ASN.1

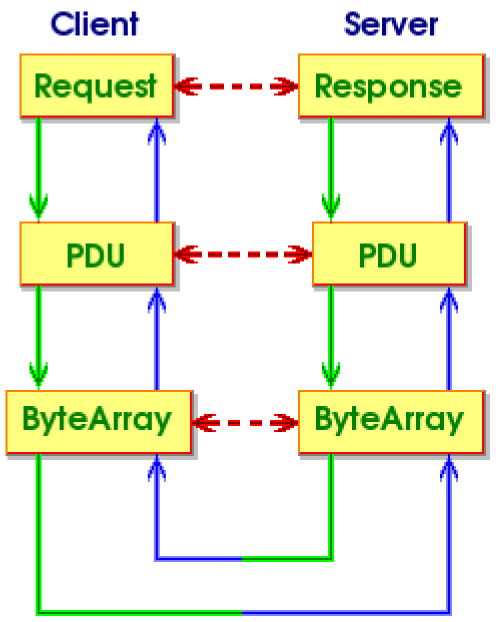
ASN.1 encoder/decoder

- ASN.1 encoder/decoder
  - Components
    - POJOs
    - PDUs/TLVs
    - ByteBuffer
  - Compiler
  - Processing
    - Encoder
    - Decoder
    - Performance
  - Implementations

Components

Communications between clients and server can be seen as a two ways / multi *layers* system. The client submits a request to the server, which replies.

Layers are just used to facilitate the implementation of this communication. From the developer point of view, working on a specific level, he just has to know the two layers above and under, but can be seen as a communication at she same level between the client and the server. Here is a view of these layers :



We have depicted three layers:

- Request/Response: This is the more abstract layer. Exchanged messages are 'human readable'. Each message is a Java Bean, containing all the information about a Request or a Response.
- PDU: As communication petween the Client and the Server is done through a network, we need to transform the beans to something more 'network friendly'. The data are stored in PDU, or **P**rotocol **D**ata **U**nit. Those PDU contain an encoded form of messages, specified in RFC 2251 and ASN.1
- ByteBuffers: To transfer PDU from/to Client to/from Server, we need to store bytes in a structure that will permit to deal with network latency. Thus we are using byte buffers, which allow us to send pieces of PDU until the whole PDU has been transmitted. (Note : ByteBuffer is also a Java NIO class, but can be seen just as a byte container. It could have been something totally different from the NIO class).

This layering allows many different implementations.

One can also imagine inter-layers used to trace debug informations.

Inter layer communication rely on a pipe-line: each layer push some piece of information to the next layer (up or down), and so on.

Each layer may implement its own strategy to fulfill the reception and transmission of the data it is responsible of :

- emission
  - asynchronous push
  - synchronous push

    - established and dedicated channel
    - multiplexed channel
- reception
    - listener
    - established and dedicated channel
    - multiplexed channel

POJOs

POJOs are Java classes that contain high level informations.

A client create an *instance* of a class to communicate with the server, which create an other one to reply. They implement a kind of *application layer* between clients and server.

Ideally, they are generated by an **ASN.1** compiler, but can be hand crafted.

PDUs/TLVs

PDU stands for **P**rotocol **D**ata **U**nit. An ASN.1 encoded element is stored in a PDU. This is what is transfered between a client and a server.

TLV stands for **Type/Length/Value**. A PDU is made of **TLV** s. Each **TLV** represent either a primitive element, and it has a **V**alue, or a constructed element, and the **V**alue is itself one ore more **TLV** (The **V** can contain more than one **TLV**). The **PDU** structure is like a tree, where the **PDU** is the whole tree, and where **TLV** are leaves (primitives) and branches (constructed)

Further information about **TLV**s can be found here :

- TLV Page Info: Informations about **Tlv**s

ByteBuffer

Buffering the incoming request or the ourgoing response is essential. As a request or a response can be huge (for example, if we want to store images), it is necessary to store bytes in buffers in order to be able to pipeline the processing. Flushing informations byte by byte is totally insane, from the network point of view.

We are using the **NIO** ByteBuffer structure to store chunks of information, before pushing them on the network, and reversly, store incoming bytes into buffers before processing the request.

Compiler

TO BE DONE ...

Processing

There are two kind of processing: **encoding** and **decoding**. Encoding is quite easy, decoding is much more complicated.

Important

Important : decoding an ASN.1 PDU is generally not possible if you have no knowledge of the grammar being decoded. To limit the size of PDUs, the encoding schemes used (PER, DER, BER, ...) permits the elimination of some TL if the constructed TLV that encapsulate the previous one is unambiguiously known. One who want to decode a PDU **MUST** know which grammar has been used.

Encoder

The encoding process is quite easy. As we know what has to be encoded, the structure of the PDU is somehow dependent on the structure of the POJO which contains the data. The only tricky things is the **Length** part, which has to be computed. As a **TLV** may have a **V** part which is itself one or more **TLV** s, its **L** part will be the sum of each included **TLV** s length. This is typically a recursive processing, but we can also process the POJO in two passes :

- the first pass compute each length
- the second pass generate the **PDU**

The Encoding Asn.1 page gives informations about the encoding process.

Decoder

The decoding process is a loop which reads PDUs and constructs objects on the fly. It can stop and restart without loosing information, as PDU may be very long (it also means that we must store a current state for each decoding).

The Decoding Asn.1 page gives informations about the encoding process.

Performance

TODO : performance against memory/scalability/failover
TODO : which kind of performance should we deliver? Maximum throughput = bandwith/average PDU size. For instance, with a 1Gb network connection, assuming that we have an average PDU size of 100 bytes, the system must deliver 1 M Pdu/s to saturate the network.

Actually, the new decoder eats 110 000 BindRequest PDU or 37 000 SearchResultEntry PDU per second on my 2.8Ghz computer, but we have to take into account the works that must be done aside.

Implementations

Three codecs are currently available :

- LdapCodec which encodes and decodes LDAP messages
- KerberosCodec which encodes an decodes KERBEROS messages
- SpnegoCodec which encodes an decodes SPNEGO messages