I need an example to understand Implicit Tagging in ASN.1

I have been going through the following tutorial

http://www.obj-sys.com/asn1tutorial/node12.html

Can you help me understand implicit tagging with an example?

tags asn.1

asked Jul 21 '10 at 6:26

Rohit Banga
4,618 11 50 116

2 Answers

I find this thread to be clear enough, it also contains (small) examples even tough they are quite 'extreme' examples at that. A more 'realistic' examples using IMPLICIT tags can be found in this page.

answered Feb 3 '11 at 16:46

lapo
1,752 13 22

for me as beginner examples aren't clear enough – Jaime Hablutzel May 29 '12 at 16:17

In ASN.1 tagging in fact serves two purposes: typing and naming. Typing means it tells an en-/decoder what kind of data type that is (is it a string, an integer, a boolean, a set, etc.), naming means that if there are multiple fields of the same type and some (or all of them) are optional, it tells the en-/decoder for which field that value is.

If you compare ASN.1 to, let's say, JSON, and you look at the following JSON data:

```
"Image": {
    "Width":  800,
    "Height": 600,
    "Title":  "View from 15th Floor"
}
```

You'll notice that in JSON every field is always explicitly named ("Image", "Width", "Height", "Title") and either explicitly or implicitly typed ("Title" is a string, because its value is surrounded by quotes, "Width" is an integer, because it has no quotes, only digits, it's not "null", "true" or "false", and it has no decimal period).

In ASN.1 this piece of data would be:

```
Image ::= SEQUENCE {
    Width  INTEGER,
    Height INTEGER,
```

```
    Title  UTF8String
 }
```

This will work without any special tagging, here only the universal tags are required. Universal tags don't name data, they just type data, so en-/decoder know that the first two values are integers and the last one is a string. That the first integer is Width and the second one is Height doesn't need to be encoded in the byte stream, it is defined by their order (sequences have a fixed order, sets don't. On the page you referred to sets are being used).

Now change the schema as follows:

```
Image ::= SEQUENCE {
    Width  INTEGER OPTIONAL,
    Height INTEGER OPTIONAL,
    Title  UTF8String
 }
```

Okay, now we have a problem. Assume that the following data is received:

```
INTEGER(750), UTF8String("A funny kitten")
```

What is 750? Width or Height? Could be Width and Height is missing or could be Height and Width is missing, both would look the same as binary stream. In JSON that would be clear as every piece of data is named, in ASN.1 it isn't. Now a type alone isn't enough, now we also need a name. That's where the non-universal tags enter the game. Change it to:

```
Image ::= SEQUENCE {
    Width  [0] INTEGER OPTIONAL,
    Height [1] INTEGER OPTIONAL,
    Title  UTF8String
 }
```

And if you receive the following data:

```
[1]INTEGER(750), UTF8String("A funny kitten")
```

You know that 750 is the Height and not the Width (there simply is no Width). Here you declare a new tag (in that case a context specific one) that serves two purposes: It tells the en-/decoder that this is an integer value (typing) and it tells it which integer value that is (naming).

But what is the difference between implicit and explicit tagging? The difference is that implicit tagging just names the data, the en-/decoder needs to **know the type implicitly for that name**, while explicit tagging names and **explicitly types the data**.

If tagging is explicit, the data will be sent as:

```
[1]INTEGER(xxx), UTF8String(yyy)
```

so even if a decoder has no idea that [1] means Height, it knows that the bytes "xxx" are to be parsed/interpreted as an integer value. Another important advantage of explicit tagging is that the type can be changed in the future without changing the tag. E.g.

```
Length ::= [0] INTEGER
```

can be changed to

```
Length ::= [0] CHOICE {
    integer INTEGER,
    real    REAL
}
```

Tag [0] still means length, but now length can either be an integer or a a floating point value. Since the type is encoded explicitly, decoders will always know how to correctly decode the value and this change is thus forward and backward compatible (at least at decoder level, not necessarily backward compatible at application level).

If tagging is implicit, the data will be sent as:

```
[1](xxx), UTF8String(yyy)
```

A decoder that doesn't know what [1] is, will not know the type of "xxx" and thus cannot parse/interpret that data correctly. Unlike JSON, values in ASN.1 are just bytes. So "xxx" may be one, two, three or maybe four bytes and how to decode those bytes depends on their data type, which is not provided in the data stream itself. Also changing the type of [1] will break existing encoders for sure.

Okay, but why would anyone use implicit tagging? Isn't it better to awalys use explicit tagging? With explicit tagging the type must also be encoded in the data stream and this will require two additional bytes per tag. For data transmissions containing several thousand (maybe even millions of) tags and where maybe every single byte counts (very slow connection, tiny packets, high packet lost, very weak processing devices) and where both sides know all custom tags anyway, why wasting bandwidth, memory, storage and/or processing time for encoding, transmitting and decoding unnecessary type information?

Keep in mind that ASN.1 is a rather old standard and it was intended to achieve a highly compact representation of data at a time where network bandwidth was very expensive and processors several hundred times slower than today. If you look at all the XML and JSON data transfers of today, is seems ridiculous to even think about saving two bytes per tag.

edited Jun 4 at 22:08

answered Jun 2 at 23:28

Mecki

44.2k 20 100 141