Network Working Group
Internet-Draft
Expires: October 10, 2001

J. Schoenwaelder
TU Braunschweig
April 11, 2001

SNMP Payload Compression
draft-irtf-nmrg-snmp-compression-01.txt

Status of this Memo

Copyright Notice

Abstract

This memo defines a mechanism for lossless compression of SNMP payloads. Compression is especially useful when retrieving large amounts of data or when SNMP encryption is being used over a transport which provides data compression.

Table of Contents

1. Introduction

This memo defines mechanisms for lossless compression of SNMP payloads. Compression is useful when retrieving large amounts of management data since the BER encoding used by SNMP is not very space efficient and the payload tends to have a high degree of redundancy.

SNMP payload compression is especially useful when retrieving large amounts of data. In particular, compression allows command responders to put more data into responses to bulk retrieval requests, which can significantly reduce the overall number of transactions needed to retrieve a certain amount of data [5].

SNMP payload compression is also useful when SNMP encryption is used. Encrypting the SNMP payload causes the data to be random in nature, rendering compression at lower protocol layers (e.g., IP Payload Compression Protocol [2]) ineffective. If both compression and encryption are required, then compression should be applied before encryption.

2. Requirements

A solution for SNMP payload compression has to satisfy the following requirements:

1. Compression must happen before encryption if compression is used together with encryption. Compression is most useful if there are regular patterns in the data. It is the nature of encryption algorithms to destroy any regular pattern and hence encrypted data does not compress very well.

2. SNMP payload compression should be able to support multiple compression algorithms. This implies that SNMP engines must be able to negotiate the compression algorithm they are using. Instead of carrying a compression algorithm identifier in every protocol message, it seems more effective to indicate compression algorithms in a MIB module (similar to authentication or encryption algorithms in SNMPv3).

3. Each SNMP message is compressed and decompressed by itself without any relation to other SNMP messages ("stateless compression"), as SNMP messages may arrive out of order or not arrive at all.

4. The size of a compressed SNMP message must never exceed the size of the uncompressed SNMP message ("non-expansion policy"). A sender may send an uncompressed SNMP message if an attempt to compress the message produces a result which is longer than the

uncompressed SNMP message. An SNMP engine configured to receive compressed SNMP messages must thus also accept uncompressed SNMP messages.

5. The abstract syntax of compressed SNMP messages must be defined using ASN.1. This ensures that a compressed SNMP message has a valid ASN.1/BER encoding which simplifies the integration into existing SNMP toolkits.

6. It is desirable to define common compression algorithms in order to achieve interoperability. The compression algorithms should be openly available and they should represent different trade-offs between compression efficiency and CPU efficiency.

3. Identifying Compressed Payload

It is necessary to distinguish between compressed and uncompressed SNMP payload. There are several ways to implement this. The following sections discuss alternatives that have been considered.

3.1 Compression as an SNMPv3 Encryption Algorithm

The idea behind the first approach is to treat compression as an additional SNMPv3 encryption algorithm. In fact, compression as well as encryption can both be treated as a loss-less data transformation. This approach has the following advantages / disadvantages:

+ No change required to the SNMPv3 specifications.

+ Compression of the complete ScopedPDU.

- Support of N encryption algorithms and M compression algorithms leads to N*M possible combinations.

- Compression requires authentication since there is no noAuthPriv security level.

- Does not work with older and widely deployed versions of SNMP (SNMPv1, SNMPv2c).

3.2 Indicating Compression in the msgFlags

To avoid some of the drawbacks of the previous approach, one can treat compression independent of encryption by allocating an unused bit in the msgFlags [3]) to indicate whether compression is used or not. However, RFC 2572 [3]) says in section 6.4:

The remaining bits in msgFlags are reserved, and must be set to zero when sending a message and should be ignored when receiving a message.

Similarly, RFC 2572 [3] specifies in section 7.1 step 7) and in section 7.2 step 5) that other bits in the msgFlags are set to zero or ignored. This means that this alternative can not be supported by an implementation which is strictly compliant to RFC 2572 [3].

In summary, this approach has the following advantages / disadvantages:

+ Combination of M compression and N encryption algorithms possible without having to define N*M algorithms.

+ Compression can be used with or without encryption or authentication.

+ Compression of the complete ScopedPDU.

- Not strictly compliant to the current SNMPv3 specifications.

- Does not work with older widely deployed versions of SNMP (SNMPv1, SNMPv2c).

3.3 Compression as a new PDU Type

The third alternative is to restrict compression to PDUs rather than ScopedPDUs and to introduce a new PDU type for compressed payloads. RFC 1157 [4] defines the SNMPv1 message header as follows:

```
Message ::= SEQUENCE {
    version   INTEGER { version-1(0) },
    community OCTET STRING,
    data      ANY  -- e.g., PDUs if trivial authentication
                -- is being used
}
```

Similarly, RFC 2572 [3] defines the ScopedPDU as follows:

```
ScopedPDU ::= SEQUENCE {
    contextEngineID  OCTET STRING,
    contextName      OCTET STRING,
    data             ANY -- e.g., PDUs as defined in RFC 1905
}
```

This means that a new PDU could be defined which holds the compressed

version of a PDU:

```
CompressedPDU ::= [42] IMPLICIT OCTET STRING
                  -- contains a compressed PDU
```

It is important to analyze how a compliant SNMP implementation behaves when it receives an unknown PDU type. From a formal point of view, any PDU which is a valid BER serialization of an ASN.1 type must be accepted since the data portion is of the ASN.1 type ANY. In practice, most SNMP implementations will only recognize the PDU types defined in the SNMP specifications.

The SNMPv3 message processing model [3] defines in section 7.2 step 7) that parse errors while decoding the ScopedPDU cause the packet to be discarded after incrementing snmpInASNParseErrs. Even an implementation which is capable to decode arbitrary PDUs will have problems to determine the pduType as defined in section 7.2 step 9). This basically means that a compliant SNMPv3 engine will simply discard compressed PDUs.

The SNMPv1 specification [4] defines in section 4.1 second step (4) that parse errors while decoding the PDU will cause the SNMP engine to drop the PDU. Hence, it can be expected that most implementations will simply drop a compressed PDU.

In summary, this approach has the following advantages / disadvantages:

+ Combination of M compression and N encryption algorithms possible without having to define N*M algorithms.

+ Compression can be used with or without encryption or authentication.

+ Works with every version of SNMP.

- Not strictly compliant to the current SNMPv3 specifications.

- Compression of the PDU rather than the ScopedPDU.

4. Negotiating Compression Algorithms

[TBD]

5. Compression Algorithms

5.1 DEFLATE

The DEFLATE algorithm is a well-know compression algorithm which achieves good compression ratios and which is used for example in RFC 2394 for IP payload compression. It is also used on the World Wide Web to compress documents before transmission over HTTP.

Using DEFLATE for SNMP payload compression however shows some undesirable aspects. First, DEFLATE compression is relatively CPU intensive. Furthermore, the DEFLATE algorithm requires to transmit a dictionary, which reduces the compression efficiency for small messages. On the other hand, DEFLATE compresses both names and values and may achieve particularly good compression if the encoded values show a similar structure.

Experiments with DEFLATE show that it should not be used on relatively short SNMP messages. Furthermore, DEFLATE introduces significant delays due to the compression computations. Finally, estimating message sizes is hard with DEFLATE since there is no upper bound for the message length addition if one adds another varbind. This has impacts in particular on the getbulk PDU implementation. It is therefore not recommended to adopt DEFLATE compression.

5.2 OID Delta Compression (ODC)

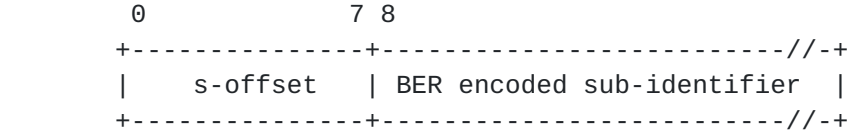
SNMP payloads use OIDs to represent the names of SNMP variables. The amount of space used for encoding these OIDs frequently exceeds the space needed to represent the values identified by the OIDs. Furthermore, subsequent OIDs usually have many sub-identifier in common.

The OID Delta Compression (ODC) algorithm has been designed to reduce the OID overhead inherent in SNMP messages. The general idea is to encode an OID as a delta to the previous OID. The ODC algorithm only achieves a reduction in the SNMP naming information. It does not compress the data of MIB variables, even if the value itself is an OID. (The reason for not compressing OID values is that they are (a) relatively infrequent and (b) compressing value OIDs has negative impact on the compression achieved for the following variable name.) In many cases (e.g., when retrieving large tables which basically contain numbers), the achieved compression ratio is significant while the CPU cycles spent on the compression algorithm itself are very small.

5.2.1 ODC Algorithm

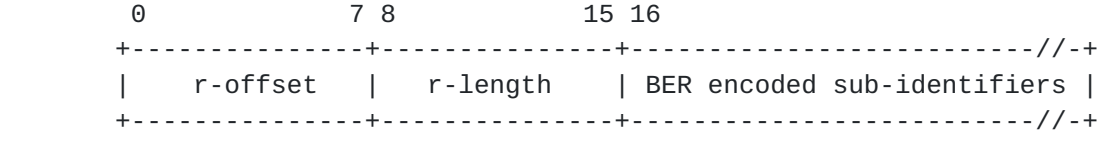
The ODC algorithm encodes OID deltas using three mechansisms:

1. Substitution of a single sub-identifier values at a certain position. A sub-identifier substitution is encoded as follows:



s-offset Defines the offset of the sub-identifier to substitute. The offset value is in the range 0-0x7f. The value 0 refers to the first OID sub-identifier.

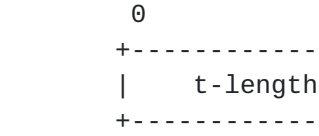
2. Substitution of ranges of sub-identifiers at a given starting position. A substitution of a range of sub-identifiers is encoded as follows:



r-offset Defines the offset of the sub-identifier range to substitute. The offset value has the most significant bit set and is in the range 0x80-0xff. The value 0x80 refers to the first OID sub-identifier.

r-length Defines the number of BER encoded sub-identifiers that follow the r-length field and which will be substituted. The range of the r-length field is 0x01-0x7f.

3. Truncation of OIDs (which may shorten or enlarge OIDs). A truncation, which may only appear at the end, is encoded as follows:



t-length Defines the new length of the OID in the range 0x01-0x7f. The t-length value specifies the number if sub-identifiers minus 1. Hence, the value 0x01 identifies an OID which consists of two sub-identifiers. Truncation may be used to shorten or enlarge an OID. New sub-identifiers will have the value 0 if an OID is enlarged.

An ODC compressed OID is simply the combination of several sub-identifier or sub-identifier range substitutions followed by an optional truncation. Note that substitutions can increase the size of the OID if the offset or range specifies sub-identifier positions outside of the previous OID. New sub-identifiers without an explicit value assignement will have the value 0. An ODC compressed OID is distinguished from a normal OID by introducing the following new ASN.1 type:

```
CompOID ::= [42] IMPLICIT OCTET STRING
```

A high-level description of the compression algorithm is as follows:

1. Loop through the SNMP PDU until you find an OID name value pair (varbind).

2. If it is the first varbind, make a copy of the OID, pass it to the output buffer and continue with the next varbind.

3. Otherwise, compute the delta to the last OID and BER encode it into the CompOID value.

4. If the CompOID representation is larger than the OID, pass the OID to the output buffer, else pass the CompOID to the output buffer.

5. Update the last OID and goto step 2 if there are additional varbinds.

Some SNMP implementations use a reverse encoding scheme where PDUs are encoded from the end to the beginning. The ODC algorithm can also be used efficiently in this case by using an OID look-ahead of 1 varbind.

A high-level description of the decompression algorithm is as follows:

1. Loop through the SNMP PDU until you find an OID name value pair (varbind).

2. If the varbind name contains an uncompressed OID, pass it to the output buffer and continue with the next varbind.

3. Otherwise, if the varbind name contains a compressed OID, loop through the compressed OID value doing the following:

    1. If the first byte is in the range 0-0x7f and there are more bytes, then decode the following byte(s) as a BER encoded

sub-identifier and perform a sub-identifier substitution.

2. If the first byte is in the range 0x80-0xff, then read the following byte as the r-length value. Decode the following r-length BER encoded sub-identifier and perform a range substitution.

3. If the first byte is in the range 0x01-0x7f and there are no more bytes, then perform a truncation.

4. Pass the decoded OID to the output buffer.

5. Update the last OID and goto step 2 if there are additional varbinds.

5.2.2 ODC Examples

This section shows some example ODC encodings. It is provided to better understand how ODC encodings work. It is not intended to give a formal analysis of the compression ratios that can be achieved on arbitrary SNMP payload.

5.2.2.1 Simple Substitutions

Lets assume a command generator uses getbulk operations to retrieve the tcpConnTable of the TCP-MIB. A good manager will do that by retrieving the tcpConnState column. The command responder will return a sequence of tcpConnState (1.3.6.1.2.1.6.13.1.1) values:

```
tcpConnState.0.0.0.0.21.0.0.0.0.0 = listen(2)
tcpConnState.0.0.0.0.22.0.0.0.0.0 = listen(2)
tcpConnState.0.0.0.0.23.0.0.0.0.0 = listen(2)
tcpConnState.0.0.0.0.98.0.0.0.0.0 = listen(2)
```

The BER encoding of this varbind list is the following sequence of bytes:

```
30:18                                  // sequence
   06:13:2B:06:01:02:01:06:0D:01:01:   // tcpConnState
         00:00:00:00:15:00:00:00:00:00 // instance identifier
   02:01:02                            // value
30:18                                  // sequence
   06:13:2B:06:01:02:01:06:0D:01:01:   // tcpConnState
         00:00:00:00:16:00:00:00:00:00 // instance identifier
   02:01:02                            // value
30:18                                  // sequence
   06:13:2B:06:01:02:01:06:0D:01:01:   // tcpConnState
```

```
         00:00:00:00:17:00:00:00:00:00  // instance identifier
   02:01:02                             // value
30:18                                   // sequence
   06:13:2B:06:01:02:01:06:0D:01:01:    // tcpConnState
         00:00:00:00:62:00:00:00:00:00  // instance identifier
   02:01:02                             // value
```

Using ODC compression, the following sequence of bytes would be used:

```
30:18                                   // sequence
   06:13:2B:06:01:02:01:06:0D:01:01:    // tcpConnState
         00:00:00:00:15:00:00:00:00:00  // instance identifier
   02:01:02                             // value
30:07                                   // sequence
   2a:02:0E:16                          // substitution
   02:02:02                             // value
30:07                                   // sequence
   2a:02:0E:17                          // substitution
   02:01:02                             // value
30:07                                   // sequence
   2a:02:0E:62                          // substitution
   02:01:02                             // value
```

In this particular example, the total size of the encoded varbind list drops from 104 bytes to 53 bytes.

5.2.2.2 Range Substitutions

This example expands the previous example by showing how range substitutions work. In this example, we assume that the command responder will return a sequence of tcpConnState values with different IP addresses in the instance identifier:

```
tcpConnState.134.169.34.190.50054.130.240.64.53.80 = closeWait(8)
tcpConnState.134.169.34.190.50366.212.185.76.85.80 = closeWait(8)
tcpConnState.134.169.34.190.53370.134.169.34.117.6000 = established(5)
tcpConnState.134.169.34.190.56398.134.169.34.190.120 = closeWait(8)
```

The BER encoding of this varbind list is the following sequence of bytes:

```
30:1F                                   // sequence
   06:1A:2B:06:01:02:01:06:0D:01:01:    // tcpConnState
      81:06:81:29:22:81:3E:83:87:06:    // instance
      81:02:81:70:40:35:50              // identifier
   02:01:08                             // value
30:1F                                   // sequence
   06:1A:2B:06:01:02:01:06:0D:01:01:    // tcpConnState
```

```
      81:06:81:29:22:81:3E:83:89:3E:    // instance
      81:54:81:39:4C:55:50              // identifier
   02:01:08                             // value
30:20                                   // sequence
   06:1B:2B:06:01:02:01:06:0D:01:01:    // tcpConnState
      81:06:81:29:22:81:3E:83:A0:7A:    // instance
      81:06:81:29:22:75:AE:70           // identifier
   02:01:05                             // value
30:20                                   // sequence
   06:1B:2B:06:01:02:01:06:0D:01:01:    // tcpConnState
      81:06:81:29:22:81:3E:83:B8:4E:    // instance
      81:06:81:29:22:81:3E:78           // identifier
   02:01:08                             // value
```

Using ODC compression, the following sequence of bytes would be used:

```
30:1F                                   // sequence
   06:1A:2B:06:01:02:01:06:0D:01:01:    // tcpConnState
      81:06:81:29:22:81:3E:83:87:06:    // instance
      81:02:81:70:40:35:50              // identifier
   02:01:08                             // value
30:10                                   // sequence
   2A:0B:8E:05:                         // range
   83:89:3E:81:54:81:39:4C:55           // substitution
   02:01:08                             // value
30:12                                   // sequence
   2A:0D:8E:06:                         // range
   83:A0:7A:81:06:81:29:22:75:AE:70     // substitution
   02:01:05                             // value
30:0E                                   // sequence
   2A:09:0E:83:A0:7A                    // substitution
   92:02:81:3E:78                       // range substitution
   02:01:08                             // value
```

In this particular example, the total size of the encoded varbind list drops from 134 bytes to 87 bytes.

5.2.2.3 Substitutions and Truncations

This example assumes that a command generator retrieves the ipNetToMediaTable defined in the IP-MIB using a getbulk on ipNetToMediaPhysAddress (1.3.6.1.2.1.4.22.1.2) and ipNetToMediaType (1.3.6.1.2.1.4.22.1.4) pairs. The following values might be returned for a system which only has one entry in the cache:

```
ipNetToMediaPhysAddress.2.224.8.8.0 = 01:00:5E:08:08:00
ipNetToMediaType.2.224.8.8.0 = dynamic(3)
ipNetToMediaNetAddress.2.224.8.8.0 = 224.8.8.0
```

ipRoutingDiscards.0 = 0

Note that the getbulk overshoots and retrieves the following instances in lexicographic order, which is an ipNetToMediaNetAddress (1.3.6.1.2.1.4.22.1.3) and an ipRoutingDiscards (1.3.6.1.2.1.4.23) instance.

The BER encoding of this varbind list is the following sequence of bytes:

```
30:19                                  // sequence
   06:0F:2B:06:01:02:01:04:16:01:02:   // ipNetToMediaPhysAddress
      02:81:60:08:08:00                // instance identifier
   04:06:01:00:5E:08:08:00             // value
30:14                                  // sequence
   06:0F:2B:06:01:02:01:04:16:01:04:   // ipNetToMediaType
      02:81:60:08:08:00                // instance identifier
   02:01:03                            // value
30:17                                  // sequence
   06:10:2B:06:01:02:01:04:16:01:03:   // ipNetToMediaNetAddress
      02:81:60:08:08:00                // instance identifier
   40:04:E0:08:08:00                   // value
30:0D                                  // sequence
   06:08:2B:06:01:02:01:04:17          // ipRoutingDiscards
   00                                  // instance identifier
   41:01:00                            // value
```

Using ODC compression, the following sequence of bytes would be used:

```
30:19                                  // sequence
   06:0F:2B:06:01:02:01:04:16:01:02:   // ipNetToMediaPhysAddress
      02:81:60:08:08:00                // instance identifier
   04:06:01:00:5E:08:08:00             // value
30:07                                  // sequence
   2A:02:09:04                         // substitution
   02:01:03                            // value
30:0A                                  // sequence
   2A:02:09:04                         // substitutions
   40:04:E0:08:08:00                   // value
30:0A                                  // sequence
   2A:05:87:02:23:00:                  // range substitution
      08                               // truncation
   41:01:00                            // value
```

In this particular example, the total size of the encoded varbind list drops from 89 bytes to 60 bytes.

6. Acknowledgments

This document is the result of discussions within the Network Management Research Group (NMRG). of the Internet Research Task Force[6] (IRTF). Special thanks go to Luca Deri, Wes Hardacker, Jean-Philippe Martin-Flatin, Joe Marzot, Aiko Pras, Ron Sprenkels, Frank Strauss, and Bert Wijnen for their comments and suggestions.

References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Shacham, A., Monsour, R., Pereira, R. and M. Thomas, "IP Payload Compression Protocol (IPComp)", RFC 2393, December 1998.

[3] Case, J., Harrington, D., Presuhn, R. and B. Wijnen, "Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)", RFC 2572, April 1999.

[4] Case, J., Fedor, M., Schoffstall, M. and J. Davin, "A Simple Network Management Protocol (SNMP)", STD 15, RFC 1157, May 1990.

[5] Sprenkels, R. and J. Martin-Flatin, "Bulk Transfers of MIB Data", Simple Times 7(1), March 1999.

[6] <http://www.irtf.org/>

Author's Address

Juergen Schoenwaelder
TU Braunschweig
Bueltenweg 74/75
38106 Braunschweig
Germany

Phone: +49 531 391-3289
EMail: schoenw@ibr.cs.tu-bs.de

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.