Internet Engineering Task Force
INTERNET-DRAFT
Expires October 2001

Sandra McLeod
SNMP Research
David Partain
Matt White
Ericsson
April 2001

SNMP Object Identifier Compression
draft-ietf-eos-oidcompression-00.txt
Revision 1.9
Document Date: 2001/04/23 21:31:04

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet- Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Abstract

This memo defines a mechanism, called OID Compression, for removal of redundant information in the object identifiers (OIDs) carried in the name portion of variable bindings in SNMP messages.

1. Background

OID Compression reduces the size of SNMP PDUs by reducing the amount of redundant information contained within the Object Identifiers (OIDS) in the name portion of multiple variable bindings in the same SNMP PDU. The hierarchical structure of OIDs makes it likely that the OIDs in a message will share some common base set of subidentifiers. This is particularly true for related objects from the same groups of MIB scalars or tables. Since SNMP messages often contain numerous related OIDs, most SNMP messages will contain some amount of redundant information in the name portion of multiple variable bindings in the variable binding list. The compression of these OIDs could result in substantial savings in the amount of encoding and space required to build SNMP PDUs.

This memo addresses the compression of OIDs within the name portion of PDU varbinds only. The inclusion of Object Identifiers in the value portion of PDU varbinds is not nearly as common and would not likely result in substantial savings.

2. Previous work

There have been a number of OID Compression techniques that have been discussed and presented in the SNMP community over many years, including those presented in the IRTF document "SNMP Payload Compression" by Juergen Schoenwaelder and in the "GetCols Operation" presentation by David Perkins. A number of OID Compression techniques that have been presented to date approach OID compression by encoding OIDs as a delta OID value between the current OID in a varbind name and a previous OID within the same SNMP PDU.

Previous Delta OID Compression techniques vary substantially in concept and implementation. However, most Delta OID Compression algorithms identify "anchor" OIDs that are used for comparison against the current OID to calculate a smaller delta OID which will be encoded in the SNMP PDU. The definition of "anchor" OIDs in previous work can vary quite a bit and may be static or dynamic.

There can be one or many "anchor" OIDs per PDU. The "anchor" OIDs can be either explicitly or implicitly identified in the PDUs. The algorithms that have been used to generate the delta OIDs between these "anchor" OIDs and the OIDs in the varbind lists also vary substantially from one approach to another. As a result, the different delta OID compression techniques vary substantially in complexity and efficiency.

With the many choices of Delta OID Compression techniques, the ultimate compression technique should provide a simple algorithm that is unambiguous in its implementation and provides good OID compression returns. The algorithm detailed in this memo provides a simple Delta OID Compression algorithm with good OID compression returns. While providing additional choices in how OIDs are compressed might provide somewhat more compression but at a complexity of implementation and processing that may not be worth the cost. The algorithm below attempts to strike a balance between efficiency of compression and minimization of processing complexity.

3. Terminology

The following is a description and clarification of what is intended by the terms "compression" and "suppression" as applied to SNMP OIDs:

* Compression

  OID compression is the reduction in the amount of information required to represent each OID in the name portion of an SNMP PDU varbind. The term compression implies a one-to-one mapping between the original OIDs in the name portions of a varbind list and the resulting set of compressed OIDs for the same varbind list. In the case of OID compression, OIDs are not omitted in the varbind lists, but rather are simply reduced in size with redundant information removed in the name portion of the varbind list.

* Suppression

  The suppression of Object Identifiers indicates a reduction in the number of identifiers required to represent a series of OIDs in an SNMP PDU varbind list. The term suppression implies a many-to-one mapping between the original set of OIDs in a varbind list and the resulting set of encoded OIDs in an SNMP PDU.

4. OID compression versus OID suppression

OID compression and suppression techniques differ in their approach to reducing redundant information and when used individually are most useful under different operational circumstances. There are circumstances, however, in which the use of both of these techniques in combination can be beneficial as well.

OID Compression reduces the amount of redundant information in a series of OIDs without omitting any OIDs in the encoded SNMP PDU. This technique can potentially be used to compress any group of OIDs in the same SNMP PDU. This makes OID compression well-suited for operations such as MIB walks, bulk data retrieval, and large configuration operations where a large percentage of the OIDs in the varbind list have some common base set of subidentifiers.

OID Suppression, on the other hand, reduces the amount of redundant information in a series of OIDs by reducing the number of identifiers that must be included in the SNMP PDU. OID suppression is particularly well-suited for table row operations (retrieval or configuration) where most of the OIDs share the same base table OID as well as instance information. In this case, the base table OID and instance information can be provided once for all of the objects in the same row.

OID Suppression will be introduced in Appendix A (and will likely be moved to a separate document in future revisions). The focus of the rest of this memo will be exclusively on OID Compression.

5. Delta OID Compression Algorithm

The Delta OID Compression algorithm discussed in this memo provides a simple technique for reducing the amount of redundant information in multiple varbind names within the same PDU. This algorithm calculates the delta OIDs by comparing each varbind name to the previous varbind name within a varbind list in a single SNMP PDU. The delta OID that is calculated between the current and previous OIDs identifies a simple OID tail replacement. The delta OID that is generated from this algorithm will contain first a value which identifies the position in the current OID at which this OID diverges from the previous OID and then following this would be the remaining subidentifiers from that position forward in the current OID.

For encoding purposes, in order to remain consistent with the current

Object Identifier encoding rules, the first value in the delta OID which identifies the subidentifier position at which the current and previous OIDs diverge will need to be split into two subidentifiers and represented in the delta OID as the first and second subidentifiers of that OID. The formula for determining the first two subidentifiers in the delta OID is as follows:

```
S1 =  ( Position at which the OIDs diverge) / 40
S2 =  ( Position at which the OIDs diverge) % 40
```

In the formula above, S1 and S2 are the subidentifier values to be determined for the delta OID. S1 will be assigned to the first subidentifier in the delta OID and is calculated as the result of the positional value divided by 40. S2 will be assigned to the second subidentifier in the delta OID and is calculated as the result of the positional value modulo 40.

Because the length of an OID is limited to a maximum of 128, the positional value that indicates the position at which the current and previous OIDs diverge must be between 0 and 128. As a result, the first subidentifier could only possibly have the values of 0, 1, 2, or 3. The second subidentifier, which represents the modulo of 40 could only possibly have a value in the range of 0 to 39. In the interest of maintaining compatibility with current SNMP implementations which may only expect values of 0, 1, or 2 (but not 3) for the first subidentifier, it is suggested that this algorithm could be restricted for use only in compressing OIDs for which the position at which the current and previous OIDs diverge is 119 or less. The result of this restriction would be that the first subidentifier would always have a value of 0, 1, or 2.

The OID Compression mechanism that is described above may only be used in new PDUs being defined in separate working group documents. That is, OID Compression MUST not be used in the GetRequest-PDU, GetNextRequest-PDU, GetBulkRequest-PDU, Response-PDU, SetRequest-PDU, InformRequest-PDU, SNMPv2-Trap-PDU, or Report-PDU.

5.1. Examples of Compressed OIDs

Example 1: Retrieval of the system group in a single message might result in following partial list of varbind names being

sent:

```
 1.3.6.1.2.1.1.1.0  -- sysDescr.0
 1.3.6.1.2.1.1.2.0  -- sysObjectID.0
 1.3.6.1.2.1.1.3.0  -- sysUpTime.0
 1.3.6.1.2.1.1.4.0  -- sysContact.0
```

If this payload were compressed, the resulting list of compressed delta OIDs would be as follows:

```
 1.3.6.1.2.1.1.1.0  -- first OID in varbind list is not compressed
 0.8.2.0            -- tail of the next OID starting at the 8th subid
 0.8.3.0            -- tail of the next OID starting at the 8th subid
 0.8.4.0            -- tail of the next OID starting at the 8th subid
```

Example 2: Retrieval of a selection of MIB-II objects might result in the following list of varbind names being sent:

```
 1.3.6.1.2.1.1.1.0      -- sysUpTime.0
 1.3.6.1.2.1.2.2.1.8.1  -- ifOperStatus.1
 1.3.6.1.2.1.2.2.1.8.2  -- ifOperStatus.2
 1.3.6.1.2.1.2.2.1.10.1 -- ifInOctets.1
 1.3.6.1.2.1.2.2.1.10.2 -- ifInOctets.2
 1.3.6.1.2.1.6.5.0      -- tcpActiveOpens.0
 1.3.6.1.2.1.6.7.0      -- tcpAttemptFails.0
 1.3.6.1.2.1.6.8.0      -- tcpEstabResets.0
```

If this payload were compressed, the resulting list of compressed delta OIDs would be as follows:

```
 1.3.6.1.2.1.1.1.0  -- first OID in varbind list is not compressed
 0.7.2.2.1.8.1      -- tail of next OID starting at the 7th subid
 0.11.2             -- tail of next OID starting at the 11th subid
 0.10.10.1          -- tail of next OID starting at the 10th subid
 0.11.2             -- tail of next OID starting at the 11th subid
 0.7.6.5.0          -- tail of next OID starting at the 7th subid
 0.8.7.0            -- tail of next OID starting at the 8th subid
 0.8.8.0            -- tail of next OID starting at the 8th subid
```

5.2. Applicability of the Algorithm

In cases where the objects requested in the same PDU are from multiple MIB groups then there is likely to be a larger cost, meaning a larger delta OID, between the OIDs that cross into the different MIB groups. However, since this algorithm makes use of a dynamic rather than a static "anchor" (which is always the previous varbind name) then this allows you to take advantage of the fact that related objects in requests are commonly grouped together, which allows for greater compression savings.

This Delta OID Compression approach was chosen primarily for its simplicity. This algorithm is well-suited for scalar objects as well as for tabular objects when performing table walks by column. However, this algorithm may be suboptimal in cases where a row of objects are being retrieved from a table, especially in cases in which the table is multiply indexed or indexed by OctetStrings. For instances of individual objects in the same row in a table, the table portion of the OIDs can be compressed but the instance information in these OIDs will not be compressed since two consecutive OIDs representing instances of different columns in a table will always diverge at the column subidentifier within the OIDs. As an example, consider the following two OIDs:

```
1.3.6.1.2.1.4.1.22.1.2.1.192.147.142.35
                   ^^^  -- ipNetToMediaPhysAddress.1.192.147.142.35
1.3.6.1.2.1.4.1.22.1.4.1.192.147.142.35
                   ^^^  -- ipNetToMediaPhysType.1.192.147.142.35
```

In this case the instance information is obviously redundant but cannot be compressed because the first subidentifier at which these two OIDs will begin to diverge is the subidentifier with the column identification ('2' for ipNetToMediaPhysAddress and '4' for ipNetToMediaPhysType). Appendix A discusses the use of OID suppression to address this case. It is anticipated that OID suppression can be used independently or in combination with OID compression to optimize the reduction of redundant information in the name portion of variable bindings in SNMP PDUs.

6. Use of OID Compression in Notifications

There are at least two reasons why it may not be useful to use OID compression when sending notifications (traps and informs). These reasons are:

1. Unlike a command generator, the notification originator has no way of knowing in advance whether the receiver is capable of parsing compressed OIDs.

   Obviously, one could extend the SnmpTargetAddrTable to include this capability, which is an option the working group may wish to consider.

2. It is unclear that there is significant benefit in using OID compression in notifications given the fact that they generally carry a small number of varbinds.

7. Special Considerations

The delta OID that is used to replace the OID must be no greater in length than the actual encoded OID that it represents in order for any benefit to be realized. In cases in which the encoded compressed OID would be greater than in length than the original encoded OID then the original OID encoding should be used.

8. Encoding of compressed OIDs

We identify compressed OIDs with two new tags such as the following (tag numbers will be finalized later):

```
uncompressedDeltaIdentifier ::=
  [APPLICATION 14]
     IMPLICIT OBJECT IDENTIFIER (SIZE (0..128)) -- Max OID length

compressedDeltaIdentifier ::=
  [APPLICATION 15]
     IMPLICIT OBJECT IDENTIFIER (SIZE (0..128)) -- Max OID length
```

The "uncompressedDeltaIdentifier" would be used for OIDs such as the first OID in a varbind list that are not compressed but are included in a PDU that could or should contain compressed OIDs if more than one varbind is provided in the varbind list.

The "compressedDeltaIdentifier" would be used for OIDs that are actually compressed in a varbind list of more than one varbind.

As a result of the new compression OID types we need to add a choice to the ObjectName syntax. This might be redefined to look similar to

the following:

```
ObjectName ::=
   CHOICE {
        nonDeltaOID             OBJECT IDENTIFIER
        uncompressedDeltaOID    uncompressedDeltaIdentifier
        compressedDeltaOID      compressedDeltaIdentifier
   }
```

The new ObjectName provides a choice of an original OBJECT IDENTIFIER which should not be used in PDUs with compression applied or one of the two delta OID types which will either be compressed or uncompressed Delta OIDs Object Identifier.

The "nonDeltaOID" and "compressedDeltaOID" tags would provide an unambigious way to distinguish between compressed and uncompressed OIDs. The purpose of the "uncompressedDeltaOID" is to allow command generators to indicate a willlingness to received compressed OIDs in response to their requests. Typically, the use of compressed OIDs in the request would indicate to the command responder that compressed OIDs in the response is desirable. However, if the command generator issues a GetBulk request with a single varbind included in the request, then this OID cannot be compressed, and the command generator cannot explicitly use compression in its request to indicate to the command resonder that it wishes to receive compressed OIDs in the response from the agent. However, if we define an additional uncompressedDeltaIdentifier type object name tag, then we can indicate that this specific OID is not compressed but compression in the response is specifically requested. All 3 ObjectName types would be valid for use with new PDUs. The "nonDeltaOID" object name type would only be used if compression was specifically not desired.

Note that the two new APPLICATION types are not included in the ObjectSyntax choices since compressed OIDs are not valid for use in the value portion of a varbind.

9. When to use compressed versus uncompressed OIDs

It is up to the command generator to determine whether the request should be made with compressed OIDs. If the command responder receives a request with one of the new PDU types then this implies that the request originator is capable of supporting OID compression. If the request received contains either compressed or uncompressed

delta OIDs, the response SHOULD contain compressed OIDs.

10. Encoding Examples

For the following set of Varbind names:

```
1.3.6.1.2.1.25.1.1.0      -- hrSystemUptime.0
1.3.6.1.2.1.25.1.5.0      -- hrSystemNumUsers.0
1.3.6.1.2.1.25.1.6.0      -- hrSystemProcesses.0
1.3.6.1.2.1.25.1.7.0      -- hrSystemMaxProcesses.0
1.3.6.1.2.1.25.3.2.1.2.1  -- hrDeviceType.1
1.3.6.1.2.1.25.3.2.1.3.1  -- hrDeviceDescr.1
1.3.6.1.2.1.25.3.2.1.5.1  -- hrDeviceStatus.1
```

The following is the uncompressed version of a PDU containing these varbinds with values:

```
30        UNIVERSAL [16] SEQUENCE OF constructor(VarBindList)
81 98     length = 152
   30         UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   11         length = 17
      06        UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      09        length = 9
      2b 06 01 02 01 19 01 01 00  = 1.3.6.1.2.1.25.1.1.0
                                  = hrSystemUptime.0
      43        APPLICATION [3] IMPLICIT INTEGER (ObjectSyntax)
      04        length = 4
      02 7c 19 74  =  41687412
   30         UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   0e         length = 14
      06         UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      09         length = 9
      2b 06 01 02 01 19 01 05 00   = 1.3.6.1.2.1.25.1.5.0
                                   = hrSystemNumUsers.0
      42         APPLICATION [2] IMPLICIT INTEGER (ObjectSyntax)
      01            length = 1
      14          = 20
   30         UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   0e         length = 14
      06          UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      09          length = 9
      2b 06 01 02 01 19 01 06 00   = 1.3.6.1.2.1.25.1.6.0
                                   = hrSystemProcesses.0
```

```
      42              APPLICATION [2] IMPLICIT INTEGER (ObjectSyntax)
      01              length = 1
      7e           = 126
   30         UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   0f         length = 15
      06           UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      09              length = 9
      2b 06 01 02 01 19 01 07 00   = 1.3.6.1.2.1.25.1.7.0
                                   = hrSystemMaxProcesses.0
      02              UNIVERSAL [2] INTEGER  (ObjectSyntax)
      02              length = 2
      0d ea        = 3562
   30         UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   18         length = 24
      06           UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      0b              length = 11
      2b 06 01 02 01 19 03 02 01 02 01   = 1.3.6.1.2.1.25.3.2.1.2.1
                                         = hrDeviceType.1
      06              UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      09              length = 9
      2b 06 01 02 01 19 03 01 03   = 1.3.6.1.2.1.25.3.1.3
   30         UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   26         length = 38
      06              UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      0b           length = 11
      2b 06 01 02 01 19 03 02 01 03 01 = 1.3.6.1.2.1.25.3.2.1.3.1
                                       = hrDeviceDescr.1
      04              UNIVERSAL [4] OCTET STRING (ObjectSyntax)
      17              length = 23
      53 75 6e 20 73 70 61 72 63 20 73 75 6e 34 6d 20 31 35 30 20 4d 48 7a
   30         UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   10         length = 16
      06              UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      0b              length = 11
      2b 06 01 02 01 19 03 02 01 05 01  = 1.3.6.1.2.1.25.3.2.1.5.1
                                        = hrDeviceStatus.1
      02           UNIVERSAL [2] INTEGER  (ObjectSyntax)
      01              length = 1
      02              = 2
```

The following is the compressed version of a PDU containing these varbinds with values.

```
1.3.6.1.2.1.25.1.1.0  -- first OID in varbind list is not compressed
```

```
   0.9.5.0              -- tail of next OID starting at the 9th subid
```

```
   0.9.6.0              -- tail of next OID starting at the 9th subid
   0.9.7.0              -- tail of next OID starting at the 9th subid
   0.8.3.2.1.2.1        -- tail of next OID starting at the 8th subid
   0.11.3.1             -- tail of next OID starting at the 11th subid
   0.11.5.1             -- tail of next OID starting at the 11th subid

   30       UNIVERSAL [16] SEQUENCE OF constructor(VarBindList)
   71       length = 113
      30             UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
      11          length = 17
         4e             APPLICATION [14] IMPLICIT OBJECT IDENTIFIER
                                              (nonCompressedDeltaOID )
         09              length = 9
         2b 06 01 02 01 19 01 01 00  = 1.3.6.1.2.1.25.1.1.0
                                     = uncompressed OID hrSystemUptime.0
         43              APPLICATION [3] IMPLICIT INTEGER (ObjectSyntax)
         04          length = 4
         02 7c 19 74  = 41687412
      30        UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
      08        length = 8
         4f             APPLICATION [15] IMPLICIT OBJECT IDENTIFIER
         03              length = 3
         09 05 00     = 0.9.5.0  -- compressed OID hrSystemNumUsers.0
                                    ==> 1.3.6.1.2.1.25.1.5.0
         42          APPLICATION [2] IMPLICIT INTEGER (ObjectSyntax)
         01          length = 1
         14           = 20
      30        UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
      08        length = 8
         4f             APPLICATION [15] IMPLICIT OBJECT IDENTIFIER
         03             length = 3
         09 06 00     = 0.9.6.0  -- compressed OID hrSystemProcesses.0
                                    ==> 1.3.6.1.2.1.25.1.6.0
         42          APPLICATION [2] IMPLICIT INTEGER (ObjectSyntax)
         01          length = 1
         7e          = 126
      30        UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
      09        length = 9
         4f          APPLICATION [15] IMPLICIT OBJECT IDENTIFIER
         03          length = 3
         09 07 00     = 0.9.7.0  -- compressed OID hrSystemMaxProcesses.0
                                    ==> 1.3.6.1.2.1.25.1.7.0
         02          UNIVERSAL [2] INTEGER  (ObjectSyntax)
         02          length = 2
```

```
          0d ea       = 3562
```

```
   30        UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   13        length = 19
      4f             APPLICATION [15] IMPLICIT OBJECT IDENTIFIER
      06          length = 6
      08 03 02 01 02 01   = 0.8.3.2.1.2.1  -- compressed OID hrDeviceType.1
                               ==> 1.3.6.1.2.1.25.3.2.1.2.1
      06          UNIVERSAL [6] OBJECT IDENTIFIER (ObjectName)
      09          length = 9
      2b 06 01 02 01 19 03 01 03   = 1.3.6.1.2.1.25.3.1.3
   30        UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   1e        length = 30
      4f             APPLICATION [15] IMPLICIT OBJECT IDENTIFIER
      03             length = 3
      0b 03 01    = 0.11.3.1    -- compressed OID hrDeviceDescr.1
                                ==>  1.3.6.1.2.1.25.3.2.1.3.1
      04          UNIVERSAL [4] OCTET STRING (ObjectSyntax)
      17          length = 23
      53 75 6e 20 73 70 61 72 63 20 73 75 6e 34 6d 20 31 35 30 20 4d 48 7a
   30        UNIVERSAL [16] SEQUENCE OF constructor (VarBind)
   08        length = 8
      4f          APPLICATION [15] IMPLICIT OBJECT IDENTIFIER
      03          length = 3
      0b 05 01    = 0.11.5.1    -- compressed OID hrDeviceStatus.1
                                  ==> 1.3.6.1.2.1.25.3.2.1.5.1
      02          UNIVERSAL [2] INTEGER  (ObjectSyntax)
      01          length = 1
      02          = 2
```

The total size of the uncompressed PDU was 155 bytes with 83 bytes required to encode the OIDs in the name portions of the variable bindings list.

The total size of the compressed PDU was 115 bytes with only 44 bytes required to encode the OIDs in the name portions of the variable bindings list. By applying the lightweight Delta OID Compression algorithm as described in this document to the variable binding list above a 47% reduction in the amount of space required to encode the OIDs for the names in the variable binding list was achieved and the total PDU size was reduced by 26% overall.

11. When to Compress OIDs

A command generator MAY send messages with compressed OIDs to a

command responder if the compressed OID message is expected to ellicit a response or the command responder has explicitly advertised the ability to support compressed OIDs. Responses to messages with compressed OIDs MUST use compressed OIDs if doing so will reduce the overall size of the response PDU.

If a command generator does not receive a response to a message with compressed OIDs and was expecting one, the message MUST be resent without OID compression unless the command responder has advertised, and the command generator read, the ability to process compressed payload messages. In the case where a command generator has determined a priori that a specific command responder is capable of processing compressed OID messages, the compressed OID message MAY be resent according to the implementation's retry mechanism.

Before generating messages that ellicit no response, a command generator MUST ascertain through advertised capabilities that the command responder is capable of processing compressed OIDs. Once the command generator has determined whether or not a particular command responder is capable of processing compressed OID messages, the command generator SHOULD cache the result and use this for future messages.

12. OID Compression with Proxy Forwarders

To be written: We must give serious consideration to how OID compression will function in an environment using proxy forwarding applications.

13. Security Considerations

To be written, assuming that what we are defining has some effect on security.

14. IANA Considerations

To be written: Probably nothing, but put here so that we don't forget about it.

15. Appendix A: OID Suppression

NOTE: This appendix is in this document for the purpose of comparison of the two approaches. It is anticipated that future companion EOS documents will includes some or all of these concepts, at which time this appendix will be removed.

OID Suppression reduces the amount of redundant information in a series of variable bindings by reducing the number of identifiers required to represent a series of OIDs in an SNMP PDU variable binding list. OID Suppression is particularly useful in the case of operations on columnar objects in which the OIDs share the same base table OID prefix and the same instance information.

Significant suppression of OIDs can be achieved through the definition of aggregate row objects which would allow multiple columns from the same conceptual row in a table to be bound together and represented as a single atomic unit with only a single OID required to represent the entire row.

15.1. Aggregate Row Objects

Within a single row in a table, the OIDs of each of the columns in the row contain a substantial amount of redundant information since each of these OIDs contain the same table prefix and instance information. The only variation among the OIDs for a group of objects in the same row of a table is the single subidentifier in the OID which represents the unique column identifier for that object.

The ability to represent a row as a single aggregate object in an SNMP PDU would provide a more efficient representation of the row as this aggregate row object could be represented with a single OID combined with a sequence of values for the objects in this row. The aggregate row object's OID would specify the table OID prefix and the instance information for that row. The value of this row object would actually be a sequence of individual values for a group of columns in this row.

15.2. Row Operations Using Aggregate Row Objects

An additional benefit to defining an aggregate row object beyond the benefit of OID suppression is the ability to perform more efficient atomic row operations. Instances of objects in the same conceptual

row in a table are often treated by manager and agent applications as a single atomic unit for operational purposes. Until now, though, the ability to perform row-based operations has been limited to the scope of MIB implementation rather than protocol implementation. There is currently no means of representing a conceptual row in a table as a single atomic unit in an SNMP operation. The ability to represent a conceptual row as a single aggregate row object would allow conceptual table rows to be represented in SNMP operations in the same manner as they are often treated logically by manager and agent applications and would allow row-based operations to be performed in a more efficient manner.

15.3. Defining Aggregate Row Objects

In order to identify aggregate row objects uniquely and unambiguously, it is necessary to define a new naming convention for aggregate row objects that distinguishes these objects from the existing conceptual row OIDs. The current convention for naming a conceptual row, as defined in RFC1902, is to append a subidentifier of '1' to the table name. For example, the conceptual row 'ifEntry' is defined by the name 'ifTable.1'.

It is proposed that aggregate row objects be defined by appending the subidentifier '2' to the table name. Using this naming convention, 'ifTable.2' would be the OID to reference aggregate row objects for the conceptual ifTable. In order to reference a specific instance of an aggregate row object, the instance information for that row would be appended to the aggregate row object's OID. Under this naming convention, the instance of the aggregate row object that represents row

In general, the definition of a variable binding for an aggregate row object would have the following format for a table with N columns:

<table>.2.<instance> = (val1, val2, ..., valN)

The value of the aggregate object would actually be a sequence of values for the columns in this row of the table.

15.4. Implicit versus Explicit column identification

Aggregate row objects can be defined with either implicit or explicit

column identification in the value sequence. Implicit column identification relies on positional context in order to map a sequence of values to their corresponding column in a table. Explicit column identification requires an explicit column identifier to be specified for each value in the aggregate object. It is foreseeable that both of these approaches will be useful under different circumstances.

Implicit column identification would require that a value be specified for each object in the row. The first value in the sequence would correspond to the first column in the table. The second value in the sequence would correspond to the second column, and so on. This approach requires less encoding for each value as it would not require that a column identifier be explicitly specified for each corresponding value. However, in the case where instances of some of the columns were missing, NULL placeholders would be required in order to maintain a one-to-one mapping between the sequence of values and the table columns.

Explicit column identification would require that a column identifier be explicitly specified for each columnar value in the aggregate row objects value sequence. This approach would require additional encoding for the column identifiers but could be beneficial in the case of tables with a large number of missing columns and could also be useful for performing operations on a subset of the columns in a table.

15.5. Encoding Aggregate Row Objects

The ASN.1 variable binding notation requires some changes to accommodate aggregate objects. It is expected that the new aggregate row object will only be valid for use in a new set of SNMP PDUs that are to be defined to address the need for simpler, more efficient row operations in a separate document. The following is a proposal for modifying the variable binding notation for these new, yet-to-be-defined PDUS to allow the varbinds in these PDUs to include both aggregate and non-aggregate objects. In the following proposal, there are no changes to the VarBindList or ObjectSyntax definitions.

```
 VarBind ::=
     SEQUENCE {
         name
             ObjectName,
         CHOICE {
```

```
            value
                ObjectSyntaxNonAggregate,
            value
                ObjectSyntaxAggregate
        }
    }

ObjectSyntaxNonAggregate ::=
    CHOICE {
        ObjectSyntax,           -- as in [RFC2578]
        ObjectSyntaxExtension,  -- new scalar types
        ObjectSyntaxNullType    -- NULL and exceptions
    }

ObjectSyntaxExtension ::=
    CHOICE {
        -- Nothing for now
        -- Eventually there will be new 64 bit types, these
        -- are being defined elsewhere
    }

--  ObjectSyntaxNullType separated for cleanliness.  There is
--  no protocol requirement for this.

ObjectSyntaxNullType ::=
    CHOICE {
        unSpecified         -- in retrieval requests
                NULL,       -- exceptions in responses
        noSuchObject[0]
                IMPLICIT NULL,
        noSuchInstance[1]
                IMPLICIT NULL,
        endOfMibView[2]
                IMPLICIT NULL
        nonInstantiatedRowObject[3]   -- plugs holes in rows
                IMPLICIT NULL
    }

ObjectSyntaxAggregate ::=
    CHOICE {
        ImplicitAggregate,
        ExplicitAggregate
     }

ImplicitAggregate ::=
```

```
        [Application 12] IMPLICIT SEQUENCE OF
             ObjectSyntaxNonAggregate

    ExplicitAggregate ::=
        [Application 13] IMPLICIT SEQUENCE OF
            EAFragment

    EAFragment ::=
        SEQUENCE {
            EAFragmentNamePart
                INTEGER (0..4294967295),
            EAFragmentValuePart
                ObjectSyntaxNonAggregate
        }
```

Note that, if no aggregate types are used and no nonInstantiatedRowObject is used, that this notation produces results equivalent to that of [RFC1905]. That is to say, if no row operations are used, then the encoding on the wire is unchanged by this notation.

16. References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC2119, March 1997.

17. Acknowledgements

The authors wish to thank Jeff Case for helpful comments as well as, in particular, David Perkins and Juergen Schoenwaelder and the NMRG of the IRTF for their previous work in this area.

18. Authors' Addresses

Sandra McLeod
SNMP Research International
3001 Kimberlin Heights Road
Knoxville, TN 37920
USA
EMail: mcleod@snmp.com

David Partain

Ericsson Radio Systems AB
P.O. Box 1248
SE-581 12 Linkoping
Sweden
EMail: David.Partain@ericsson.com

Matt White
Ericsson IP Infrastructure
7301 Calhoun Place
Rockville, MD 20855
EMail: Matt.White@ericsson.com

Table of Contents