CSTA 2 Java BER

Encode/Decode API

User's Guide

Objective Systems, Inc. August 2009

Introduction

The *Objective Systems' CSTA Java BER Encode/Decode API for Windows* is a Single Threaded jar library for encoding and decoding messages from the CSTA Phase 2 ASN.1 specification using the Basic Encoding Rules (BER) as defined in ITU standard X.690.

This API has been developed in the Java programming language. The Objective Systems ASN1C compiler was used to generate the structures and encode/decode functions. These were compiled with Java 1.5 compiler and packaged into a JAR file.

Contents of the Package

The following diagram shows the directory tree structure that comprises the Java CSTA BER encode/decode package:

```
csta2fw
 |
 +- build
 |
 +- doc
 |
 +- lib
 |
 +- src
 |
 +- sample
 |
 +- specs
```

The contents of the package will differ if the evaluation version is installed rather than the stand-alone package. The evaluation version should be installed on top of a current ASN1C evaluation package. The sources and JAR file may be generated by executing make or nmake from the build directory.

The purpose and contents of the various subdirectories are as follows:

- build – Contains a makefile for building the sources and JAR file.
- doc – Contains this document.
- lib – Contains the JAR file once it has been built. The JAR file is called csta2fw.jar and contains the CSTA phase two (ECMA-218), ROSE (ITU-T X.880 / ISO 13712-1), and ACSE (X.227 / ISO 8650) implementations.
- src – Contains the sources generated by ASN1C.
- sample – Contains a sample program to demonstrate how to use the enclosed package.
- specs – Contains the CSTA(ECMA-218), ROSE(X.880), ACSE(X.227), Information Framework(X.501), and UsefulDefinition (X.501) ASN.1 specifications that were used in the compilation.

Getting Started

The package is delivered as a zip file that can be installed in any directory on the development system.[1] All makefiles and internal sample programs use relative directory paths, so it is not necessary to create any type of top-level environment variables.

All of the necessary class files have been built using Java JDK 1.5. The code can be tested by executing the sample programs in the sample subdirectory. These sample programs consist of a reader and writer program. The writer program populates a data variable with some data, calls an encode function, and then writes the encoded byte stream to a file. The reader program reads this file, decodes the data into a Java structure, and then prints the decoded results.

CSTA explicit Association

CSTA protocol operates within an application association (otherwise known as a CSTA association or association) as provided by ISO 8649 (ACSE). This association can be either:

- an implicit association achieved via off-line agreement or
- an explicit association realized through the use of ACSE.

The initialization sequence of CSTA messages for the implicit and explicit associations is described in the following sections.

Explicit/dynamic association can be realized by using acse.jar (ACSE ASN.1 implementation) API. An application context is established as follow:

1) The system generates AARQ-apdu & waits for AARE-apdu from receiving System.
2) Receiving system receives the AARQ-apdu, selects the protocol version to be used by identifying highest csta version that is common to both system. Receiving System generates AARE-apdu using selected csta protocol version.

It is necessary for the requesting and responding systems to specify the CSTA services that they support. As with the protocol version information, this is also achieved by carrying additional information in the User Information field of the A-ASSOCIATE request and response PDUs. The application association requestor shall:

- list the services required from the serving application;
- list the services it can supply.

The responder shall include similar information for the responding application. At this point the association requestor will either accept or reject the association.

Example of ACSE connection request is available in sample/acseConnection directory.

1) application context Name - CSTA object identifier.

2) CSTA protocol version information, which is carried within the "User Information" field of the ACSE request and response PDUs.

---

1 This is only true of the stand-alone version; evaluation versions should be installed in ASN1C_INSTALL_DIR\java.

Encoding CSTA Messages with ROSE Header

The CSTA specification specifies a two-phase protocol using ROSE for the common headers. In order to encode a message of this type, the following steps must be performed:

1. A CSTA base message type must be populated, and

2. The results must be plugged into a ROSE message structure and then this is encoded to produce the finished message.

The user should use the writer program (*writer.java*) in the one of the sample directories as a guide when reading the rest of the procedure.

1 Encoding a CSTA message

To populate a CSTA message component, a variable of one of the various CSTA class structures must first be populated with data. These structures normally correspond to the ARGUMENT or RESULT types specified in a CSTA2 Information Object for OPERATION class. For example, the following information object specifies the messages that are exchanged for the ***makeCall*** operation:

```
makeCall OPERATION ::= {
      ARGUMENT    MakeCallArgument
      RESULT      MakeCallResult
      ERRORS      {universalFailure}
      CODE local : 10
}
```

In this information object, CODE field defines the value "local:10" to identify for ***makeCall*** operation. ARGUMENT field defines the `MakeCallArgument` type, which can be used as invoke the ***`makeCall`*** operation. RESULT field defines the MakeCallResult type, which is the used to return the result of the ***`makeCall`*** operation. ERRORS field defines the another information object, which contains the error type can be produced in ***`makeCall`*** operation.

Table 1.1 is developed from the csta phase 2 information object definitions. This table contains operation name, operation code, Argument type & Result type for this operation. To encode/invoke the operation user will require to set the operation code value & encode the Argument type defined in this table. And to decode the operation user will require to check the operation code value & decode the corresponding result type or argument type. e.g For invoke the ***makeCall*** operation, user will need to set the CSTA_ROSE_PDU_invoke.operationCode value to local:10 & encode the `MakeCallArgument type in` CSTA_ROSE_PDU_invoke. Argument open type.

In this case, MakeCallArgument is encoded and sent as a request message (or invoke as it is known in ROSE). The entity receiving this message is then required to respond with the result message of MakeCallResult type or one of the defined errors in `universalFailure information object`.

The sample program shows how to encode a MakeCallArgument:

```
MakeCallArgument ::=
            SEQUENCE
            {callingDevice       DeviceID,
             calledDirectoryNumber  CalledDeviceID,
             extensions    CSTACommonArguments  OPTIONAL}
```

2 Encoding a ROSE Header

Once the argument is populated and encoded, the ROSE header must be added. This is a common header that is added to all messages that support the ROSE protocol. In the case of an ROSE OPERATION, a ROSE Invoke message must be sent to the other entity.

The ROSE header required to send an invoke message consists of 4 fields:

1. Invoke ID: this is an arbitrary identifier that acts as a "handle" for matching responses to requests when messages are exchanged. Any result or error received in response to this invoke request will contain this identifier value.

2. Linked ID: this is another Invoke ID that is used when a sub-operation within the existing operation is initiated. The Linked ID is the Invoke ID of the parent (i.e. the encapsulating) operation.

3. Operation Code: this identifies the operation to the receiving entity. Table 1.1 can be used to find out the operation code value for particular operation. e.g. ***makeCall*** operation correspond to the "local : 10" value.

4. Message Data : this is a open type. CSTA populated message data is placed in this open type. Table 1.1 can be used to find out the Type for particular operation. E.g. ***makeCall*** operation correspond to the MakeCallArgument type.

The following is a snippet from the writer.cs sample program showing how the header is added:

```
/* Populate header structure */

invoke.invokeId = new InvokeId();

/* arbitrary number: should be unique */
invoke.invokeId.set_present(new Asn1Integer(1));


/* This is where we get the previously encoded message component */

/* operation code for "makeCall" operation from Table 1.1 */
invoke.opcode.set_local (new Asn1Integer(10));
invoke.argument = makeCallArgument;

pdu.set_invoke(invoke);
```

The header identifies the operation to be performed (opcode = 10 = makeCall) and assigned a unique invoke identifier. This invoke identifier serves as a session ID that can be used to match requests with respose if asynchronous communications are used. The last part of the populate logic gets the previously encoded message component from encoding the make call argument data. This is the open type onto which the ROSE header is pre-pended.

Decoding CSTA Messages

CSTA messages are decoded by reversing the procedure that was used to encode them. In other words, the following two distinct decode operations must be performed. The ROSE & CSTA message will be decoded CSTA_ROSE_PDU type decode call

This is the inverse of the encoding procedure presented earlier. The user should use the reader program (*reader.cs*) in the sample directory as a guide when reading the rest of the procedure.

The procedure to decode a complete CSTA message is as follows:

1. Read an encoded message from an input stream.

2. Create an *Asn1BerDecodeBuffer* object to wrap the message buffer that the message was read into.

3. Create a *CSTA_ROSE_PDU* object and use it in conjunction with the decode buffer object created above to decode the header.

4. The header fields can now be examined. An application will first check Invoke ID to find out the response for different session. For our example, value of the Invoke ID field should "1", which is random unique number we have set during encode procedure. Than to identify the operation, check the operation code value, which should be "local:10" for a result/error for our invoke request.

Table 1.1: Operation Table for CSTA II

Operation name	Operation Identifier	Operation Invoke type	Operation Result type
alternateCall	local: 1	AlternateCallArgument	AlternateCallResult
answerCall	local: 2	AnswerCallArgument	AnswerCallResult
callCompletion	local: 3	CallCompletionArgument	CallCompletionResult
clearCall	local: 4	ClearCallArgument	ClearCallResult
clearConnection	local: 5	ClearConnectionArgument	ClearConnectionResult
conferenceCall	local: 6	ConferenceCallArgument	ConferenceCallResult
consultationCall	local: 7	ConsultationCallArgument	ConsultationCallResult
divertCall	local: 8	DivertCallArgument	DivertCallResult
holdCall	local: 9	HoldCallArgument	HoldCallResult
makeCall	local: 10	MakeCallArgument	MakeCallResult
makePredictiveCall	local: 11	MakePredictiveCallArgument	MakePredictiveCallResult
queryDevice	local: 12	QueryDeviceArgument	QueryDeviceResult
reconnectCall	local: 13	ReconnectCallArgument	ReconnectCallResult
retrieveCall	local: 14	RetrieveCallArgument	RetrieveCallResult
setFeature	local: 15	SetFeatureArgument	SetFeatureResult
transferCall	local: 16	TransferCallArgument	TransferCallResult
associateData	local: 17	AssociateDataArgument	AssociateDataResult
parkCall	local: 18	ParkCallArgument	ParkCallResult
sendDTMFTones	local: 19	SendDTMFTonesArgument	SendDTMFTonesResult
singleStepConf	local: 20	SingleStepConfArgument	SingleStepConfResult
cSTAEventReport	local: 21	CSTAEventReportArgument	
routeRequest	local: 31	RouteRequestArgument	
reRouteRequest	local: 32	ReRouteRequestArgument	
routeSelectRequest	local: 33	RouteSelectRequestArgument	

routeUsedRequest	local: 34	RouteUsedRequestArgument	
routeEndRequest	local: 35	RouteEndRequestArgument	
singleStepTrans	local: 50	SingleStepTransArgument	SingleStepTransResult
escapeService	local: 51	EscapeServiceArgument	EscapeServiceResult
systemStatus	local: 52	SystemStatusArgument	SystemStatusResult
monitorStart	local: 71	MonitorStartArgument	MonitorStartResult
changeMonitorFilter	local: 72	ChangeMonitorFilterArgument	ChangeMonitorFilterResult
monitorStop	local: 73	MonitorStopArgument	MonitorStopResult
snapshotDevice	local: 74	SnapshotDeviceArgument	SnapshotDeviceResult
snapshotCall	local: 75	SnapshotCallArgument	SnapshotCallResult
startDataPath	local: 110	StartDataPathArgument	StartDataPathResult
stopDataPath	local: 111	StopDataPathArgument	StopDataPathResult
sendData	local: 112	SendDataArgument	SendDataResult
sendMulticastData	local: 113	SendMulticastDataArgument	SendMulticastDataResult
sendBroadcastData	local: 114	SendBroadcastDataArgument	SendBroadcastDataResult
suspendDataPath	local: 115	SuspendDataPathArgument	SuspendDataPathResult
dataPathSuspended	local: 116	DataPathSuspendedArgument	DataPathSuspendedResult
resumeDataPath	local: 117	ResumeDataPathArgument	ResumeDataPathResult
dataPathResumed	local: 118	DataPathResumedArgument	DataPathResumedResult
fastData	local: 119	FastDataArgument	FastDataResult
concatenateMessage	local: 500	ConcatenateMessageArgument	ConcatenateMessageResult
deleteMessage	local: 501	DeleteMessageArgument	DeleteMessageResult
playMessage	local: 502	PlayMessageArgument	PlayMessageResult
queryVoiceAttribute	local: 503	QueryVoiceAttributeArgument	QueryVoiceAttributeResult
reposition	local: 504	RepositionArgument	RepositionResult
resume	local: 505	ResumeArgument	ResumeResult
review	local: 506	ReviewArgument	ReviewResult
setVoiceAttribute	local: 507	SetVoiceAttributeArgument	SetVoiceAttributeResult
stop	local: 508	StopArgument	StopResult
suspend	local: 509	SuspendArgument	SuspendResult
synthesizeMessage	local: 510	SynthesizeMessageArgument	SynthesizeMessageResult
recordMessage	local: 511	RecordMessageArgument	RecordMessageResult

NOTE: For all the above Operations or Information Objects, return error type is “UniversalFailure”.