "Using ASN.1 to Describe 3GPP Messages" [http://www.obj-sys.com/docs/UsingASNtoDescribe3GPPMessages.pdf], describes how messages were approximated using ASN.1. (Note that this paper does not reflect our more recently capability to directly compile CSN.1 notation.)

The end result is an API that consists of Java types similar to what a user would get by compiling a standard ASN.1 specification. The other benefit of this approach is that in addition to encode/decode functions, supporting functions, such as print, can be generated from the definitions.

Note that one or more ASN.1 modules are associated with each of the above 3GPP specifications. In some of these modules, PDU ("Protocol Definition Unit") types are defined. These PDU types are used to represent an entire group of (or possibly all) messages from that module. Common fields are also factored out into the PDU type.

For each module, a separate Java package under com.objsys.nas contains the classes for that module. The Java packages are given in the following table:

Table 1. Module Packages

com.objsys.nas.TS24008IES	Information elements defined in 24.008
com.objsys.nas.TS24301IES	Information elements defined in 24.301.
com.objsys.nas.TS24301Msgs	Messages and information elements defined in 24.301. Type PDU is the toplevel type for these messages.
com.objsys.nas.TS24501IES	Information elements defined in 24.501.
com.objsys.nas.TS24501Msgs	Messages and information elements defined in 24.501. Type PDU is the toplevel type for these messages.

Contents of the Add-on Package

The following diagram shows the directory tree structure that comprises the add-on:

```
nasapi
|
+- build.xml
+- acconfig.xml
+- custsrc
+- doc
|   +- html
+- README.txt
+- specs
+- test_ts24301_msgs
+- test_ts24501_msgs
```

The package should be installed so that the 'nasapi' directory is contained in the 'java' subdirectory of the ASN1C installation.

- build.xml, acconfig.xml: Ant build and config file for generating the code and building the objsys_nas JAR file.
- custsrc: Contains custom code that is incorporated with the generated code.
- doc: Contains documentation. The PDF file is the user guide.

  The html subfolder contains the user guide in HTML format.

  After generating code, Javadoc documentation can be generated here by running `ant genjavadoc`. The JavaDoc is probably not very useful since the code (and its comments) are generated, but it might help with browsing the classes.
- specs: Contains the specification files from which code was created. As noted, these specification are often approximations of messages that 3GPP has not defined using ASN.1. You may find the specifications helpful for correlating the Java classes to the message and IE structures defined in the 3GPP specifications.
- test_ts24301_msgs: Contains tests for the 24.301 messages.
- test_ts24501_msgs: Contains tests for the 24.501 messages.

Contents of the Standalone Package

The following diagram shows the directory tree structure that comprises the standalone:

```
nasapi
|
+- build.xml
+- custsrc
+- doc
|   +- html
+- objsys_nas.jar
+- README.txt
+- specs
+- src
+- test_ts24301_msgs
+- test_ts24501_msgs
xmlpull
```

The package can be installed in any directory on the target system.

- build.xml: Ant build file for re-building the objsys_nas JAR file. (Unlimited source kits only)
- custsrc: Contains custom code that is incorporated with the generated code. (Unlimited source kits only)
- doc: Contains documentation. The PDF file is the user guide.

  The html subfolder contains the user guide in HTML format.
- objsys_nas.jar: JAR file containing NAS and necessary run-time class files for using the standalone kit.
- specs: Contains the specification files from which code was created. As noted, these specification are often approximations of messages that 3GPP has not defined using ASN.1. You may find the specifications helpful for correlating the Java classes to the message and IE structures defined in the 3GPP specifications.
- src: Contains Java NAS source files as well as run-time class files necessary for re-building the objsys_nas.jar file. (Unlimited source kits only)
- test_ts24301_msgs: Contains tests for the 24.301 messages.
- test_ts24501_msgs: Contains tests for the 24.501 messages.
- xmlpull: Directory containing helper JAR for decoding XML and XER messages.

Getting Started

The first thing you will need to do for the Add-on kit is build the JAR for it. This involves generating code, so you must have an ASN1C SDK license in place. To run the build, use the provided Ant build script. Run `ant build`. The build will generate code using asn1c, compile the code, and produce the objsys_nas.jar JAR file.

For the standalone kits, the objsys_nas.jar JAR file is already provided. It contains the class files for the generated NAS code, as well as those run-time class files that the NAS code depends on.

For the standalone unlimited source kit, the objsys_nas.jar JAR file can be re-built if necessary. To run the build, use the provided Ant build script as described above for add-on kits. The build will compile code and reproduce the objsys_nas.jar JAR file.

The easiest way to get started using the add-on is to examine the test_* subdirectories within the package. These contain test programs for encoding and decoding all of the different message types defined in the standards. Typically, code from within these samples can be used to form larger programs that can encode or decode larger message sets.

Licensing

The generated code will include embedded license information, based on your ASN1C SDK license. The purpose of this is to allow the code to run during development. For production usage, you will need to obtain an ASN1C Java runtime license. If you purchase a license for the unlimited ASN1C Java runtime, a license key/file won't be required. Otherwise, a runtime license file will be required and you will need to follow the normal procedure for an ASN1C runtime license (e.g. set ACLICFILE to point to your runtime license file).

Note that you can edit the Ant build script to use the `-nortkey` ASN1C command line option to prevent embedding license information in the code. This is useful for eliminating the possibility that you are relying on a temporary, embedded license key.

Using JSON (JER) and XML (XER) Encoding

The generated code includes encode and decode functions for JSON (ITU-T X.697 JER) and XML (ITU-T X.693 XER). For guidance on using these functions, refer to the ASN1C SDK Java User Guide.

The sample reader programs (see the next section), as test programs, also include code that roundtrips NAS data through JER and XER encodings. These programs can also serve as examples of doing conversions to/from these other encodings.

Sample Programs

Numerous sample programs are included in this package.

Sample programs can be built by using Apache Ant.

To build: `ant build`

To run the writer program: `ant writer`

To run the reader program: `ant reader`

The writer program encodes to message.dat and creates writer.log. The reader program decodes from message.dat and creates reader.log.

With respect to the Add-on kit, the ASN1C SDK includes a perl script that can be used to run all of the sample programs. The sample.pl script is located in the SDK installation folder. Commmands to use include: `perl sample.pl java nasapi/test_ts24301_msgs [clean|test]`. `perl sample.pl java nasapi/test_ts24501_msgs [clean|test]`.

Encoding Messages

Encoding 24.301 Messages

Encoding begins with a PDU ("Protocol Definition Unit") type which encompasses the messages for the corresponding specification.

Class PDU (com.objsys.nas.TS24301Msgs.PDU) has the following fields:

```
public SecProtMsgHeader secHdr;  // optional
public L3HdrOptions l3HdrOpts;
public ProtoDiscr protoDiscr = null;
public PDU_pti pti;
public _NAS_PROTOCOL_CLASS_msgType msgType;
public Asn1Type data;
public Asn1Null eom;
```

The `secHdr` field is for use with security-protected messages, which are not currently supported for Java. This field will thus not be used.

`l3HdrOpts` holds the EPS Bearer identity (for Session Management messages) or the Security header type (for Mobility Management messages) `protoDiscr` identifies whether the message is a Mobility Management or Session Management message. It will be either `ProtoDiscr.epsSessMgmt()` or `ProtoDiscr.epsMobMgmt()` (these are static enum values).

`pti` is the procedure transaction identity. It is only used for Session Management messages, but for Mobility Management messages it must be set to "no value".

`msgType` identifies the message type. This, together with `protoDiscr`, determines the actual type of the object in the `data` field. Class `_TS24301MsgsValues`defines constants for the various messages (see code example below).

`data` holds the message payload. For example, for an Authentication Request message, it is an `AuthRequest` object. A lengthy comment in PDU.java identifies the Java type that should be in this field for each {protocol discriminator, message type} pair.

`eom` is not used. It is there as a result of the techniques used to model these non-ASN.1 messages in a way ASN1C can work with.

The general procedure to encode a message is as follows:

1. Create an instance of the generated PDU type (e.g. `com.objsys.nas.TS24301Msgs.PDU`) and the specific message type to be sent (e.g. `com.objsys.nas.TS24301Msgs.AuthRequest`).
2. Populate the types. For the PDU, set the header fields described above. Set the message object into the PDU's data field.
3. Initialize the encode buffer.
4. Call the PDU encode function
5. Get the message from the buffer to work with the binary message.

An example with some comments follows. This example is based on the AuthRequest sample.

```
/* Create the PDU and Message objects */
com.objsys.nas.TS24301Msgs.PDU pdu = new PDU();
com.objsys.nas.TS24301Msgs.AuthRequest data = new AuthRequest();

/* Populate data structure */
pdu.protoDiscr = ProtoDiscr.epsMobMgmt();
pdu.pti = new PDU_pti(PDU_pti._NOVALUE, null);
pdu.l3HdrOpts = new L3HdrOptions();
pdu.l3HdrOpts.set_secHdr(new Asn1Integer(0));
pdu.msgType = new _NAS_PROTOCOL_CLASS_msgType(_TS24301MsgsValues.mt_AuthRequest);
pdu.data = data;

...

/* Create a message buffer object */

Asn1NasEncodeBuffer encodeBuffer = new Asn1NasEncodeBuffer();

/* Encode the PDU into the buffer */
pdu.encode (encodeBuffer);

/* Write message to a file.  You could also use encodeBuffer.getMsgCopy()
   to get the message into a byte array.
*/
```

```
   encodeBuffer.write (new FileOutputStream (filename));
```

Some general tips for encoding:

- Fields that are not optional must be assigned or a NullPointerException will occur. In particular, integer and boolean types are represented using objects (Asn1Integer and Asn1Boolean), so even fields having 0 or false as their value need to be assigned explicitly.
- Length fields will generally be automatically computed and encoded, but this is not true in all cases. Fields marked with `--<is3GLength/>` in the ASN.1 specification files will be automatically computed. If in doubt, you can look at the generated encode function or experiment to determine whether a given field is automatically computed or not.

Encoding 24.501 Messages

Encoding begins with a PDU ("Protocol Definition Unit") type which encompasses the messages for the corresponding specification.

Class PDU (com.objsys.nas.TS24501Msgs.PDU) has the following fields:

```
            public NAS5GSecProtMsgHdr secHdr;  // optional
            public NAS5GProtoDiscr protoDiscr = null;
            public PDU_hdrData hdrData;
            public _NAS5G_PROTOCOL_CLASS_msgType msgType;
            public Asn1Type data;
            public Asn1Null eom;
```

The `secHdr` field is for use with security-protected messages, which are not currently supported for Java. This field will thus not be used.

`protoDiscr` identifies whether the message is a Mobility Management or Session Management message. It will be either `NAS5GProtoDiscr.sessMgmt5G()` or `NAS5GProtoDiscr.mobMgmt5G()` (these are static enum values).

`hdrData` will contain either a `PDU_hdrData_mm` or `PDU_hdrData_sm`, depending on the protocol.

`msgType` identifies the message type (not surprisingly). This, together with `protoDiscr`, determines the actual type of the object in the `data` field. Class `_TS24501MsgsValues`defines constants for the various messages (see code example below).

`data` holds the message payload. For example, for an Authentication Request message, it is an `AuthRequest` object. A lengthy comment in PDU.java identifies the Java type that should be in this field for each {protocol discriminator, message type} pair.

`eom` is not used. It is there as a result of the techniques used to model these non-ASN.1 messages in a way ASN1C can work with.

`PDU_hdrData_mm` also has a `secHdrType` field. The only supported value is `NAS5GSecHdrType.noSec()`.

The general procedure to encode a message is as follows:

1. Create an instance of the generated PDU type (e.g. `com.objsys.nas.TS24501Msgs.PDU`) and the specific message type to be sent (e.g. `com.objsys.nas.TS24501Msgs.AuthRequest`).
2. Populate the types. For the PDU, set the header fields described above. Set the message object into the PDU's data field.

3. Initialize the encode buffer.
4. Call the PDU encode function
5. Get the message from the buffer to work with the binary message.

An example with some comments follows. This example is based on the AuthRequest sample.

```
/* Create the PDU and Message objects */
com.objsys.nas.TS24501Msgs.PDU pdu = new PDU();
com.objsys.nas.TS24501Msgs.AuthRequest data = new AuthRequest();

/* Populate data structure */
pdu.protoDiscr = NAS5GProtoDiscr.mobMgmt5G();
pdu.hdrData = new PDU_hdrData();
PDU_hdrData_mm mmHeader = new PDU_hdrData_mm();
pdu.hdrData.set_mm(mmHeader);

mmHeader.secHdrType = NAS5GSecHdrType.noSec();
mmHeader.spare1 = new Asn1Integer(0);

pdu.msgType = new _NAS5G_PROTOCOL_CLASS_msgType(_TS24501MsgsValues.mt_AuthRequ
pdu.data = data;

...

/* Create a message buffer object */

Asn1NasEncodeBuffer encodeBuffer = new Asn1NasEncodeBuffer();

/* Encode the PDU into the buffer */
pdu.encode (encodeBuffer);

/* Write message to a file.  You could also use encodeBuffer.getMsgCopy()
to get the message into a byte array.
*/
encodeBuffer.write (new FileOutputStream (filename));
```

Some general tips for encoding:

- Fields that are not optional must be assigned or a NullPointerException will occur. In particular, integer and boolean types are represented using objects (Asn1Integer and Asn1Boolean), so even fields having 0 or false as their value need to be assigned explicitly.
- Length fields will generally be automatically computed and encoded, but this is not true in all cases. Fields marked with `--<is3GLength/>` in the ASN.1 specification files will be automatically computed. If in doubt, you can look at the generated encode function or experiment to determine whether a given field is automatically computed or not.

Decoding Messages

Decoding begins with a PDU ("Protocol Definition Unit") type which encompasses the messages for the corresponding standard.

The following are the basic steps to decode with a PDU decode function:

1. Prepare a buffer for decoding
2. Create the instance of the PDU to receive the decoded data
3. Call the desired PDU decode function to decode the message

An example with some comments follows. This example is based on the AuthRequest sample.

```
/* Create a decode buffer on a file stream.
   The decode buffer could also be created on any InputStream or on a byte
   array.
*/
FileInputStream in = new FileInputStream (filename);

Asn1NasDecodeBuffer decodeBuffer = new Asn1NasDecodeBuffer (in);

/* Create PDU object to decode into.
For 24.301:
com.objsys.nas.TS24301Msgs.PDU pdu = new com.objsys.nas.TS24301Msgs.PDU();
*/
com.objsys.nas.TS24501Msgs.PDU pdu = new com.objsys.nas.TS24501Msgs.PDU();


/* Decode the data */
pdu.decode (decodeBuffer);

/* Do whatever you need to with pdu and the message it contains.
   Use pdu.msgType to determine the message type.
*/

/* For example, print the content */
pdu.print (System.out, "pdu", 0);

/* Or, cast to the correct message type */
AuthRequest msg = (AuthRequest)pdu.data;
```