



Universidade do Minho  
Escola de Engenharia

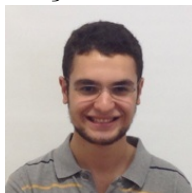
## **Codificação/Descodificação de PDUs SNMPv2c**

### **Gestão de Redes : Trabalho Prático 3**

Mestrado Integrado em Engenharia Informática  
Universidade do Minho

1º Semestre  
2017-2018

Gonçalo Pereira



a74413

José Silva



a75280

11 de fevereiro de 2018

## Contents

<b>1</b>	<b>Resolução do problema</b>	<b>3</b>
1.1	Encoder . . . . .	4
1.2	Decoder . . . . .	6
1.3	Execução do código . . . . .	7

# 1 Resolução do problema

Inicialmente, e de modo a que fosse possível a incorporação de uma API *open source* ASN1C de codificação/descodificação, procedemos à realização do tutorial fornecido pelos professores Fábio Gonçalves e Bruno Dias.

O ASN.1 é uma linguagem que permite descrever cada item declarado a sua constituição. Este tipo linguagem é usado na definição de objetos nas MIBs. Para tal este trabalho serve para observar os passos por que um PDU do Net-SNMP passa até chegar a um dado *host*, ou seja, como está declarado o PDU e como é codificado e descodificado.

Após a correta instalação do mesmo, remetemos para a geração de todos os ficheiros futuramente criados pela integração das definições ASN.1 para o SNMPv2c. Neste passo criamos um ficheiro em ASN1 com a declaração de um PDU SNMPv2 e de onde foram geradas todas as estruturas que serão usadas durante todo o trabalho, subdividindo-as em duas outras: uma para codificação (encoder) e outra para descodificação (decoder).
























 ANY	30/01/2018 17:06	Ficheiro C
 ANY	30/01/2018 17:06	C/C++ Header
 ApplicationSyntax	30/01/2018 17:06	Ficheiro C
 ApplicationSyntax	30/01/2018 17:06	C/C++ Header
 asn_application	30/01/2018 17:06	Ficheiro C
 asn_application	30/01/2018 17:06	C/C++ Header
 asn_bit_data	30/01/2018 17:06	Ficheiro C
 asn_bit_data	30/01/2018 17:06	C/C++ Header
 asn_codecs	30/01/2018 17:06	C/C++ Header
 asn_codecs_prim	30/01/2018 17:06	Ficheiro C
 asn_codecs_prim	30/01/2018 17:06	C/C++ Header
 asn_internal	30/01/2018 17:06	Ficheiro C
 asn_internal	30/01/2018 17:06	C/C++ Header
 asn_ioc	30/01/2018 17:06	C/C++ Header
 asn_random_fill	30/01/2018 17:06	Ficheiro C
 asn_random_fill	30/01/2018 17:06	C/C++ Header
 asn_SEQUENCE_OF	30/01/2018 17:06	Ficheiro C
 asn_SEQUENCE_OF	30/01/2018 17:06	C/C++ Header
 asn_SET_OF	30/01/2018 17:06	Ficheiro C
 asn_SET_OF	30/01/2018 17:06	C/C++ Header
 asn_system	30/01/2018 17:06	C/C++ Header
 ber_decoder	30/01/2018 17:06	Ficheiro C
 ber_decoder	30/01/2018 17:06	C/C++ Header

Figure 1: Exemplo de ficheiros gerados.

## 1.1 Encoder

Seguidamente, procedemos à implementação do codificador, inicializando todas as estruturas existentes, extraídas através dos respetivos ficheiros `.h` presentes nas mesmas pastas, de modo a que fosse possível preenchê-las sempre que necessário.

```
ApplicationSyntax_t * createApplicationSyntaxUnsigned(Unsigned32_t unsigned_integer_value){  
    ApplicationSyntax_t *application;  
    application = calloc(1, sizeof(ApplicationSyntax_t));  
    application->present = ApplicationSyntax_PR_unsigned_integer_value;  
    application->choice.unsigned_integer_value = unsigned_integer_value;  
    return application;  
}  
  
//Passo 2.1 para object simple  
ObjectSyntax_t * createObjectSyntaxSimple(SimpleSyntax_t *simple){  
    ObjectSyntax_t *object_syntax;  
    object_syntax = calloc(1, sizeof(ObjectSyntax_t));  
    object_syntax->present = ObjectSyntax_PR_simple;  
    object_syntax->choice.simple = *simple;  
    return object_syntax;  
}  
  
//Passo 2.2 para object simple  
ObjectSyntax_t * createObjectSyntaxApplication(ApplicationSyntax_t *application){  
    ObjectSyntax_t *object_syntax;  
    object_syntax = calloc(1, sizeof(ObjectSyntax_t));  
    object_syntax->present = ObjectSyntax_PR_application_wide;  
    object_syntax->choice.application_wide = *application;  
    return object_syntax;  
}
```

Figure 2: Exemplo de inicialização das estruturas.

```

//Passo 3
ObjectName_t * createObjectName(uint8_t* name, size_t name_size){
    int i;
    ObjectName_t *object_name;
    object_name = calloc(1, sizeof(ObjectName_t));
    object_name->buf = name;
    object_name->size = name_size;
    //printf("\n%d\n", name_size);
    return object_name;
}

//Passo 4
VarBind_t * createVarBind(ObjectName_t *object_name, ObjectSyntax_t *object_syntax){

    VarBind_t * var_bind;
    var_bind = calloc(1, sizeof(VarBind_t));
    var_bind->name = *object_name;
    var_bind->choice.present = choice_PR_value;
    var_bind->choice.choice.value = *object_syntax;
    return var_bind;
}

//Passo 5
VarBindList_t * createVarBindList(VarBind_t * var_bind){

    VarBindList_t * varlist;
    varlist = calloc(1, sizeof(VarBindList_t));
    int r = ASN_SEQUENCE_ADD(&varlist->list, var_bind);
    return varlist;
}

```

Figure 3: Exemplo de inicialização das estruturas.

Nesse mesmo codificador foi criada uma função `main` cujo intuito será o de receção de dados, efetuando o seu `parsing` de modo a processar um comando recebido e a preencher as respetivas estruturas consoante o seu objetivo. Nesta mesma `main`, cada comando corresponderá a uma "escolha", sendo que preencherá as estruturas que lhe correspondem.

Por fim, e remetente à parte do mesmo codificador, toda a escrita é feita em ficheiro, sendo guardado o resultado do `encoding` num ficheiro em formato binário, de nome `encoded.bin`, e um ficheiro em modo `debug`, com uma sintaxe em XER (XML Encoding Rules), no ficheiro `debugE.xml`.

```

3025 0201 0204 0870 7269 7661 7465 00a0
1602 0101 0201 0002 0100 300b 3009 0604
c0a8 0001 0201 00

```

Figure 4: Exemplo de um ficheiro binário gerado.

```

<Message>
  <version>2</version>
  <community>70 72 69 76 61 74 65 00</community>
  <data>
    A0 16 02 01 01 02 01 00 02 01 00 30 0B 30 09 06
    04 C0 A8 00 01 02 01 00
  </data>
</Message>

```

Figure 5: Exemplo de um ficheiro XML gerado.

Apesar de poder ser possível escrever o OID na linha de comandos, a nossa aplicação não guarda este valor. Não conseguimos guardar o OID porque se trata de uma string e nas estruturas dadas pela definição do PDU não é possível guardar valores deste tipo.

## 1.2 Decoder

Uma vez o codificador realizado e ficheiro binário do comando gerado, resta agora descodificar o comando. No que toca ao descodificador este lê o ficheiro binário e "desconstrói" as estruturas de forma inversa ao codificador, ou seja, desde a estrutura `Message` até à estrutura mais simples, seja ela `Simple Syntax` ou `Application Syntax`.

Para tal colocamos a leitura do ficheiro num buffer e de seguida procedemos à reconstrução da estrutura `Message` a partir do método `asn_decode`. Depois reconstruímos as estruturas que estão incluídas na `Message` e no PDU.

```

fread(buffer, buffer_size, 1, file);
fclose(file);

//Passo 1
Message_t* message = 0;
asn_dec_rval_t rval = asn_decode(0, ATS_BER, &asn_DEF_Message, (void **)&message, buffer, buffer_size);

```

Figure 6: Construção da estrutura `Message`.

Além de reconstruir as estruturas também ter de as saber distinguir por modo a imprimir o comando corretamente e não criar estruturas desnecessárias. Dentro das estruturas existem campos onde é possível aceder sem se obter a estrutura na sua integridade. Campos esses que são `present` do PDU e da `Object Syntax`, este campo é bastante importante para poder identificar qual comando que se trata e quais os parâmetros que leva à frente.

```

switch(pdu->present){
case(PDUs_PR_get_request):
    command = "get";
    GetRequest_PDU_t getReq
    printf("Request id: %ld")
    printf("Error Index: %ld")
    printf("Error Status: %ld")
    var_bindings = pdu->cho
    break;
case(PDUs_PR_get_next_request):

```

Figure 7: Distinção de PDUs.

Por fim e por forma a fazer **debug** também criamos um ficheiro **xml** no decodificador para verificar se a leitura do ficheiro binário e a estrutura **Message** estavam bem construídas e verificar se está idêntico ao que o codificador criou.

```

<Message>
  <version>2</version>
  <community>70 75 62 6C 69 63 00 00</community>
  <data>
    .....
    A0 16 02 01 01 02 01 00 02 01 00 30 0B 30 09 06
    04 C0 A8 01 02 02 01 00
  </data>
</Message>

```

Figure 8: Exemplo de inicialização das estruturas.

O decodificador consegue interpretar quase todas as estruturas codificadas porém o nosso codificador não consegue estruturar certos comandos pelo que não temos a certeza do correto funcionamento para tais comandos. Além disso não consegue imprimir o OID pois o nosso codificador não coloca o OID numa estrutura.

### 1.3 Execução do código

Finalmente, e de modo a exemplificar um dos testes, utilizámos o comando `snmpget -v 2 -c public 192.168.1.2 system.sysUpTime.0`, devolvendo os seguintes resultados:

```

root@JPVS:/mnt/c/Users/jpedr/Documents/GitHub/GR/Encode# ./encoder snmpget -v 2 -c p
ublic 192.168.1.2 system.sysUpTime.0system.sysUpTime.0
First encode... done
Final encode... done
Binary written successfully into encoded.bin
XML written succesfully into debugE.xml
Success

```

Figure 9: Execução do comando no codificador.

```

3025 0201 0204 0870 7562 6c69 6300 00a0
1602 0101 0201 0002 0100 300b 3009 0604
c0a8 0102 0201 00

```

Figure 10: Ficheiro binário gerado pelo codificador.

```

<Message>
  <version>2</version>
  <community>70 75 62 6C 69 63 00 00</community>
  <data>
    A0 16 02 01 01 02 01 00 02 01 00 30 0B 30 09 06
    04 C0 A8 01 02 02 01 00
  </data>
</Message>

```

Figure 11: Ficheiro XML gerado pelo codificador.

```

root@JPVS:/mnt/c/Users/jpedr/Documents/GitHub/GR/Decode# ./decoder encoded.bin
Request id: 1
Error Index: 0
Error Status: 0
snmpget -v 2 -c public 192.168.1.2 OID

```

Figure 12: Descodificação do ficheiro previamente codificado e respetivos resultados.

Conforme dito anteriormente, o decodificador não imprime o OID pelo que imprime uma string a dizer OID.