Network Working Group
Request for Comments: 1228

G. Carpenter
B. Wijnen
T.J. Watson Research Center, IBM Corp.
May 1991

# SNMP-DPI
# Simple Network Management Protocol
# Distributed Program Interface

## Status of this Memo

This RFC describes a protocol that International Business Machines Corporation (IBM) has been implementing in most of its SNMP agents to allow dynamic extension of supported MIBs. This is an Experimental Protocol for the Internet community. Discussion and suggestions for improvement are requested. Please refer to the current edition of the "IAB Official Protocol Standards" for the standardization state and status of this protocol. Distribution of this memo is unlimited.

## ABSTRACT

The Simple Network Management Protocol (SNMP) [1] Distributed Program Interface (DPI) is an extension to SNMP agents that permits end-users to dynamically add, delete or replace management variables in the local Management Information Base without requiring recompilation of the SNMP agent. This is achieved by writing a so-called sub-agent that communicates with the agent via the SNMP-DPI.

For the author of a sub-agent, the SNMP-DPI eliminates the need to know the details of ASN.1 [2] or SNMP PDU (Protocol Data Unit) encoding/decoding [1, 3].

This protocol has been in use within IBM since 1989 and is included in the SNMP agents for VM, MVS and OS/2.

Potentially useful sample sub-agent code and implementation examples are available for anonymous FTP from the University of Toronto.

## MOTIVATION

The Simple Network Management Protocol [1] defines a protocol that permits operations on a collection of variables. This set of variables is called the Management Information Base (MIB) and a core set of variables has previously been defined [4, 5]; however, the design of the MIB makes provision for extension of this core set. Thus, an enterprise or individual can define variables of their own which represent information of use to them. An example of a

potentially interesting variable which is not in the core MIB would be CPU utilization (percent busy). Unfortunately, conventional SNMP agent implementations provide no means for an end-user to make available new variables.

The SNMP DPI addresses this issue by providing a light-weight mechanism by which a process can register the existence of a MIB variable with the SNMP agent. When requests for the variable are received by the SNMP agent, it will pass the query on to the process acting as a sub-agent. This sub-agent then returns an appropriate answer to the SNMP agent. The SNMP agent eventually packages an SNMP response packet and sends the answer back to the remote network management station that initiated the request.

None of the remote network management stations have any knowledge that the SNMP agent calls on other processes to obtain an answer. As far as they can tell, there is only one network management application running on the host.

## THEORY OF OPERATION

### CONNECTION ESTABLISHMENT

Communication between the SNMP Agent and its clients (sub-agents) takes place over a stream connection. This is typically a TCP connection, but other stream-oriented transport mechanisms can be used. As an example, the VM SNMP agent allows DPI connections over IUCV (Inter-User Communications Vehicle) [6, 7]. Other than the connection establishment procedure, the protocol used is identical in these environments.

### REGISTRATION

Regardless of the connection-oriented transport mechanism used, after establishing a connection to the SNMP agent, the sub-agent registers the set of variables it supports. Finally, when all the variable classes have been registered, the sub-agent then waits for requests from the SNMP agent or generates traps as required.

### DPI ARCHITECTURE

There are three requests that can be initiated by the SNMP agent: GET, GET-NEXT and SET. These correspond directly to the three SNMP requests that a network management station can make. The sub-agent responds to a request with a RESPONSE packet.

There are currently two requests that can be initiated by a sub-agent: REGISTER and TRAP.

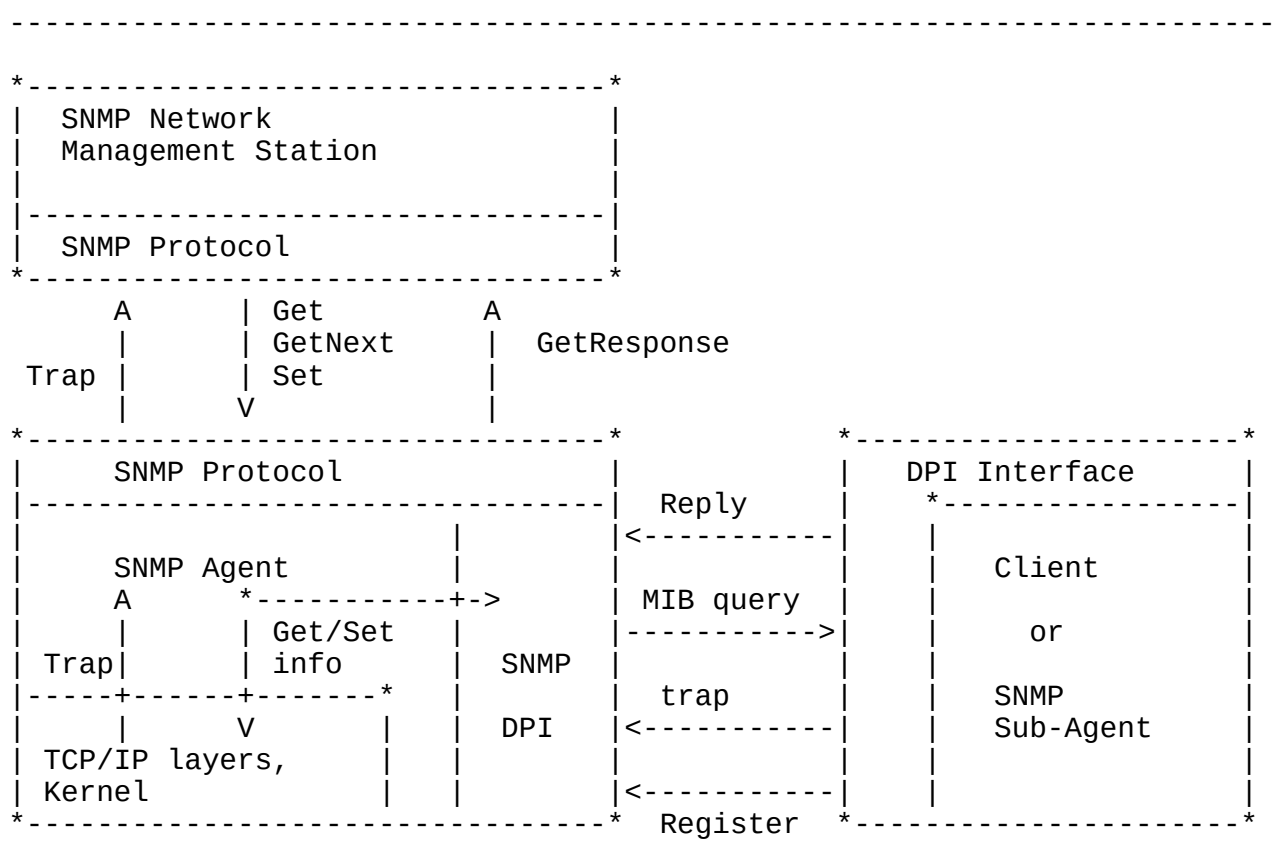


Figure 1. SNMP DPI overview

Remarks for Figure 1:

- The SNMP agent communicates with the SNMP manager via the standard SNMP protocol.
- The SNMP agent communicates with the TCP/IP layers and kernel (operating system) in an implementation-dependent manner. It potentially implements the standard MIB view in this way.
- An SNMP sub-agent, running as a separate process (potentially even on another machine), can register objects with the SNMP agent.
- The SNMP agent will decode SNMP Packets. If such a packet contains a Get/GetNext or Set request for an object registered by a sub-agent, it will send the request to the sub-agent via the corresponding query packet.
- The SNMP sub-agent sends responses back via a RESPONSE packet.
- The SNMP agent then encodes the reply into an SNMP packet and sends it back to the requesting SNMP manager.
- If the sub-agent wants to report an important state change, it

sends a TRAP packet to the SNMP agent, which will encode it into an SNMP trap packet and send it to the manager(s).

## SNMP DPI PROTOCOL

This section describes the actual protocol used between the SNMP agent and sub-agents. This information has not previously been published.

## CONNECTION ESTABLISHMENT

In a TCP/IP environment, the SNMP agent listens on an arbitrary TCP port for a connection request from a sub-agent. It is important to realize that a well-known port is not used: every invocation of the SNMP agent will potentially result in a different TCP port being used.

A sub-agent needs to determine this port number to establish a connection. The sub-agent learns the port number from the agent by sending it one conventional SNMP get-request PDU. The port number is maintained by the SNMP agent as the object whose identifier is 1.3.6.1.4.1.2.2.1.1.0; this variable is registered under the IBM enterprise-specific tree. The SNMP agent replies with a conventional SNMP response PDU that contains the port number to be used. This response is examined by the sub-agent and the port number is extracted. The sub-agent then establishes the connection to the specified port.

On the surface, this procedure appears to mean that the sub-agent must be able to create and parse SNMP packets, but this is not the case. The DPI Application Program Interface (API) has a library routine, query_DPI_port(), which can be used to generate and parse the required SNMP packets. This routine is very small (under 100 lines of C), so it does not greatly increase the size of any sub-agent).

For completeness, byte-by-byte descriptions of the packets generated by the SNMP DPI API routine query_DPI_port() are provided below. This is probably of little interest to most readers and reading the source to query_DPI_port() provides much of the same information.

## SNMP PDU TO GET THE AGENT'S DPI PORT

As noted, before a TCP connection to the SNMP agent can be made, the sub-agent must learn which TCP port that the agent is listening on. To do so, it can issue an SNMP GET for an IBM enterprise-specific variable 1.3.6.1.4.1.2.2.1.1.0.

NOTE: the object instance of ".0" is included for clarity in this document.

The SNMP PDU can be constructed as shown below. This PDU must be sent to UDP port 161 on the host where the agent runs (probably the same host where the sub-agent runs).

Table 1. SNMP PDU for GET DPI_port. This is the layout of an SNMP PDU for GET DPI_port

| OFFSET | VALUE | FIELD |
|---|---|---|
| 0 | 0x30 | ASN.1 header |
| 1 | 34 + len | pdu_length, see formula below |
| 2 | 0x02 0x01 0x00 0x04 | version (integer, length=1, value=0), community name (string) |
| 6 | len | length of community name |
| 7 | community name | |
| 7 + len | 0xa0 0x1b | SNMP GET request: request_type=0xa0, length=0x1b |
| 7 + len + 2 | 0x02 0x01 0x01 | SNMP request ID: integer, length=1, ID=1 |
| 7 + len + 5 | 0x02 0x01 0x00 | SNMP error status: integer, length=1, error=0 |
| 7 + len + 8 | 0x02 0x01 0x00 | SNMP index: integer, length=1, index=0 |
| 7 + len + 11 | 0x30 0x10 | Varbind list, length=0x10 |
| 7 + len + 13 | 0x30 0x0e | Varbind, length=0x0e |
| 7 + len + 15 | 0x06 0x0a | Object ID, length=0x0a |
| 7 + len + 17 | 0x2b 0x06 0x01 0x04 0x01 0x02 0x02 0x01 0x01 0x00 | Object instance: 1.3.6.1.4.1.2.2.1.1.0 |
| 7 + len + 27 | 0x05 0x00 | null value, length=0 |

The formula to calculate the length field "pdu_length" is as follows:

pdu_length = length of version field and string tag (4 bytes)

```
+   length of community length field (1 byte)
+   length of community name (depends...)
+   length of SNMP GET request (29 bytes)

=   34 + length of community name
```

SNMP PDU CONTAINING THE RESPONSE TO THE GET

Assuming that no errors occured, then the port is returned in the last 2 octets of the received packet. The format of the packet is shown below:

Table 2. SNMP RESPONSE PDU for GET of Agent's DPI port. This is the layout of an SNMP RESPONSE PDU for GET DPI_port

| OFFSET | VALUE | FIELD |
|---|---|---|
| 0 | 0x30 | ASN.1 header |
| 1 | 36 + len | length, see formula below |
| 2 | 0x02 0x01 0x00 0x04 | version (integer, length=1, value=0), community name (string) |
| 6 | len | length of community name |
| 7 | community name | |
| 7 + len | 0xa2 0x1d | SNMP RESPONSE: request_type=0xa2, length=0x1d |
| 7 + len + 2 | 0x02 0x01 0x01 | SNMP request ID: integer, length=1, ID=1 |
| 7 + len + 5 | 0x02 0x01 0x00 | SNMP error status: integer, length=1, error=0 |
| 7 + len + 8 | 0x02 0x01 0x00 | SNMP index: integer, length=1, index=0 |
| 7 + len + 11 | 0x30 0x12 | Varbind list, length=0x12 |
| 7 + len + 13 | 0x30 0x10 | Varbind, length=0x10 |
| 7 + len + 15 | 0x06 0x0a | Object ID, length=0x0a |

Table 2. SNMP RESPONSE PDU for GET of Agent's DPI port. This is the layout of an SNMP RESPONSE PDU for GET DPI_port

| OFFSET | VALUE | FIELD |
|---|---|---|
| 7 + len + 17 | 0x2b 0x06 0x01<br>0x04 0x01 0x02<br>0x02 0x01 0x01<br>0x00 | Object instance:<br>1.3.6.1.4.1.2.2.1.1.0 |
| 7 + len + 27 | 0x02 0x02 | integer, length=2 |
| 7 + len + 29 | msb lsb | port number (msb, lsb) |

The formula to calculate the length field "pdu_length" is as follows:

```
pdu_length =   length of version field and string tag (4 bytes)
           +   length of community length field (1 byte)
           +   length of community name (depends...)
           +   length of SNMP RESPONSE (31 bytes)

           =   36 + length of community name
```

SNMP DPI PACKET FORMATS

Each request to or response from the agent is constructed as a "packet" and is written to the stream.

Each packet is prefaced with the length of the data remaining in the packet. The length is stored in network byte order (most significant byte first, least significant last). The receiving side will read the packet by doing something similar to:

```
unsigned char len_bfr[2];
char *bfr;
int len;

read(fd,len_bfr,2);
len = len_bfr[0] * 256 + len_bfr[1];
bfr = malloc(len);
read(fd,bfr,len);
```

NOTE: the above example makes no provisions for error handling or a read returning less than the requested amount of data. This is not a suggested coding style.

The first part of every packet identifies the application protocol being used, as well as some version information. The protocol major version is intended to indicate in broad terms what version of the protocol is used. The protocol minor version is intended to identify major incompatible versions of the protocol. The protocol release is intended to indicate incremental modifications to the protocol. The constants that are valid for these fields are defined in Table 10 on page 18.

The next (common) field in all packets is the packet type. This field indicates what kind of packet we're dealing with (SNMP DPI GET, GET-NEXT, SET, TRAP, RESPONSE or REGISTER). The permitted values for this field are defined in Table 11 on page 18.

Table 3. SNMP DPI packet header. This header is present in all packets.

| OFFSET | FIELD |
|---|---|
| 0 | packet length to follow (MSB) |
| 1 | packet length to follow (LSB) |
| 2 | protocol major version |
| 3 | protocol minor version |
| 4 | protocol release |
| 5 | packet type |

>From this point onwards, the contents of the packet are defined by the protocol being used. The remainder of this section describes:

- the structure of packets for the SNMP DPI protocol, version 1.0.
- The constants as defined with this version of the protocol.

REGISTER

In order to register a branch in the MIB tree, an SNMP sub-agent sends an SNMP DPI REGISTER packet to the agent.

Such a packet contains the standard SNMP DPI header plus REGISTER-specific data, which basically is a null terminated string representing the object ID in dotted ASN.1 notation (with a trailing dot!).

Table 4. SNMP DPI REGISTER packet. This is the layout of an SNMP DPI REGISTER packet

| OFFSET | FIELD |
|---|---|
| 0 | packet length to follow (MSB) |
| 1 | packet length to follow (LSB) |
| 2 | protocol major version |
| 3 | protocol minor version |
| 4 | protocol release |
| 5 | packet type = SNMP_DPI_REGISTER |
| 6 | null terminated object ID |

GET

When the SNMP agent receives a PDU containing an SNMP GET request for a variable that a sub-agent registered with the agent, it passes an SNMP DPI GET packet to the sub-agent.

Such a packet contains the standard SNMP DPI header plus GET-specific data, which is basically a null terminated string representing the object ID in dotted ASN.1 notation.

Table 5. SNMP DPI GET packet. This is the layout of an SNMP DPI GET packet

| OFFSET | FIELD |
|---|---|
| 0 | packet length to follow (MSB) |
| 1 | packet length to follow (LSB) |
| 2 | protocol major version |
| 3 | protocol minor version |
| 4 | protocol release |
| 5 | packet type = SNMP_DPI_GET |
| 6 | null terminated object ID |

GET-NEXT

When the SNMP agent receives a PDU containing an SNMP GET-NEXT request for a variable for which a sub-agent may be authoritative, it passes an SNMP DPI GET-NEXT packet to the sub-agent.

Such a packet contains the standard SNMP DPI header plus GET-NEXT-specific data. These data take the form of two null terminated strings. The first string represents the object ID in dotted ASN.1 notation; the second string represents the group ID in dotted ASN.1 notation.

Table 6. SNMP DPI GET NEXT packet. This is the layout of an SNMP DPI GET NEXT packet

| OFFSET | FIELD |
|---|---|
| 0 | packet length to follow (MSB) |
| 1 | packet length to follow (LSB) |
| 2 | protocol major version |
| 3 | protocol minor version |
| 4 | protocol release |
| 5 | packet type = SNMP_DPI_GET_NEXT |
| 6 | null terminated object ID |
| 6 + len | null terminated group ID |
| NOTE: len=strlen(object ID)+1 | |

SET

When the SNMP agent receives a PDU containing an SNMP SET request for a variable that a sub-agent registered with the agent, it passes an SNMP DPI SET packet to the sub-agent.

Such a packet contains the standard SNMP DPI header plus SET specific data, which is basically a null terminated string representing the object ID in ASN.1 notation, with the type, value length and value to be set. The permitted types for the type field are defined in Table 12 on page 19. Integer values are sent as 4-byte elements in network byte order (most significant byte first, least significant byte last).

Table 7. SNMP DPI SET packet. This is the layout of an SNMP DPI SET packet

| OFFSET | FIELD |
|---|---|
| 0 | packet length to follow (MSB) |
| 1 | packet length to follow (LSB) |
| 2 | protocol major version |
| 3 | protocol minor version |
| 4 | protocol release |
| 5 | packet type = SNMP_DPI_SET |
| 6 | null terminated object ID |
| 6 + len | SNMP Variable Type Value |
| 6 + len + 1 | Length of value (MSB) |
| 6 + len + 2 | Length of value (LSB) |
| 6 + len + 3 | Value |

NOTE: len=strlen(object ID)+1

RESPONSE

An SNMP sub-agent must respond to a GET, GET_NEXT or SET request that it has received from the agent (unless it fails or has a bug). To do so, it sends an SNMP DPI RESPONSE packet to the agent.

Such a packet contains the standard SNMP DPI header plus RESPONSE specific data, which basically is an error_code plus (if there was no error), the name/type/value tuple representing the returned object. This is described as by a string representing the object ID in ASN.1 notation, plus the type, value length and value of the object that was manipulated. The permitted types for the type field are defined in Table 12 on page 19. Integer values are sent as 4-byte elements in network byte order (most significant byte first, least significant byte last).

Table 8. SNMP DPI RESPONSE packet. This is the layout of an SNMP DPI RESPONSE packet

| OFFSET | FIELD |
|---|---|
| 0 | packet length to follow (MSB) |
| 1 | packet length to follow (LSB) |
| 2 | protocol major version |
| 3 | protocol minor version |
| 4 | protocol release |
| 5 | packet type = SNMP_DPI_RESPONSE |
| 6 | SNMP error code |
| 7 | null terminated object ID |
| 7 + len | SNMP Variable Type Value |
| 7 + len + 1 | Length of value (MSB) |
| 7 + len + 2 | Length of value (LSB) |
| 7 + len + 3 | Value |

NOTE: len=strlen(object ID)+1

TRAP

An SNMP sub-agent can request the agent to generate a TRAP by sending an SNMP DPI TRAP packet to the agent.

Such a packet contains the standard SNMP DPI header plus TRAP specific data, which is basically the generic and specific trap code, plus a name/type/value tuple. The tuple is described by a string representing the object ID in ASN.1 notation, plus the type, value length and value of the object that is being sent in the trap. The permitted types for the type field are defined in Table 12 on page 19. Integer values are sent as 4-byte elements in network byte order (most significant byte first, least significant byte last).

Table 9. SNMP DPI TRAP packet. This is the layout of an SNMP DPI TRAP packet

| OFFSET | FIELD |
|---|---|
| 0 | packet length to follow (MSB) |
| 1 | packet length to follow (LSB) |
| 2 | protocol major version |
| 3 | protocol minor version |
| 4 | protocol release |
| 5 | packet type - SNMP_DPI_TRAP |
| 6 | SNMP generic trap code |
| 7 | SNMP specific trap code |
| 8 | null terminated object ID |
| 8 + len | SNMP Variable Type Value |
| 8 + len + 1 | Length of value (MSB) |
| 8 + len + 2 | Length of value (LSB) |
| 8 + len + 3 | Value |

NOTE: len=strlen(object ID)+1

## CONSTANTS AND VALUES

This section describes the constants that have been defined for this version of the SNMP DPI Protocol.

### PROTOCOL VERSION AND RELEASE VALUES

Table 10. Protocol version and release values

| FIELD | VALUE |
|---|---|
| protocol major version | 2 (SNMP DPI protocol) |
| protocol minor version | 1 (version 1) |
| protocol release | 0 (release 0) |

Any other values are currently undefined.

### PACKET TYPE VALUES

The packet type field can have the following values:

Table 11. Valid values for the packet type field

| VALUE | PACKET TYPE |
|---|---|
| 1 | SNMP_DPI_GET |
| 2 | SNMP_DPI_GET_NEXT |
| 3 | SNMP_DPI_SET |
| 4 | SNMP_DPI_TRAP |
| 5 | SNMP_DPI_RESPONSE |
| 6 | SNMP_DPI_REGISTER |

VARIABLE TYPE VALUES

The variable type field can have the following values:

Table 12. Valid values for the Value Type field

| VALUE | VALUE TYPE |
|---|---|
| 0 | text representation |
| 129 | number (integer) |
| 2 | octet string |
| 3 | object identifier |
| 4 | empty (no value) |
| 133 | internet address |
| 134 | counter (unsigned) |
| 135 | gauge (unsigned) |
| 136 | time ticks (1/100ths seconds) |
| 9 | display string |

NOTE: Fields which represent values that are stored as a 4-byte integer are indicated by ORing their base type value with 128.

Error Code Values for SNMP Agent Detected Errors

The error code can have one of the following values:

| Table 13. Valid values for the SNMP Agent Minor Error Code field | |
|---|---|
| VALUE | SNMP AGENT ERROR CODE |
| 0 | no error |
| 1 | too big |
| 2 | no such name |
| 3 | bad value |
| 4 | read only |
| 5 | general error |

SNMP DPI APPLICATION PROGRAM INTERFACE

This section documents an API that implements the SNMP DPI. This information has been previously published [6, 8], but the information provided below is more current as of May 14, 1991.

OVERVIEW OF REQUEST PROCESSING

GET PROCESSING

A GET request is the easiest to process. When the DPI packet is parsed, the parse tree holds the object ID of the variable being requested.

If the specified object is not supported by the sub-agent, it would return an error indication of "no such name". No name/type/value information would be returned.

```
unsigned char *cp;

cp = mkDPIresponse(SNMP_NO_SUCH_NAME,0);
```

If the object is recognized, then the sub-agent creates a parse tree representing the name/type/value of the object in question (using the DPI API routine mkDPIset()), and returns no error indication. This is demonstrated below (a string is being returned).

```
        char *obj_id;

        unsigned char *cp;
        struct dpi_set_packet *ret_value;
        char *data;

        /* obj_id = object ID of variable, like 1.3.6.1.2.1.1.1 */
        /* should be identical to object ID sent in get request */
        data = "a string to be returned";
        ret_value = mkDPIset(obj_id,SNMP_TYPE_STRING,
                        strlen(data)+1,data);
        cp = mkDPIresponse(0,ret_value);
```

SET PROCESSING

Processing a SET request is only slightly more difficult than a GET request. In this case, additional information is made available in the parse tree, namely the type, length and value to be set.

The sub-agent may return an error indication of "no such name" if the variable is unrecognized, just as in a GET request. If the variable is recognized, but cannot be set, an error indication of "no such name" should be also be returned, although it is tempting to return a "read only" error.

GET NEXT PROCESSING

GET-NEXT requests are the most complicated requests to process. After parsing a GET-NEXT request, the parse tree will contain two parameters. One is the object ID on which the GET-NEXT operation is being performed. The semantics of the operation are that the sub-agent is to return the name/type/value of the next variable it supports whose name lexicographically follows the passed object ID.

It is important to realize that a given sub-agent may support several discontiguous sections of the MIB tree. In such a situation it would be incorrect to jump from one section to another. This problem is correctly handled by examining the second parameter which is passed. This parameter represents the "reason" why the sub-agent is being called. It holds the prefix of the tree that the sub-agent had indicated it supported.

If the next variable supported by the sub-agent does not begin with that prefix, the sub-agent must return an error indication of "no such name". If required, the SNMP agent will call upon the sub-agent again, but pass it a different group prefix. This is illustrated in the discussion below:

Assume there are two sub-agents. The first sub-agent registers two distinct sections of the tree, A and C. In reality, the sub-agent supports variables A.1 and A.2, but it correctly registers the minimal prefix required to uniquely identify the variable class it supports.

The second sub-agent registers a different section, B, which appears between the two sections registered by the first agent.

If a remote management station begins dumping the MIB, starting from A, the following sequence of queries would be performed:

```
  Sub-agent 1 gets called:
         get-next(A,A) == A.1
         get-next(A.1,A) = A.2
         get-next(A.2,A) = error(no such name)

  Sub-agent 2 is then called:
         get-next(A.2,B) = B.1
         get-next(B.1,B) = error(no such name)

  Sub-agent 1 gets called again:
         get-next(B.1,C) = C.1
```

REGISTER REQUESTS

A sub-agent must register the variables it supports with the SNMP agent. The appropriate packets may be created using the DPI API library routine mkDPIregister().

```
      unsigned char *cp;

      cp = mkDPIregister("1.3.6.1.2.1.1.2.");
```

NOTE: object IDs are registered with a trailing dot (".").

TRAP REQUESTS

A sub-agent can request that the SNMP agent generate a trap for it. The sub-agent must provide the desired values for the generic and specific parameters of the trap. It may optionally provide a name/type/value parameter that will be included in the trap packet. The DPI API library routine mkDPItrap() can be used to generate the required packet.

## DPI API LIBRARY ROUTINES

This section documents Application Program Interfaces to the DPI.

### QUERY_DPI_PORT()

```
int port;
char *hostname, *community_name;

port = query_DPI_port(hostname, community_name);
```

The query_DPI_port() function is used by a DPI client to determine what TCP port number is associated with the DPI. This port number is needed to connect() to the SNMP agent. If the port cannot be determined, -1 is returned.

The function is passed two arguments: a string representing the host's name or IP address and the community name to be used when making the request.

This function enables a DPI client to "bootstrap" itself. The port number is obtained via an SNMP GET request, but the DPI client does not have to be able to create and parse SNMP packets--this is all done by the query_DPI_port() function.

NOTE: the query_DPI_port() function assumes that the community name does not contain any null characters. If this is not the case, use the _query_DPI_port() function which takes a third parameter, the length of the community name.

### MKDPIREGISTER

```
#include "snmp_dpi.h"

unsigned char *packet;
int len;

/* register sysDescr variable */
packet = mkDPIregister("1.3.6.1.2.1.1.1.");

len = *packet * 256 + *(packet + 1);
len += 2;  /* include length bytes */
```

The mkDPIregister() function creates the necessary register-request packet and returns a pointer to a static buffer holding the packet contents. The null pointer (0) is returned if there is an error detected during the creation of the packet.

The length of the remainder packet is stored in the first two bytes of the packet, as demonstrated in the example above.

NOTE: object identifiers are registered with a trailing dot (".").

MKDPISET

```
#include "snmp_dpi.h"

struct dpi_set_packet *set_value;

char *obj_id;
int type, length;
char *value;

set_value = mkDPIset(obj_id, type, length, value);
```

The mkDPIset() function can be used to create the portion of a parse tree that represents a name/value pair (as would be normally be returned in a response packet). It returns a pointer to a dynamically allocated parse tree representing the name/type/value information. If there is an error detected while creating the parse tree, the null pointer (0) is returned.

The value of type can be one of the following (which are defined in the include file "snmp_dpi.h"):

- SNMP_TYPE_NUMBER
- SNMP_TYPE_STRING
- SNMP_TYPE_OBJECT
- SNMP_TYPE_INTERNET
- SNMP_TYPE_COUNTER
- SNMP_TYPE_GAUGE
- SNMP_TYPE_TICKS

The value parameter is always a pointer to the first byte of the object’s value.

NOTE: the parse tree is dynamically allocated and copies are made of the passed parameters. After a successful call to mkDPIset(), they can be disposed of in any manner the application chooses without affecting the parse tree contents.

MKDPIRESPONSE

```
#include "snmp_dpi.h"

unsigned char *packet;
```

```
int error_code;
struct dpi_set_packet *ret_value;

packet = mkDPIresponse(error_code, ret_value);

len = *packet * 256 + *(packet + 1);
len += 2;  /* include length bytes */
```

The mkDPIresponse() function creates an appropriate response packet. It takes two parameters. The first is the error code to be returned. It may be 0 (indicating no error) or one of the following (which are defined in the include file "snmp_dpi.h"):

- SNMP_NO_ERROR
- SNMP_TOO_BIG
- SNMP_NO_SUCH_NAME
- SNMP_BAD_VALUE
- SNMP_READ_ONLY
- SNMP_GEN_ERR

If the error code indicates no error, then the second parameter is a pointer to a parse tree (created by mkDPIset()) which represents the name/type/value information being returned. If an error is indicated, the second parameter is passed as a null pointer (0).

If the packet can be created, a pointer to a static buffer containing the packet contents is returned. This is the same buffer used by mkDPIregister(). If an error is encountered while creating the packet, the null pointer (0) is returned.

The length of the remainder packet is stored in the first two bytes of the packet, as demonstrated in the example above.

NOTE: mkDPIresponse() always frees the passed parse tree.

MKDPITRAP

```
#include "snmp_dpi.h"

unsigned char *packet;

int generic, specific;
struct dpi_set_packet *ret_value;

packet = mkDPItrap(generic, specific, ret_value);

len = *packet * 256 + *(packet + 1);
len += 2;  /* include length bytes */
```

The mkDPItrap() function creates an appropriate trap request packet. The first two parameters correspond to to value of the generic and specific fields in the SNMP trap packet. The third field can be used to pass a name/value pair to be provided in the SNMP trap packet. This information is passed as the set-packet portion of the parse tree. As an example, a linkDown trap for interface 3 might be generated by the following:

```
struct dpi_set_packet *if_index_value;
unsigned long data;
unsigned char *packet;
int len;

data = 3;  /* interface number = 3 */
if_index_value = mkDPIset("1.3.6.1.2.1.2.2.1.1", SNMP_TYPE_NUMBER,
        sizeof(unsigned long), &data);
packet = mkDPItrap(2, 0, if_index_value);
len = *packet * 256 + *(packet + 1);
len += 2;  /* include length bytes */
write(fd,packet,len);
```

If the packet can be created, a pointer to a static buffer containing the packet contents is returned. This is the same buffer used by mkDPIregister(). If an error is encountered while creating the packet, the null pointer (0) is returned.

The length of the remainder packet is stored in the first two bytes of the packet, as demonstrated in the example above.

NOTE: mkDPItrap() always frees the passed parse tree.

PDPIPACKET

```
#include "snmp_dpi.h"

unsigned char *packet;

struct snmp_dpi_hdr *hdr;

hdr = pDPIpacket(packet)
```

The pDPIpacket() function parses a DPI packet and returns a parse tree representing its contents. The parse tree is dynamically allocated and contains copies of the information within the DPI packet. After a successful call to pDPIpacket(), the packet may be disposed of in any manner the application chooses without affecting the contents of the parse tree. If an error is encountered during the parse, the null pointer (0) is returned.

NOTE: the relevant parse tree structures are defined in the include file "snmp_dpi.h", and that file remains the definitive reference.

The root of the parse tree is represented by a snmp_dpi_hdr structure:

```
struct snmp_dpi_hdr {
    unsigned char  proto_major;
    unsigned char  proto_minor;
    unsigned char  proto_release;

    unsigned char  packet_type;
    union {
         struct dpi_get_packet    *dpi_get;
         struct dpi_next_packet   *dpi_next;
         struct dpi_set_packet    *dpi_set;
         struct dpi_resp_packet   *dpi_response;
         struct dpi_trap_packet   *dpi_trap;
    } packet_body;
};
```

The field of immediate interest is packet_type. This field can have one of the following values (which are defined in the include file "snmp_dpi.h"):

- SNMP_DPI_GET
- SNMP_DPI_GET_NEXT
- SNMP_DPI_SET

The packet_type field indicates what request is being made of the DPI client. For each of these requests, the remainder of the packet_body will be different.

If a get request is indicated, the object ID of the desired variable is passed in a dpi_get_packet structure:

```
struct dpi_get_packet {
    char *object_id;
};
```

A get-next request is similar, but the dpi_next_packet structure also contains the object ID prefix of the group that is currently being traversed:

```
struct dpi_next_packet {
    char *object_id;
    char *group_id;
};
```

If the next object whose object ID lexicographically follows the object ID indicated by object_id does not begin with the suffix indicated by group_id, the DPI client must return an error indication of SNMP_NO_SUCH_NAME.

A set request has the most amount of data associated with it and this is contained in a dpi_set_packet structure:

```
struct dpi_set_packet {
    char      *object_id;
    unsigned char  type;
    unsigned short value_len;
    char      *value;
};
```

The object ID of the variable to be modified is indicated by object_id The type of the variable is provided in type and may have one of the following values:

- SNMP_TYPE_NUMBER
- SNMP_TYPE_STRING
- SNMP_TYPE_OBJECT
- SNMP_TYPE_EMPTY
- SNMP_TYPE_INTERNET
- SNMP_TYPE_COUNTER
- SNMP_TYPE_GAUGE
- SNMP_TYPE_TICKS

The length of the value to be set is stored in value_len and value contains a pointer to the value.

NOTE: the storage pointed to by value will be reclaimed when the parse tree is freed. The DPI client must make provision for copying the value contents.

FDPIPARSE

```
#include "snmp_dpi.h"

struct snmp_dpi_hdr *hdr;

fDPIparse(hdr);
```

The routine fDPIparse() frees a parse tree previously created by a call to pDPIpacket This routine is declared as void--it has no return value.

NOTE: after calling fDPIparse(), no further references to the parse

tree can be made.

## AGENT IMPLEMENTATION ISSUES

Although the SNMP DPI protocol is completely documented in this paper, the document itself is somewhat biased towards clearly defining the interface provided to sub-agents (i.e., it provides a specification of a C language API). This detailed coverage is possible because the client side of the interface is completely self-contained.

The agent side of the interface has to be integrated into individual vendor implementations, many of which may have a unique organizational structure in an attempt to address various performance and storage constraints. This makes it infeasible to provide much more than suggestions for SNMP agent implementers. Unfortunately, this leaves room for a large amount of interpretation which can lead to implementations that don't necessarily work they way they should--too much ambiguity can be a bad thing.

The following characteristics of an agent implementation are to be considered mandatory:

## DUPLICATE REGISTRATIONS

With this release of the protocol, order of registration is significant. The last sub-agent to register a variable is the one that is deemed to be authoritative. Variables implemented by the base SNMP agent are considered to have been registered prior to any sub-agent registrations. Thus sub-agents may re-implement support for variables that were incorrectly implemented by a vendor.

## AUTOMATIC DEREGISTRATION ON CLOSE

All SNMP DPI connections are carried over a stream connection. When the connection is closed by the client (no matter what the cause), the agent must automatically unregister all of the variables that were registered by the sub-agent.

## TIMELY RESPONSE CONSTRAINTS

A sub-agent must respond to a request in a timely fashion. In this version of the protocol, we specify that a sub-agent must respond to a request by the SNMP agent within 5 seconds. If the sub-agent does not respond in time, the SNMP agent should terminate the connection and unregister all of the variables that were previously registered by the sub-agent in question.

NOTE: agent implementations that do not have access to a timer may not be able to implement this. In that case, they leave themselves open to being placed in a state where they are blocked forever if the sub-agent malfunctions.

## SUPPORT FOR MULTIPLE MIB VIEWS

Some agents allow different MIB views to be selected based on the community name used. It is not the intention of this document to pass judgement on the various approaches that have been proposed or implemented, but instead merely to recognize the existence of implementations that support this feature.

The point of this discussion is to specify clearly that objects supported by an SNMP DPI sub-agent are to be registered under the MIB view that was selected by the community name used in the SNMP GET request that obtained the DPI_port value.

The SNMP DPI does not specify a reserved port, but instead sub-agents bootstrap themselves by making an SNMP GET request for the DPI_port variable. This variable represents the TCP port to which the sub-agent should connect. It should be understood that there is no reason why the SNMP agent cannot have several listens (passive opens) active, each corresponding to a distinct MIB view. The port number returned then would be different based on the community name used in the SNMP GET request for the DPI_port variable.

## CONSIDERATIONS FOR THE NEXT RELEASE

The SNMP DPI protocol makes provision for extension and parallel use of potentially incompatible releases. The discussion above documents the protocol as it is currently in use and has not discussed features of interest that should be considered for a future revision.

## UNREGISTER

For closure, an UNREGISTER request could be of use.

## SUPPORT FOR ATOMIC SETS

The SNMP protocol [1] specifies that:

> Each variable assignment specified by the SetRequest-PDU should be effected as if simultaneously set with respect to all other assignments specified in the same message.

The SNMP DPI has no provision for backing out a successfully processed SET request if one of the subsequent variable assignments

fails. This omission is a reflection of several biases:

- o the SNMP DPI was intended to be light-weight.
- o a belief that the SNMP RFC prescribes semantics which are infeasible to implement unless the range of applications is restricted.

It has been suggested that a new request, TEST_SET, be added to the DPI protocol. Processing of a SET request would then be performed as follows:

- o all variables would be processed using TEST_SET unless any error occurred. The subagents would verify that they could process the request.
- o if no error occurred, each of the variables would be reprocessed, this time with a SET request.

A problem with such an approach is that it relies on the TEST_SET operation to make an assertion that the request can be successfully performed. If this is not possible, then it cannot be asserted that the prescribed semantics will be provided. Such situations do exist, for example, a SET request that causes the far-end channel service unit to be looped up--one does not know if the operation will be successful until it is performed.

## SAMPLE SNMP DPI API IMPLEMENTATION

The following C language sources show an example implementation of the SNMP DPI Application Programming Interface as it would be exposed to the sub-agents.

## SAMPLE SNMP DPI INCLUDE FILE

```
  /* SNMP distributed program interface */

  #define SNMP_DPI_GET            1
  #define SNMP_DPI_GET_NEXT       2
  #define SNMP_DPI_SET            3
  #define SNMP_DPI_TRAP           4
  #define SNMP_DPI_RESPONSE       5
  #define SNMP_DPI_REGISTER       6

  #define SNMP_DPI_PROTOCOL       2
  #define SNMP_DPI_VERSION        1
  #define SNMP_DPI_RELEASE        0

  /* SNMP error codes from RFC 1098 (1067) */
```

```
  #define SNMP_NO_ERROR           0
  #define SNMP_TOO_BIG            1
  #define SNMP_NO_SUCH_NAME       2
  #define SNMP_BAD_VALUE          3
  #define SNMP_READ_ONLY          4
  #define SNMP_GEN_ERR            5

  /* variable types */
  #define SNMP_TYPE_TEXT          0       /* textual representation */
  #define SNMP_TYPE_NUMBER        (128|1) /* number */
  #define SNMP_TYPE_STRING        2       /* text string */
  #define SNMP_TYPE_OBJECT        3       /* object identifier */
  #define SNMP_TYPE_EMPTY         4       /* no value */
  #define SNMP_TYPE_INTERNET      (128|5) /* internet address */
  #define SNMP_TYPE_COUNTER       (128|6) /* counter */
  #define SNMP_TYPE_GAUGE         (128|7) /* gauge */
  #define SNMP_TYPE_TICKS         (128|8) /* time ticks (1/100th sec) */
  #define SNMP_TYPE_MASK          0x7f    /* mask for type */


  struct dpi_get_packet {
          char    *object_id;
  };

  struct dpi_next_packet {
          char    *object_id;
          char    *group_id;
  };

  struct dpi_set_packet {
          char            *object_id;
          unsigned char   type;
          unsigned short  value_len;
          char            *value;
  };

  struct dpi_resp_packet {
          unsigned char   ret_code;
          struct dpi_set_packet   *ret_data;
  };

  struct dpi_trap_packet {
          unsigned char   generic;
          unsigned char   specific;
          struct dpi_set_packet   *info;
  };

  struct snmp_dpi_hdr {
```

```
          unsigned char   proto_major;
          unsigned char   proto_minor;
          unsigned char   proto_release;

          unsigned char   packet_type;
          union {
                  struct dpi_get_packet   *dpi_get;
                  struct dpi_next_packet  *dpi_next;
                  struct dpi_set_packet   *dpi_set;
                  struct dpi_resp_packet  *dpi_response;
                  struct dpi_trap_packet  *dpi_trap;
          } packet_body;
  };

  extern struct snmp_dpi_hdr *pDPIpacket();
  extern void fDPIparse();
  extern unsigned char *mkMIBquery();
  extern unsigned char *mkDPIregister();
  extern unsigned char *mkDPIresponse();
  extern unsigned char *mkDPItrap();
  extern struct dpi_set_packet *mkDPIset();
```

SAMPLE QUERY_DPI_PORT() FUNCTION

```
  #ifdef VM

  #include <manifest.h>
  #include <snmp_vm.h>
  #include <bsdtime.h>
  #include <bsdtypes.h>
  #include <socket.h>
  #include <in.h>
  #include <netdb.h>
  #include <inet.h>

  #else

  #include <sys/time.h>
  #include <sys/types.h>
  #include <sys/socket.h>
  #include <netinet/in.h>
  #include <netdb.h>
  #include <arpa/inet.h>

  #endif

  static unsigned char asn1_hdr[] = {0x30};
```

```
  /* insert length of remaining packet, not including this */
  static unsigned char version[] = {0x02, 0x01, 0x00, 0x04};

  /* integer, len=1, value=0, string */
  /* insert community name length and community name */
  static unsigned char request[] = {
      0xa0, 0x1b,         /* get request, len=0x1b */
      0x02, 0x01, 0x01,   /* integer, len=1,request_id = 1 */
      0x02, 0x01, 0x00,   /* integer, len=1, error_status = 0 */
      0x02, 0x01, 0x00,   /* integer, len=1, error_index = 0 */
      0x30, 0x10,         /* varbind list, len=0x10 */
      0x30, 0x0e,         /* varbind , len=0x0e */
      0x06, 0x0a,         /* object ID, len=0x0a */
      0x2b, 0x06, 0x01, 0x04, 0x01, 0x02, 0x02, 0x01, 0x01, 0x00,
      0x05, 0x00          /* value, len = 0 */
  };

  static          extract_DPI_port();

  query_DPI_port(hostname, community_name)
  char           *hostname;
  char           *community_name;
  {
      int             community_len;
      int             rc;

      community_len = strlen(community_name);

      rc = _query_DPI_port(hostname, community_name, community_len);
      return (rc);
  }

  /* use if community_name has embedded nulls */

  _query_DPI_port(hostname, community_name, community_len)
  char           *hostname;
  char           *community_name;
  int             community_len;
  {
      unsigned char   packet[1024];
      int             packet_len;
      int             remaining_len;
      int             fd, rc, sock_len;
      struct sockaddr_in sock, dest_sock;
      struct timeval  timeout;
      unsigned long   host_addr, read_mask;
      int             tries;
```

```
host_addr = lookup_host(hostname);
packet_len = 0;
bcopy(asn1_hdr, packet, sizeof(asn1_hdr));
packet_len += sizeof(asn1_hdr);

remaining_len = sizeof(version) + 1 +
  community_len + sizeof(request);

packet[packet_len++] = remaining_len & 0xff;
bcopy(version, packet + packet_len, sizeof(version));
packet_len += sizeof(version);
packet[packet_len++] = community_len & 0xff;
bcopy(community_name, packet + packet_len, community_len);
packet_len += community_len;
bcopy(request, packet + packet_len, sizeof(request));
packet_len += sizeof(request);

fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd < 0) {
return (-1);
}
bzero(&sock, sizeof(sock));
sock.sin_family = AF_INET;
sock.sin_port = 0;
sock.sin_addr.s_addr = 0;
rc = bind(fd, &sock, sizeof(sock));
if (rc < 0)
return (-1);
timeout.tv_sec = 3;
timeout.tv_usec = 0;
bzero(&dest_sock, sizeof(dest_sock));
dest_sock.sin_family = AF_INET;
dest_sock.sin_port = htons(161);
dest_sock.sin_addr.s_addr = host_addr;

tries = 0;
while (++tries < 4) {
rc = sendto(fd, packet, packet_len, 0, &dest_sock,
         sizeof(dest_sock));
read_mask = 1 << fd;
rc = select(read_mask + 1, &read_mask, 0, 0, &timeout);
if (rc <= 0)
    continue;
sock_len = sizeof(dest_sock);
packet_len = recvfrom(fd, packet, sizeof(packet), 0,
                &dest_sock, &sock_len);
if (packet_len <= 0) {
    return (-1);
```

```
        }
        rc = extract_DPI_port(packet, packet_len);
        return (rc);
        }
        return (-1);
    }

    static          extract_DPI_port(packet, len)
    unsigned char   packet[];
    int             len;

    {
        int             offset;
        int             port;

        /* should do error checking (like for noSuchName) */
        offset = len - 2;
        port = (packet[offset] << 8) + packet[offset + 1];
        return (port);
    }
```

SAMPLE DPI FUNCTIONS

```
    /* DPI parser */

    #ifdef VM
    #include "manifest.h"
    #endif

    #include "snmp_dpi.h"

    static struct dpi_get_packet *pDPIget();
    static struct dpi_next_packet *pDPInext();
    static struct dpi_set_packet *pDPIset();
    static struct dpi_trap_packet *pDPItrap();
    static struct dpi_resp_packet *pDPIresponse();

    static void     fDPIget();
    static void     fDPInext();
    static void     fDPIset();
    static void     fDPItrap();
    static void     fDPIresponse();

    static int      cDPIget();
    static int      cDPInext();
    static int      cDPIset();
    static int      cDPItrap();
    static int      cDPIresponse();
```

```
  static struct snmp_dpi_hdr *mkDPIhdr();
  static struct dpi_get_packet *mkDPIget();
  static struct dpi_next_packet *mkDPInext();
  struct dpi_set_packet *mkDPIset();


  extern char    *malloc();

  static unsigned char new_packet[1024];
  static int      packet_len;

  struct snmp_dpi_hdr *pDPIpacket(packet)
  unsigned char  *packet;
  {
      struct snmp_dpi_hdr *hdr;
      int             len, offset;


      hdr = (struct snmp_dpi_hdr *) malloc(sizeof(struct snmp_dpi_hdr));
      if (hdr == 0)
          return (0);

      len = (packet[0] << 8) + packet[1];
      len += 2;
      offset = 2;
      hdr->proto_major = packet[offset++];
      hdr->proto_minor = packet[offset++];
      hdr->proto_release = packet[offset++];
      hdr->packet_type = packet[offset++];
      switch (hdr->packet_type) {
      case SNMP_DPI_GET:
      case SNMP_DPI_REGISTER:
          hdr->packet_body.dpi_get =
            pDPIget(packet + offset, len - offset);
          break;
      case SNMP_DPI_GET_NEXT:
          hdr->packet_body.dpi_next =
            pDPInext(packet + offset, len - offset);
          break;
      case SNMP_DPI_SET:
          hdr->packet_body.dpi_set =
            pDPIset(packet + offset, len - offset);
          break;
      case SNMP_DPI_TRAP:
          hdr->packet_body.dpi_trap =
            pDPItrap(packet + offset, len - offset);
          break;
      case SNMP_DPI_RESPONSE:
```

```
            hdr->packet_body.dpi_response =
              pDPIresponse(packet + offset, len - offset);
            break;
        }
        return (hdr);
    }

    static struct dpi_get_packet *pDPIget(packet, len)
    unsigned char  *packet;
    int             len;
    {
        struct dpi_get_packet *get;
        int             l;

        get = (struct dpi_get_packet *)
          malloc(sizeof(struct dpi_get_packet));
        if (get == 0)
            return (0);
        l = strlen(packet) + 1;
        get->object_id = malloc(l);
        strcpy(get->object_id, packet);
        return (get);
    }

    static struct dpi_next_packet *pDPInext(packet, len)
    unsigned char  *packet;
    int             len;
    {
        struct dpi_next_packet *next;
        int             l;
        unsigned char  *cp;

        next = (struct dpi_next_packet *)
          malloc(sizeof(struct dpi_next_packet));
        if (next == 0)
            return (0);
        cp = packet;
        l = strlen(cp) + 1;
        next->object_id = malloc(l);
        strcpy(next->object_id, cp);
        cp += l;
        l = strlen(cp) + 1;
        next->group_id = malloc(l);
        strcpy(next->group_id, cp);
        return (next);
    }

    static struct dpi_set_packet *pDPIset(packet, len)
```

```
  unsigned char  *packet;
  int             len;
  {
      struct dpi_set_packet *set;
      int             l;
      unsigned char  *cp;

      if (len == 0)
          return (0);     /* nothing to parse */
      set = (struct dpi_set_packet *)
        malloc(sizeof(struct dpi_set_packet));
      if (set == 0)
          return (0);

      cp = packet;
      l = strlen(cp) + 1;
      set->object_id = malloc(l);
      strcpy(set->object_id, cp);
      cp += l;
      set->type = *(cp++);
      l = (*(cp++) << 8);
      l += *(cp++);
      set->value_len = l;
      set->value = malloc(l);
      bcopy(cp, set->value, l);
      return (set);
  }

  static struct dpi_trap_packet *pDPItrap(packet, len)
  unsigned char  *packet;
  int             len;
  {
      struct dpi_trap_packet *trap;

      trap = (struct dpi_trap_packet *)
        malloc(sizeof(struct dpi_trap_packet));
      if (trap == 0)
          return (0);

      trap->generic = *packet;
      trap->specific = *(packet + 1);
      trap->info = pDPIset(packet + 2, len - 2);
      return (trap);
  }

  static struct dpi_resp_packet *pDPIresponse(packet, len)
  unsigned char  *packet;
  int             len;
```

```
  {
      struct dpi_resp_packet *resp;

      resp = (struct dpi_resp_packet *)
        malloc(sizeof(struct dpi_resp_packet));
      if (resp == 0)
          return (0);

      resp->ret_code = *packet;
      resp->ret_data = pDPIset(packet + 1, len - 1);
      return (resp);
  }

  void            fDPIparse(hdr)
  struct snmp_dpi_hdr *hdr;
  {
      if (hdr == 0)
          return;
      switch (hdr->packet_type) {
      case SNMP_DPI_GET:
      case SNMP_DPI_REGISTER:
          fDPIget(hdr);
          break;
      case SNMP_DPI_GET_NEXT:
          fDPInext(hdr);
          break;
      case SNMP_DPI_SET:
          fDPIset(hdr);
          break;
      case SNMP_DPI_TRAP:
          fDPItrap(hdr);
          break;
      case SNMP_DPI_RESPONSE:
          fDPIresponse(hdr);
          break;
      }
      free(hdr);
  }

  static void     fDPIget(hdr)
  struct snmp_dpi_hdr *hdr;
  {
      struct dpi_get_packet *get;

      get = hdr->packet_body.dpi_get;
      if (get == 0)
          return;
      if (get->object_id)
```

```
            free(get->object_id);
        free(get);
    }

    static void     fDPInext(hdr)
    struct snmp_dpi_hdr *hdr;
    {
        struct dpi_next_packet *next;

        next = hdr->packet_body.dpi_next;
        if (next == 0)
            return;
        if (next->object_id)
            free(next->object_id);
        if (next->group_id)
            free(next->group_id);
        free(next);
    }

    static void     fDPIset(hdr)
    struct snmp_dpi_hdr *hdr;
    {
        struct dpi_set_packet *set;

        set = hdr->packet_body.dpi_set;
        if (set == 0)
            return;
        if (set->object_id)
            free(set->object_id);
        if (set->value)
            free(set->value);
        free(set);
    }

    static void     fDPItrap(hdr)
    struct snmp_dpi_hdr *hdr;
    {
        struct dpi_trap_packet *trap;
        struct dpi_set_packet *set;

        trap = hdr->packet_body.dpi_trap;
        if (trap == 0)
            return;

        set = trap->info;
        if (set != 0) {
            if (set->object_id)
                free(set->object_id);
```

```
            if (set->value)
                free(set->value);
            free(set);
        }
        free(trap);
    }

    static void     fDPIresponse(hdr)
    struct snmp_dpi_hdr *hdr;
    {
        struct dpi_resp_packet *resp;
        struct dpi_set_packet *set;

        resp = hdr->packet_body.dpi_response;
        if (resp == 0)
            return;

        set = resp->ret_data;
        if (set != 0) {
            if (set->object_id)
                free(set->object_id);
            if (set->value)
                free(set->value);
            free(set);
        }
        free(resp);
    }

    unsigned char  *cDPIpacket(hdr)
    struct snmp_dpi_hdr *hdr;
    {
        int             rc, len;
        if (hdr == 0) {
            return (0);
        }
        packet_len = 2;
        new_packet[packet_len++] = hdr->proto_major;
        new_packet[packet_len++] = hdr->proto_minor;
        new_packet[packet_len++] = hdr->proto_release;
        new_packet[packet_len++] = hdr->packet_type;
        switch (hdr->packet_type) {
        case SNMP_DPI_GET:
        case SNMP_DPI_REGISTER:
            rc = cDPIget(hdr->packet_body.dpi_get);
            break;
        case SNMP_DPI_GET_NEXT:
            rc = cDPInext(hdr->packet_body.dpi_next);
            break;
```

```
        case SNMP_DPI_SET:
            rc = cDPIset(hdr->packet_body.dpi_set);
            break;
        case SNMP_DPI_TRAP:
            rc = cDPItrap(hdr->packet_body.dpi_trap);
            break;
        case SNMP_DPI_RESPONSE:
            rc = cDPIresponse(hdr->packet_body.dpi_response);
            break;
        }
        if (rc == -1)
            return (0);
        len = packet_len - 2;
        new_packet[1] = len & 0xff;
        len >>= 8;
        new_packet[0] = len & 0xff;
        return (new_packet);
    }

    static int      cDPIget(get)
    struct dpi_get_packet *get;
    {
        if (get->object_id == 0)
            return (-1);

        strcpy(&new_packet[packet_len], get->object_id);
        packet_len += strlen(get->object_id) + 1;
        return (0);
    }

    static int      cDPInext(next)
    struct dpi_next_packet *next;
    {
        if (next->object_id == 0)
            return (-1);
        if (next->group_id == 0)
            return (-1);

        strcpy(&new_packet[packet_len], next->object_id);
        packet_len += strlen(next->object_id) + 1;
        strcpy(&new_packet[packet_len], next->group_id);
        packet_len += strlen(next->group_id) + 1;
        return (0);
    }

    static int      cDPIset(set)
    struct dpi_set_packet *set;
    {
```

```
    int             len;

    if (set->object_id == 0)
        return (-1);
    if ((set->value == 0) && (set->value_len != 0))
        return (-1);

    strcpy(&new_packet[packet_len], set->object_id);
    packet_len += strlen(set->object_id) + 1;
    new_packet[packet_len++] = set->type;
    len = set->value_len >> 8;
    new_packet[packet_len++] = len & 0xff;
    new_packet[packet_len++] = set->value_len & 0xff;
    bcopy(set->value, &new_packet[packet_len], set->value_len);
    packet_len += set->value_len;
    return (0);
}

static int      cDPIresponse(resp)
struct dpi_resp_packet *resp;
{
    int             rc;

    if (resp == 0)
        return (-1);

    new_packet[packet_len++] = resp->ret_code;
    if (resp->ret_data != 0) {
        rc = cDPIset(resp->ret_data);
    } else
        rc = 0;
    return (rc);
}

static int      cDPItrap(trap)
struct dpi_trap_packet *trap;
{
    int             rc;

    new_packet[packet_len++] = trap->generic;
    new_packet[packet_len++] = trap->specific;
    if (trap->info != 0)
        rc = cDPIset(trap->info);
    else
        rc = 0;
    return (rc);
}
```

```
unsigned char  *mkMIBquery(cmd, oid_name, group_oid, type, len, value)
int             cmd;
char           *oid_name, *group_oid;
int             type, len;
char           *value;
{
    struct snmp_dpi_hdr *hdr;
    unsigned char  *cp;

    hdr = mkDPIhdr(cmd);
    if (hdr == 0)
        return (0);
    switch (hdr->packet_type) {
    case SNMP_DPI_GET:
    case SNMP_DPI_REGISTER:
        hdr->packet_body.dpi_get = mkDPIget(oid_name);
        break;
    case SNMP_DPI_GET_NEXT:
        hdr->packet_body.dpi_next = mkDPInext(oid_name, group_oid);
        break;
    case SNMP_DPI_SET:
        hdr->packet_body.dpi_set =
          mkDPIset(oid_name, type, len, value);
        break;
    }
    cp = cDPIpacket(hdr);
    fDPIparse(hdr);
    return (cp);
}

unsigned char  *mkDPIregister(oid_name)
char           *oid_name;
{
    return (mkMIBquery(SNMP_DPI_REGISTER, oid_name));
}

unsigned char  *mkDPIresponse(ret_code, value_list)
int             ret_code;
struct dpi_set_packet *value_list;
{
    struct snmp_dpi_hdr *hdr;
    struct dpi_resp_packet *resp;
    unsigned char  *cp;

    hdr = mkDPIhdr(SNMP_DPI_RESPONSE);
    resp = (struct dpi_resp_packet *)
      malloc(sizeof(struct dpi_resp_packet));
    if (resp == 0) {
```

```
        free(hdr);
        return (0);
    }
    hdr->packet_body.dpi_response = resp;
    resp->ret_code = ret_code;
    resp->ret_data = value_list;
    cp = cDPIpacket(hdr);
    fDPIparse(hdr);
    return (cp);
}

unsigned char  *mkDPItrap(generic, specific, value_list)
int             generic, specific;
struct dpi_set_packet *value_list;
{
    struct snmp_dpi_hdr *hdr;
    struct dpi_trap_packet *trap;
    unsigned char  *cp;

    hdr = mkDPIhdr(SNMP_DPI_TRAP);
    trap = (struct dpi_trap_packet *)
      malloc(sizeof(struct dpi_trap_packet));
    if (trap == 0) {
        free(hdr);
        return (0);
    }
    hdr->packet_body.dpi_trap = trap;
    trap->generic = generic;
    trap->specific = specific;
    trap->info = value_list;
    cp = cDPIpacket(hdr);
    fDPIparse(hdr);
    return (cp);
}

static struct snmp_dpi_hdr *mkDPIhdr(type)
int             type;
{

    struct snmp_dpi_hdr *hdr;

    hdr = (struct snmp_dpi_hdr *) malloc(sizeof(struct snmp_dpi_hdr));
    if (hdr == 0)
        return (0);
    hdr->proto_major = SNMP_DPI_PROTOCOL;
    hdr->proto_minor = SNMP_DPI_VERSION;
    hdr->proto_release = SNMP_DPI_RELEASE;
    hdr->packet_type = type;
```

```
    return (hdr);
}

static struct dpi_get_packet *mkDPIget(oid_name)
char           *oid_name;
{
    struct dpi_get_packet *get;
    int             l;

    get = (struct dpi_get_packet *)
      malloc(sizeof(struct dpi_get_packet));
    if (get == 0)
        return (0);

    l = strlen(oid_name) + 1;
    get->object_id = malloc(l);
    strcpy(get->object_id, oid_name);
    return (get);
}

static struct dpi_next_packet *mkDPInext(oid_name, group_oid)
char           *oid_name;
char           *group_oid;
{
    struct dpi_next_packet *next;
    int             l;

    next = (struct dpi_next_packet *)
      malloc(sizeof(struct dpi_next_packet));
    if (next == 0)
        return (0);
    l = strlen(oid_name) + 1;
    next->object_id = malloc(l);
    strcpy(next->object_id, oid_name);
    l = strlen(group_oid) + 1;
    next->group_id = malloc(l);
    strcpy(next->group_id, group_oid);
    return (next);
}

struct dpi_set_packet *mkDPIset(oid_name, type, len, value)
char           *oid_name;
int             type;
int             len;
char           *value;
{
    struct dpi_set_packet *set;
    int             l;
```

```
      set = (struct dpi_set_packet *)
        malloc(sizeof(struct dpi_set_packet));
      if (set == 0)
          return (0);

      l = strlen(oid_name) + 1;
      set->object_id = malloc(l);
      strcpy(set->object_id, oid_name);
      set->type = type;
      set->value_len = len;
      set->value = malloc(len);
      bcopy(value, set->value, len);
      return (set);
  }
```

## SAMPLE SOURCES FOR ANONYMOUS FTP

The complete source to two SNMP DPI-related programs is available for anonymous ftp from the University of Toronto. The host name to use is "vm.utcs.utoronto.ca" (128.100.100.2). The files are in the "anonymou.204" minidisk, so one must issue a "cd anonymou.204" after having logged in. Don't forget to use the binary transmission mode.

### The Ping Engine

This program is an SNMP DPI sub-agent which allows network management stations to perform remote PINGs. The source to this applications is in the file "ping_eng.tarbin". The source to the SNMP DPI API is also contained within the archive.

### The DPI->SMUX daemon

This program illustrates what is required to include the SNMP DPI in an SNMP agent. This is actually a SMUX-based agent that works with the ISODE SNMP agent and provides an interface for SNMP DPI sub-agents. The source to this program is in the file "dpid.tarbin". ISODE 6.7, or later, is a prerequisite.

## References

[1] Case, J., Fedor, M., Schoffstall, M., and J. Davin, "Simple Network Management Protocol", RFC 1157, SNMP Research, Performance Systems International, Performance Systems International, MIT Laboratory for Computer Science, May 1990.

[2] Information processing systems - Open Systems Interconnection, "Specification of Abstract Syntax Notation One (ASN.1)",

International Organization for Standardization, International Standard 8824, December 1987.

[3] Information processing systems - Open Systems Interconnection, "Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)", International Organization for Standardization, International Standard 8825, December 1987.

[4] McCloghrie K., and M. Rose, "Management Information Base for Network Management of TCP/IP-based internets", RFC 1156, Performance Systems International and Hughes LAN Systems, May 1990.

[5] Rose, M., and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-based internets", RFC 1155, Performance Systems International and Hughes LAN Systems, May 1990.

[6] International Business Machines, Inc., "TCP/IP for VM: Programmer's Reference", SC31-6084-0, 1990.

[7] International Business Machines, Inc., "Virtual Machine System Facilities for Programming, Release 6", SC24-5288-01, 1988.

[8] International Business Machines, Inc., "TCP/IP Version 1.1 for OS/2 EE: Programmer's Reference", SC31-6077-1, 1990.

## Security Considerations

Security issues are not discussed in this memo.

Authors' Addresses

Geoffrey C. Carpenter
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Phone: (914) 945-1970

Email: gcc@watson.ibm.com

Bert Wijnen
IBM International Operations
Watsonweg 2
1423 ND Uithoorn
The Netherlands

Phone: +31-2975-53316

Email: wijnen@uitvm2.iinus1.ibm.com