SNMP traps (simple network management protocol)

Nasser M. Abbasi

Nov 25, 2000 Compiled on January 31, 2024 at 4:05am

Contents

This is a note on SNMP traps. To give an example of how to parse an SNMP v1 trap.

SNMP messages are of two types. synchronous and asynchronous. synchronous messages are a request-response type of protocol. The SNMP manager sends a request to the agent, and the agents sends back a response to port 161. asynchronous messages, are traps, or alerts, send by the agent to the manager at port 162, and they designate events that happened on the agent side.

The trap PDU, is defined in RFC 1157 as:

```
Trap-PDU ::=
         [4]
              IMPLICIT SEQUENCE {
                 enterprise          -- type of object generating
                                     -- trap, see sysObjectID in [5]
                     OBJECT IDENTIFIER,

                 agent-addr          -- address of object generating
                     NetworkAddress, -- trap

                 generic-trap        -- generic trap type
                     INTEGER {
                         coldStart(0),
                         warmStart(1),
                         linkDown(2),
```

```
                        linkUp(3),
                        authenticationFailure(4),
                        egpNeighborLoss(5),
                        enterpriseSpecific(6)
                    },

                specific-trap     -- specific code, present even
                    INTEGER,      -- if generic-trap is not
                                  -- enterpriseSpecific

                time-stamp        -- time elapsed between the last
                  TimeTicks,      -- (re)initialization of the network
                                  -- entity and the generation of the
                                     trap

                variable-bindings   -- "interesting" information
                     VarBindList
            }
```

Based on the above, the SNMP message for a trap is

```
+--------+---------+--------+----+---------+--------+---------------+
|version |community|PDU type|OID |Agent IP |trap ID |trap specific ID|..
---------+---------+--------+----+---------+--------+---------------+

+-----------+-----------....----+
|Time stamp |VarBindList ....   |
+-----------+------------------+
```

A trap, has a generic trap ID, and specific-trap ID.

The generic traps are:

1. The coldStart Trap
2. The warmStart Trap
3. The linkDown Trap
4. The linkUp Trap
5. The authenticationFailure Trap
6. The egpNeighborLoss Trap
7. The enterpriseSpecific Trap

If the generic trap id is the enterpriseSpecific, then the field 'trap specific ID' is used to find the exact trap ID. This field is defined and set by each company to use for their specific needs.

1 Processing on SNMP messages

from RFC 1157:

Similarly, the top-level actions of a protocol entity which receives a message are as follows:

1. It performs a rudimentary parse of the incoming datagram to build an ASN.1 object corresponding to an ASN.1 Message object. If the parse fails, it discards the datagram and performs no further actions.

2. It then verifies the version number of the SNMP message. If there is a mismatch, it discards the datagram and performs no further actions.

3. The protocol entity then passes the community name and user data found in the ASN.1 Message object, along with the datagram's source and destination transport addresses to the service which implements the desired authentication scheme. This entity returns another ASN.1 object, or signals an authentication failure. In the latter case, the protocol entity notes this failure, (possibly) generates a trap, and discards the datagram and performs no further actions.

4. The protocol entity then performs a rudimentary parse on the ASN.1 object returned from the authentication service to build an ASN.1 object corresponding to an ASN.1 PDUs object. If the parse fails, it discards the datagram and performs no further actions. Otherwise, using the named SNMP community, the appropriate profile is selected, and the PDU is processed accordingly. If, as a result of this processing, a message is returned then the source transport address that the response message is sent from shall be identical to the destination transport address that the original request message was sent to.

Related RFC's (from RFC 1212):

RFC 1065, which defined the Structure of Management Information (SMI).

RFC 1066, which defined the Management Information Base (MIB).

Both of these documents were designed so as to be compatible with both the SNMP and the OSI network management framework.

2 Parsing an SNMP v1 UDP pkt

Using the little Java program shown below, capture a UDP pkt to a file.

The following is the full ethernet frame containing the IP pkt, containing the UDP pkt containing the SNMP message:

```
UDP:  ----- UDP Header -----
UDP:
UDP:  Source port = 4801
UDP:  Destination port = 162
UDP:  Length = 118
UDP:  Checksum = A835
UDP:

                ethernet
    +---------------------------------+
    |                                 |
 0: 0800 2080 e76c 0090 f2c8 a400 0800 4500    .. ..l........E.
16: 008a 0021 0000 fb11 6597 ac1c 8136 ab47    ...!....e....6.G
32: 8110 12c1 00a2 0076 a835 306c 0201 0004    .......v.50l....
48: 0753 4e4d 5076 3263 a45e 0609 2b06 0104    .SNMPv2c.^..+...
64: 0109 092b 0240 0400 0000 0002 0106 0201    ...+.@..........
80: 0143 0400 eee4 1c30 3f30 1306 0e2b 0601    .C.....0?0...+..
96: 0401 0909 2b01 0106 0103 3202 0101 3013    ....+.....2...0.
112: 060e 2b06 0104 0109 092b 0101 0601 0432    ..+......+.....2
128: 0201 0330 1306 0e2b 0601 0401 0909 2b01    ...0...+......+.
144: 0106 0105 3202 0104                        ....2...
```

The total ethernet message is 152 bytes.

The Ethernet header is 14 bytes. (6+6+2).

the IP header is 20 bytes (no options).

The UDP header is 8 bytes.

This leaves the SNMP message = 152- (14+20+8) = 110 bytes.

removing ethernet, leaves:

```
                                    +-..IP
                                    |
                                    4500                  E.
 16: 008a 0021 0000 fb11 6597 ac1c 8136 ab47    ...!....e....6.G
```

```
 32: 8110 12c1 00a2 0076 a835 306c 0201 0004    .......v.50l....
         |                 |
     ..---+-----------------+
       IP          UDP

 48: 0753 4e4d 5076 3263 a45e 0609 2b06 0104    .SNMPv2c.^..+...
 64: 0109 092b 0240 0400 0000 0002 0106 0201    ...+.@..........
 80: 0143 0400 eee4 1c30 3f30 1306 0e2b 0601    .C.....0?0...+..
 96: 0401 0909 2b01 0106 0103 3202 0101 3013    ....+.....2...0.
112: 060e 2b06 0104 0109 092b 0101 0601 0432    ..+......+.....2
128: 0201 0330 1306 0e2b 0601 0401 0909 2b01    ...0...+......+.
144: 0106 0105 3202 0104                        ....2...
```

removing IP and UDP, leaves SNMP pkt:

```
 32:                         306c 0201 0004    .......v.50l....
 48: 0753 4e4d 5076 3263 a45e 0609 2b06 0104    .SNMPv2c.^..+...
 64: 0109 092b 0240 0400 0000 0002 0106 0201    ...+.@..........
 80: 0143 0400 eee4 1c30 3f30 1306 0e2b 0601    .C.....0?0...+..
 96: 0401 0909 2b01 0106 0103 3202 0101 3013    ....+.....2...0.
112: 060e 2b06 0104 0109 092b 0101 0601 0432    ..+......+.....2
128: 0201 0330 1306 0e2b 0601 0401 0909 2b01    ...0...+......+.
144: 0106 0105 3202 0104                        ....2...
```

How to parse this message?

SNMP message data represent ASN.1 notation encoded using BER encoding.

First lets look at the ASN.1 message structure expected for an SNMP trap, and we look how it is decoded.

The SNMP Trap ASN.1 structure is:

```
       SNMP Trap Message ::=
               SEQUENCE
               {
                       version    INTEGER,
                       community  OCTET STRING,
                       IMPLICIT SEQUENCE
                       {
                               enterprise  OBJECT IDENTIFIER,
                               agent-addr  Network Address,
```

```
                             generic-trap INTEGER
                             {
                                         coldStart(0),
                                         warmStart(1),
                                         linkDown(2),
                                         linkUp(3),
                                         authenticationFailure(4)
                                         egpNeighborLoss(5),
                                         enterpriseSpecific(6)
                              },
                              specific-trap  INTEGER,
                              time-stamp     TimeTicks,
                              variable-bindings  SEQUENCE OF  VarBind
                        }
              }

          VarBind ::=
                  SEQUENCE {
                      name   ObjectName,
                      value  ObjectSyntax
                  }
```

```
<------------SEQUENCE--------------------------------------------->
                                        <-------..IMPLICT SEQUENCE
+----+----+----+---+-------+----+---+---------+----+---+---+--
|type|len |type|len|version|type|len|community|type|len|OID| etc...
+----+----+----+---+-------+----+---+---------+----+---+---+--
```

In a more simplified form, the above TRAP SNMP message can be shown as

```
+-------+---------+--------+--------+--------------+---------------+
|version|community|AgentOID|Agent IP|generic trapID|specific trap ID|..
+--------+--------+----------+---------+--------------+-----------+


 +-----------+--------+----------+----------+-----------+-----
 |Time stamp |Var name|var value | var name | var value | ....
 +-----------+--------+----------+----------+-----------+----
```

data that represents the above structure is encoded in BER as `<tag,length,value>` entities. So there is a tag ID for SEQUENCE, tag ID for INTEGER, tag ID for OBJECT IDENTIFIER, etc.., The length tells how many bytes are used for the value itself.

i.e. The tag is used to tell the type of value that follows, and the length used to tell how many bytes occupied by the value.

The following are some of the standard tag ID's (from CCITT X.208) :

ASN.1 Type	Tag (decimal)	Tag (Hex)	bits
INTEGER	2	02	0000 0010
BIT STRING	3	03	0000 0011
OCTET STRING	4	04	0000 0100
NULL	5	05	0000 0101
OID	6	06	0000 0110
Sequence of Sequence	16	10	0001 0000
SET of SET	17	11	0001 0001
PrintableString	19	13	0001 0011
IA5String	22	16	0001 0110
UTCTime	23	17	0001 0111

bit 8 and 7 have special meaning in the tag. if noth are 0, the class of the ASN.1 type is called universal, if bit 7 is 1, and bit 8 is 0, the class is application, if bit 8 is 1, and bit 7 is 0, the class of type is context specific, and if both bit 8 and 7 are 1, then the class is private.

This allows one to customize types if they need to:

bit 8	bit 7	class of type
0	0	universal
0	1	application
1	0	context-specific
1	1	private

If bit 6 is 0, this means the encoding is primitive, i.e. the tag ID is specified in bits 5-1.

It is possible to use more than one byte to specify the tag ID, in this case, bits 1-5 all will have the value 1, and the following bytes will give the tag value, where last byte bit 8 is 0 to indicate last byte is this long tag encoding.

If bit 6 of the first type byte is 1, this means the type is constructed.

How about the length? The length is tricky, since many bytes can be used for the length itself, to tell how many bytes are used for the value. If bit 8 of the length byte is 0, this means the length value is all contained in this byte. i.e. bit 7-1 give the length. If bit 8 is 1, then bits 7-1 gives the number of bytes (not including the first byte) that represent

the value of the length based 265, most significant digit first. (btw, SnmpOnJava does not support number of length bytes more than 4, i.e. indefinite length).

so, to parse an SNMP message we do (in very simplified way):

read SNMP message out of UDP pkt.

set index to first byte in message

```
LOOP
   get byte (and advance index to next byte).
   determine TAG ID of byte read. (based on table and rules  above).
   get byte (the length byte) and advance to next byte.
   if length byte indicates simple length (i.e. bit 8 is 0) Then
      value = get as many bytes following this byte as the length
              above indicated.
   else
      construct length amount by reading number of bytes following
      the first length byte.
      value = get as many bytes following last length byte
              as above length indicates.
   end if

   if no more bytes left in message then
      exit LOOP
   end if

end LOOP.
```

Lets now see how this applies to our SNMP message above, here it is again: (110 bytes).

```
 32:                          306c 0201 0004    .......v.50l....
 48: 0753 4e4d 5076 3263 a45e 0609 2b06 0104    .SNMPv2c.^..+...
 64: 0109 092b 0240 0400 0000 0002 0106 0201    ...+.@..........
 80: 0143 0400 eee4 1c30 3f30 1306 0e2b 0601    .C.....0?0...+..
 96: 0401 0909 2b01 0106 0103 3202 0101 3013    ....+.....2...0.
112: 060e 2b06 0104 0109 092b 0101 0601 0432    ..+......+.....2
128: 0201 0330 1306 0e2b 0601 0401 0909 2b01    ...0...+......+.
144: 0106 0105 3202 0104                        ....2..
```

The first type is 0x30, or '0011 0000', we see that bit 6 is '1', which means encoding is constructed, and the TAG ID is bites 5-1, which is '1 0000' which is 16, i.e. from above

table, it is SEQUENCE of SEQUENCE type. (i.e. the type of the whole SNMP msg).

The next length byte is 0x6c, which is '0110 1010', we see that bit 8 is '0', this means the length of the SEQUENCE of SEQUENCE value is indicated in bits 7-1 only, which is 108. i.e. the SNMP message has length 108 (not counting the tag+length 2 bytes we just processed). This matches what we have already from UDP header telling us the message we 110 bytes.

Next we see 0x02, which is '0000 0010', which from the table shows the type to be INTEGER, the length is the next byte 0x01, meaning that one byte following it contain the value. The next byte is 0x00, meaning this type has value 0. Looking at the ASN.1 structure, we see this is the SNMP version number, i.e. we have received an SNMP version 0 message.

(this is another problem, we should be getting version=1, to mean SNMPv2c, I need to play more with router to see if I can get it to send v2 traps).

Next we 0x04, which is '0000 0100' which means this is an OCTECT STRING from looking at the above table.

The next byte is 0x07 which is '0000 0111' and since bit 8 is '0', this means the length of the data for this type is contained in this byte, and the length is 7 bytes, then reading the following 7 bytes as octet string gives:

```
      48:     4e4d 5076 3263                          SNMPv2c
```

and from the ASN.1 structure we see this is the community string.

After this we see byte 0xa4, i.e. '1010 0100', we notice that bit 8-7 is '10', ie. this is context specific type from table above. also notice that bit 6 is '1' meaning encoding is constructed. so bits 5-1 gives the tag id, i.e '0 0100' which is 4, or OCTET STRING (constructed).

The length byte is 0x5e, i.e. '0101 1100' , and since bit 8 is '0', the length is all contained in this byte, which is 94. meaning this type value (the OCTET STRING) contains 94 bytes.

Looking at the ASN.1 we see that this is the IMPLICIT SEQUENCE part of the SNMP trap message. (i.e. the body of the trap message, everything after the version + agentOID.

The first byte in the IMPLICIT SEQUENCE is 0x06 which is '0000 0110', which is from the table means it is an OBJECT IDENTIFIER (OID). next, the length byte is 0x09, which is '0000 1001', and since bit 8 is '0', the length is all in this byte, which is 9 bytes. the next 9 bytes are:

```
48:                               2b06 0104                +...
64: 0109 092b 02                                    ...+.
```

Notice that if the first byte is 0x2b, it is automatically replaced by '1' followed by '3'. (i.e. iso(1) -> org(3)) so given the above, we have the OID as:

```
    from 0x2b
      ------
     /      \
      '1' '3'  '6' '1'  '4'  '1'  '9'  '9'  '43' '2'
```

ie. the agent OID is 1.3.6.1.4.1.9.9.43.2, and looking at the oid files we see that

CISCO-CONFIG-MAN-MIB.oid:

`"ciscoConfigManMIBNotificationPrefix" "1.3.6.1.4.1.9.9.43.2"`

Following the agent OID, the next byte is 0x40, which is '0000 0100', which from the table is tag ID for OCTET STRING.

The length byte is 0x04, which is '0000 0100', and since bit 8 is '0', the length is 4. and from the ASN.1 structure we see this is the Agent address, so the next 4 bytes are the value, which is 0x00 0x00 0x00 0x00, so the IP address is '0.0.0.0'. (this shows the problem, the agent is not setting its IP address correctly in the trap SNMP message).

Following this, the type byte is 0x02, which is INTEGER, the length byte following is 0x01, meaning one byte value, which is the next byte 0x06. from ASN.1 we see this to be as the generic trap ID. and since the trap ID is 6, this means the specific trap ID gives more information as to what trap this is.

The type byte after this is 0x02, which is an INTEGER, the length is 0x01, and the value then is the byte following this which is 0x01, which means that the specific trap ID is '1'. (in this case the OID of trap is build by post appending '.0.' followed by this ID which is '1' to the agent OID, making the trap OID as

```
            '1.3.6.1.4.1.9.9.43.2' + '.0.1' =

            '1.3.6.1.4.1.9.9.43.2.0.1'
```

Notice that if this were a generic trap, then the trap OID is constructed as this :

```
            '1.3.6.1.6.3.1.1.5' + generic + 1.
```

where '1.3.6.1.6.3.1.1.5' is the OID for "SnmpTraps". (from SNMPv2-MIB.oid).

Following the above, the next byte is 0x43 which is '0100 0011', and since bit 7 is '1' this means this is 'application' specific type. the value of the tag is 3, which is BIT STRING. the length byte is 0x04, i.e. '0000 0100', and since bit 8 is '0', the length is 4. so, the next 4 bytes contain the value of this type. From the ASN.1 structure, we see this is the TIME STAMP value.

The next 4 bytes are:

```
                      value
                     --------
            tag len  /        \
    80:  01 43  04   00 eee4 1c
```

the time stamp is amount in time in 10's of milliseconds since the device was back on-line and the trap was generated.

Following the TIME STAMP is the var binds. this is a sequence of <name,value> pairs, each encoded as <tag,length,value> as well.

3 Program to capture UDP pkt to a file

Compile and run this program as. The listing is below. The code is at PortListen.java.

```
javac PortListen.java
java  PortListen 162
```

```
//by Nasser M. Abbasi. Written sometime in the year 2000.

import java.net.*;
import java.io.FileOutputStream;

class PortListen {
    static private final boolean PKT= false; // make true to save pkt to a file
    static private final boolean DEBUG = false;

    static private DatagramSocket _trapSocket = null;

    static private final int _BUFFER_SIZE = 8 * 1024;
    static private long _numberOfUDP_pkts=0;
```

```
    public static void main(String args[]) throws Exception
    {
        if ( args.length != 1 ) {
            System.err.println("usage: portlisten <port>");
            return;
        }

        int port=0;

        try {
            port= new Integer(args[0]).intValue();
        }
        catch ( Exception e ) {
            System.err.println("invalid port number supplied \""+ args[0] + "\"");
        }

        init(port);
        run();
    }

    static private void init(int portNumber) throws Exception
    {

        try {
            _trapSocket    = new DatagramSocket(portNumber);
        }
        catch ( SocketException e ) {
            System.err.println("Socket thread Exception creating socket");
            throw e;
        }
    }

    static private void savePkt(int howMany, byte buf[])
    {
        try {
            java.io.File file = new java.io.File("pkt.log");
            System.err.println("pkt will be written to " + file.getAbsolutePath());

            FileOutputStream f = new FileOutputStream("pkt.log");
            for ( int j = 0; j < howMany; j++ )
                f.write(buf[j]);
```

```
            f.close();
        }
        catch ( Exception exp ) {
            System.err.println("Exeption in savePkts");
            exp.printStackTrace();
        }
    }


    static private void run() throws ThreadDeath, Exception
    {
        byte[] buf = null;
        buf = new byte[_BUFFER_SIZE];

        MAIN_LOOP:
        while ( true ) {
            try {

                DatagramPacket pkt = new DatagramPacket(buf, buf.length);

                // blocking operation
                _trapSocket.receive(pkt);

                System.err.println("got UDP pkt size="+ "length=" +  pkt.getLength() );

                if ( _numberOfUDP_pkts == Long.MAX_VALUE )
                    _numberOfUDP_pkts = 0;

                _numberOfUDP_pkts++;

                System.err.println("UDP pkts read so far:" + _numberOfUDP_pkts);

                if ( PKT ) {
                    savePkt(pkt.getLength(), buf);
                }
            }
            catch ( Exception e ) {
                System.err.println("Exeption in run");
                e.printStackTrace();
                throw e;
```

```
            }
        }//while(...)
    }
}
```