BindRequest

BindRequest Message

This is the very first message sent by a client to a Ldap Server. It contains the identification of the user and itss credentials, which may be either **simple** or **sasl**.

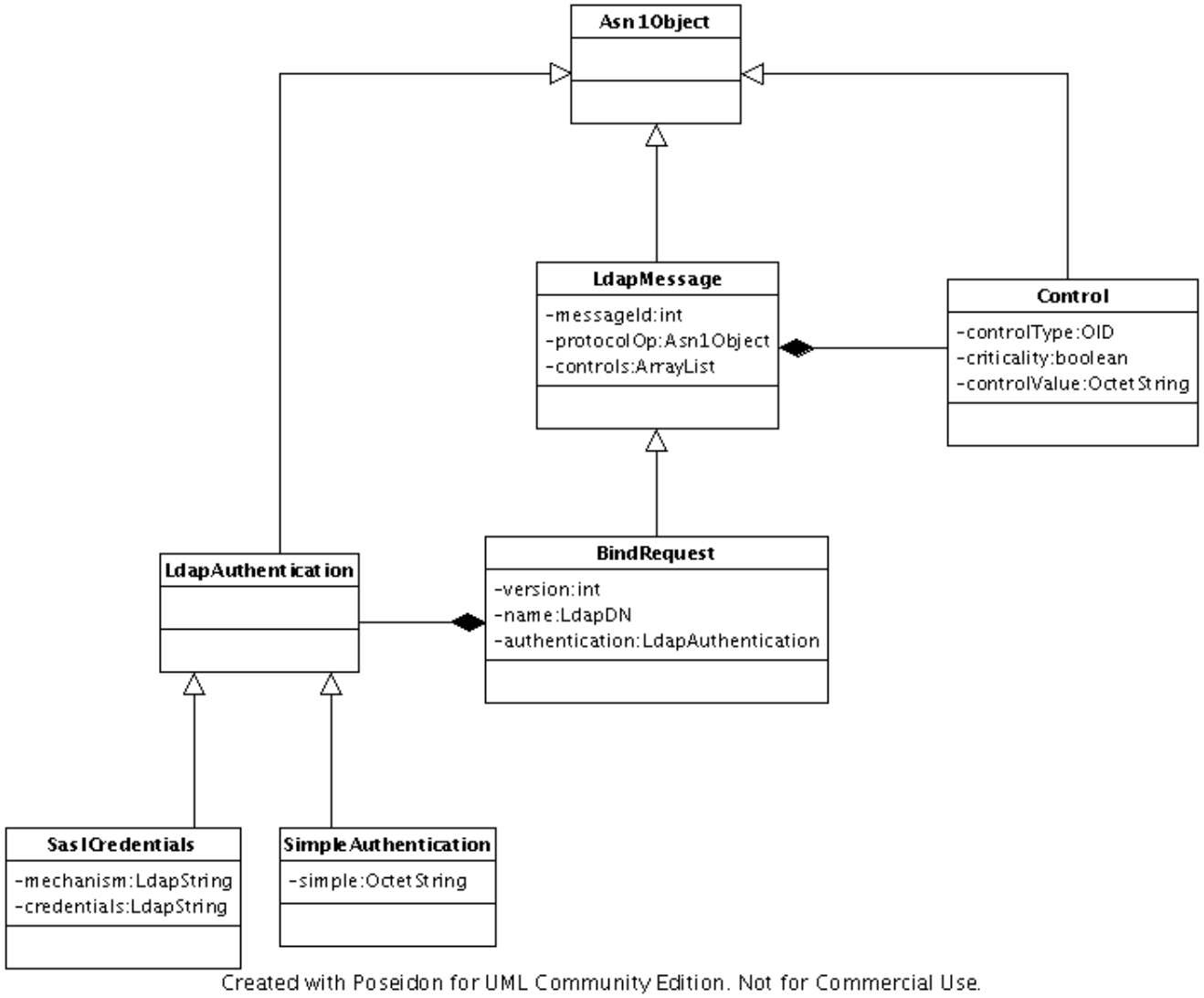
Message structure

Here is the ASN.1 grammar for a BindRequest (you can find it in RFC 2251)

```
LDAPMessage ::= SEQUENCE {
    messageID  MessageID,
    protocolOp CHOICE {
        bindRequest  BindRequest,
        ... },
    controls   [0] Controls OPTIONAL }

MessageID ::= INTEGER (0 .. maxInt)

maxInt INTEGER ::= 2147483647 -- (2^^31 - 1) --

BindRequest ::= [DIRxSBOX:APPLICATION 0] SEQUENCE {
    version         INTEGER (1 .. 127),
    name            LDAPDN,
    authentication  AuthenticationChoice }

AuthenticationChoice ::= CHOICE {
    simple  [0] OCTET STRING,
            -- 1 and 2 reserved
    sasl    [3] SaslCredentials }

SaslCredentials ::= SEQUENCE {
    mechanism    LDAPString,
    credentials  OCTET STRING OPTIONAL }

LDAPDN ::= LDAPString

LDAPString ::= OCTET STRING
```

Java implementation

We have a Java Bean which contains all the necessary informations. Here is the Class diagram for this message:



We have a kind of complicated structure to deal with ! In this class diagram, we have only the data that are parts of the Ldap Message:

Ldap Message structure:

- a MessageId: an integer between 0 and 2,147,483,647
- a protocolOp: it contains the body of the BindRequest
- a list of controls

BindRequest structure:

- a version: here it will always be the value **3**
- a name: it can be null, if the user performs its authentication with **SASL**
- an authentication: it contains the authentication used

Simple authentication structure:

- a simple: the password

SASL authentication structure:

- a mechanism: the name of the used mechanism (KERBEROS_V4, ...)
- a credentials: bytes used to store the user's credentials

Control structure (if any):

- a control type: the corresponding OID
- a criticality flag: used to tells the server how to handle the control
- a control value: the control's value

Each structure is implemented as a Java Class.

encoding the PDU

Now, we have to transform these Java classes to a PDU

First, here is the BindRequest Message we will use:

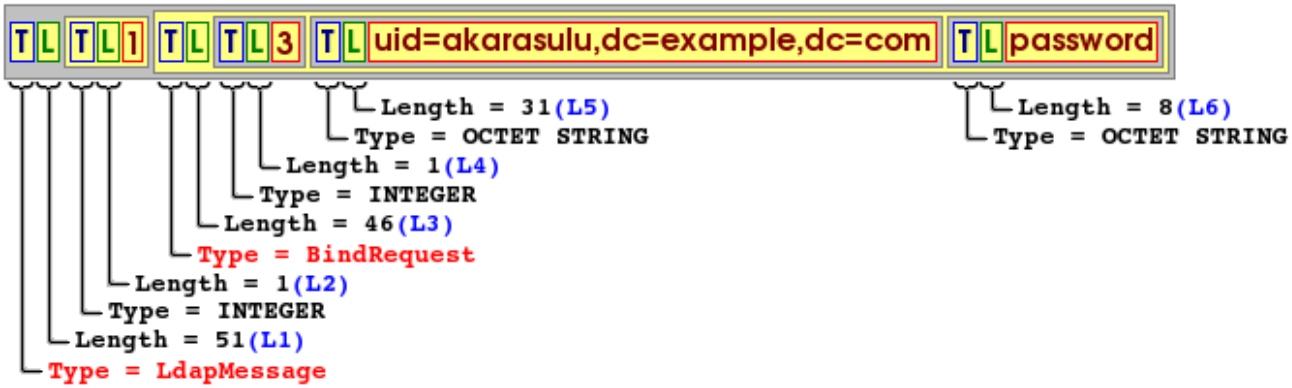
```
LdapMessage
    message Id : 1
    BindRequest
        Version : '3'
        Name : 'uid=akarasulu,dc=example,dc=com'
        Simple authentication : 'password'
```

and here is the resulting PDU:

```
0x30 0x33
  0x02 0x01 0x01
  0x60 0x2E
    0x02 0x01 0x03
    0x04 0x1F 0x75 0x69 0x64 0x3D 0x61 0x6B 0x61 0x72 0x61 0x73 0x75 0x6C 0x75 0x2C 0x64 0x63
              0x3D 0x65 0x78 0x61 0x6D 0x70 0x6C 0x65 0x2C 0x64 0x63 0x3D 0x63 0x6F 0x6D
    0x80 0x08 0x70 0x61 0x73 0x73 0x77 0x6F 0x72 0x64
```

In the resulting PDU, we have separated each primitive TLV, so every line is a single TLV.

We can represent the PDU structure with this schema:



Th PDU has three levels of encapsulation.

- The first level is a constructed TLV, which length is 51 bytes. That is the LdapMessage envelope
- the second level is composed of two types of elements: Primitive types like Integer (Version) and a Constructed Type: the BindRequest
- The third level is the BindRequest itself, composed of Primitive TLV only.

So we have only two constructed TLV in this sample: LdapMessage and BindRequest.

Computing the PDU length

We will have to store the encoded result in a ByteBuffer, so we must compute its length One could think that it's easy in this specific case, because **L1** is the total length . This is not so simple.

- first, L1 is the length of the Value, so we need to add the T and L lengths
- second, even if valid for LdapMessages, this is not a general case. A PDU may be composed of many successive PDUs, thus the first TLV length might not represents the total size.

The best solution is to walk the structure and to compute each length. We have the following relations :

```
PDU length = T1.length + L1.length + L1
L1 = T2.length + L2.length + L2 + T3.length + L3.length + L3
L2 = messageId.length
L3 = T4.length + L4.length + L4 + T5.length + L5.length + L5 + T6.length + L6.length + L6
L4 = version.length
L5 = name.length
L6 = authentication.simple.length
```

We can see that it's not possible to know L1 before having evaluated L2 and L3, and for that we also need to compute L4, L5 and L6.

We cannot either dump bytes as soon as we have read the data from the JavaBean, because we will have to update L1 and the end of the processing.

We have many possible strategies to solve this problem :

- use of a recursive approach
- use a standard approach, using transient values to store temporary computations.
- combine those two strategies

in conjonction with two possible ways to generate the bytes :

- construct the PDU using many ByteBuffers and updating the cumulative lengths on the fly
- construct the PDU by dumping computed lengths and values

The first solution leads to use a tree of temporary ByteBuffers which will be assembled before being sent, and is recursive.
The second solution need to first computes the length, and second to generate the full PDU

For many reasons, the second strategy has been adopted, as it's faster to implement, and faster to execute.

So we will first compute the lengths by reordering the way lengths are calculated:

```
L2 = messageId.length
L4 = version.length
L5 = name.length
L6 = authentication.simple.length

L3 = T4.length + L4.length + L4 + T5.length + L5.length + L5 + T6.length + L6.length + L6

L1 = T2.length + L2.length + L2 + T3.length + L3.length + L3

PDU length = T1.length + L1.length + L1
```

First, we get all the primitive lengths, and store them in temporary variables.
Then we compute the BindRequest length (L3), finishing with the LdapMessage length (L1), finishing with the PDU length.

We are now able to allocate a ByteBuffer which length will be equal to **Pdu Length**.

We will also store in each objects the computed values, to avoid a double evaluation. Here, we will store a *ldapMessageLength*, and a *bindRequestLength* (We will also store internal lengths for Controls, if need)

This is a little bit over-splimplistic, in reallity, we ask the BindRequest class to compute its internal length. This is the reason why we have described the adopted solution as a combined solution (recursive/linear)

Generating the PDU

As soon as we have computed the PDU's lengths, we can feed the ByteBuffer.

We just have to respect the Grammar order:

1. a constructed TLV for the LdapMessage envelope
2. the messageId
3. a constructed TLV for the BindRequest envelope
4. the version
5. the name
6. the simple authentication

Note : we will ask the BindRequest object to generate its bytes and to pour them into the allocated ByteBuffer.

That's it! We now have a ByteBuffer which is a PDU. MINA, it's up to you ...

Conclusion

The encoding process is simple. One who wants to implement an encoder has to be warned about some tricks :

- String cannot be encoded as is. In Java, a String is encoded in Unicode, so special characters cannot be stored in single bytes. To compute String's length, you **must** call the String.getBytes() function. Then, you have to store the bytes, and use the length of this byte array to know the real String's length, expressed in bytes.
- you will have to store lengths, too. but if the Value is longer than 127 bytes, the length part will not fit in a single byte. You will have to use a second byte, sometime a third or a fourth one. Consider using the Length.getNbBytes (length) and Length.getBytes (length) methods to do so (the first one gives back the number of bytes, the second one generates the bytes).
- if the Value part is null or empty, the length will be 0. It has to be encoded.