CSC573 – Project
Exploring snmp

Milestone 2

Team Members:
Gaurav Kataria
Mike Cho
Omer Ansari
Vishal Bhargava

INDEX

Summary of snmp agent design:
This milestone goes into design details on the snmp agent. We have broken down the agent as a combination of separate APIs. This helps in assignation of work and manageability of the project.
Basically, the snmp agent is broken down into several components which are briefly discussed here. To get more details on each API, please read the respective portion of the document.

1. Socket Interface: this creates the socket to listen on port 161, and uses recvfrom and sendto functions to receive the data from the client, and send requests back to the client respectively.
2. BER decoder/encoder: the decoder part basically takes in the incoming ASN.1/BER encoded bitstream from the UDP payload and decodes it to regular ascii format. The encoder does the opposite of this.
3. Query Sanity Checker: this API ensures that all the fields in the incoming request are valid syntactically, and either "approves" of the request or throws the respective error message as perceived while parsing the fields of the packet.
4. Agent Core. This is the heart of the engine. It takes in the request, and pulls/writes the requested data from/to within the kernel and outputs it.
5. Response Message Creator is a suite of functions which depending upon the success/failure of the query, creates the respective packet format structure, ready to be handed over to the BER encoder, to get encoded prior to being sent out

All these APIs are used by the main() function (*defined in pseudocode in Appendix A*). In short, upon the initialization of the daemon, main() uses the Socket Interface to bind on port 161 UDP, and blocks until it receives a query. A received query is passed straight to the BER decoder, which creates an ascii version of the packet. This is then passed through the Query Sanity Checker (QSC) to verify each field in the snmp request, and check for its veracity. If the the check fails, the QSC churns out an error which triggers main() to call Response Message Creator (RMC) API to create an err packet, which is then sent to the BER encoder, and once encoded, sent to the client via a socket interface api. If the check passes, the request is sent to the agent core, which either extracts the requested data, or sets the specified variable on the system and returns this data/OID combination out, which main() sends back to the client in a similar way it would have sent the error message.

A test plan is also included in this documentation discussing various functional/failure test cases. **The same test plan also defines the specs for our demo also.** For these tests, an snmp client would be written in Perl. (documented in Appendix B)

Query Sanity Checker

NAME
=====

query_sanity_checker - checks the sanity of the values sent in the request received from the client, and populates a data structure to be used by main()

SYNOPSIS
========
#include <lib/snmpd_types.h>
#include <lib/oids_supported.h>

struct *checked_query_result query_sanity_checker(
struct decoded_packet_struct *decoded_output)

DESCRIPTION
===========

Query Sanity Checker is invoked by main() when the snmp payload has been decodedand the data retrieved by the client needs to be checked for sanity.
query_sanity_check is passed decoded_output (which is a pointer to the struct type decoded_packet_struct) as was received from the ber_decoder.
the structs checked_query_result, and query_sanity_checker are defined and explained in (lib/snmpd_types.h[1])
It verifies each field within the decoded_output and if there are is any error in any field, returns the error_status code in the checked_query_result.

If there is no error, the error_status = 0, the struct checked_query_result would contain the requested OID in checked_query_result->requested_oid.

This would be used by main() to extract the data from agent_core()

DETAIL
======

Basically, the decoded_output is a struct whose fields each have one to one mapping with the snmp message format.

the query_sanity_checker would employ (sub)functions within it to check field:
(all CAPS fields are defined in lib/snmpd_types.h)

note, each one of the (sub)functions below would output the respective ERR_* output as specified in lib/snmpd_types.h. If there is no error, they would return ERR_NOERROR

version - int check_version(version) check if the version is VERSION

community - int check_community(community,request_type) would verify

[1] See Appendix D

if request_type is SNMPSET, that the communty string is the read write community specified in the /etc/snmpd.cfg
if request_type is SNMPGET, the commstring is the RO commstring in the snmpd.cfg file

pdu_type - int check_pdu_type(pdu_type) would verify if the pdu type is SNMPGET, SNMPGETNEXT, SNMPGETRESPONSE or SNMPSET.

oid cleaned_up_OID[] = reformat_oid(decoded_packet_struct->OID) - would reformat the oid which would be received as a string from ber_decoder, to an array of integers.

check_oid - int check_oid(requested_oid) would use the libs/oids_supported.h file to verify if the oid received is
(a) a supported oid (by looking at cleaned_up_OID[6]: supported values are:
1 (system), 4 (ip), 5(icmp), and 6(tcp)
(b) based on cleaned_up_OID[6] check for the length of the oid,
length for
system = 9, ip = 15, icmp = 9, tcp = 20.

if length(cleaned_up_OID) > anticipated length,
then checked_query_result.error_status = ERR_TOOBIG
and return.

if length(cleaned_up_OID) < length, and
decoded_packet_struct.pdu_type != SNMPGETNEXT
then checked_query_result.error_status = ERR_NOSUCHNAME
and return.

(c) if cleaned_up_OID[6] is 1 or 5, check the COMPLETE oid as this OID belongs to a static branch. this would save the agent_core from having the worry about the "0" leaf at the end of the OID.

(d) verify if the first 6 OIDs match 1 3 6 1 2 1, and if not,
then checked_query_result.error_status = ERR_NOSUCHNAME
and return.

if all the above are not hit, then
then checked_query_result.error_status = ERR_NOERROR and
checked_query_result.requested_oid[] = cleaned_up_OID[] and
checked_query_result.request_id = decoded_packet_stuct.request_id;

along with the above, if decoded_output->pdu_type = SNMPSET,
then
checked_query_result->snmpset_value = decoded_packet_struct.snmpset_value;
and return.

Response Message Creator

NAME
====

create_good_msg, create_err_msg - a suite of api to create either a response packet for a successful query or for an unsuccessful query.

SYNOPSIS
========

```
struct packet_to_be_encoded *create_good_msg(
                                    struct decoded_packet_struct *decoded_output
                                    struct oid_datatype_value *oid_and_data)

struct packet_to_be_encoded *create_err_msg(
                                    struct decoded_packet_struct *decoded_output,
                                    struct checked_query_result *query_checker_output)
```

DESCRIPTION
==========

There are two types of messages that the above api's can generate.
A successful request would cause the create_good_msg function to be called from main().

This inputs to this function are two data structures. One is of type decoded_packet_struct, which was the output of the ber_decoder.
The other is a struct of type oid_datatype_value, which is the output from agent_core.

All the data structures referred to are documented in lib/snmpdh_types.h

If a request is incorrect, the create_err_msg would be called during main(). This also takes in the decoded_packet_struct type as one input, and takes a struct of type checked_query_result, which is the outcome of the query_sanity_checker.

main() checks for checked_query_result->error_status and upon a non zero value called create_err_msg.

Both functions have the same output type, which is a struct of type packet_to_be_encoded.

All the data structures referred to are documented in lib/snmpdh_types.h

DETAILS
=======

Basically, upon a successful query, the following happens:

The version, community and request_id fields are copied from the decoded_output structure to the respective fields in the packet_to_be_encoded structure.

The error_status field is set to ERR_NOERROR

The packet_to_be_encoded struct has within it a struct of type oid_data_type. Essentially, the struct oid_and_data as received from the agent_core earlier by main() is directly written into packet_to_be_encoded->oid_datatype_value.

then the create_good_msg function returns the pointer to the packet_to_be_encoded struct.

Upon an unsuccessful query,
the same info from decoded_output struct is copied into the packet_to_be_encoded.

However, the packet_to_be_encoded->error_status field is populated by checked_query_result->error_status.

Furthermore, if packet_to_be_encoded->error_status != 0, then
packet_to_be_encoded->oid_data_type->value = NULL
then create_err_msg returns with a pointer to the struct of packet_to_be_encoded.

Note, that create_err_msg can also be called if agent_core returns oid_datatype_value->value == -1.
In this instance, the packet_to_be_encoded->error_status is set to ERR_READONLY and create_err_msg returns.

AgentCore:

NAME
====

agent_core - get/set the value on the agent

SYNOPSIS
========

#include <lib/snmpd_types.h>

struct oid_datatype_value *agent_core(
short int request_type,
oid *request_oid,
char *set_value)

DESCRIPTION
==========

The Agent Core is responsible for

(a) verifying the OID "instance" and
(b) extracting or the requested data from the system for read requests and
(c) setting the value for the OID instance for write requests.

there would be three variables passed to agentcore.
.request_type (defined in lib/snmpd_types.h) is the request type (get, getnext or set) as defined in RFC 1157.
.the request_oid (type oid also defined in lib/snmpd_types.h) is the oid passed to this api.
.a string of the value to be set if the request_type is SNMPSET.
If request_type != SNMPSET, *set_value would be ignored.
Note this is received as a string, but would be casted to the respective datatype as defined in lib/snmpd_types.h)

The output would be a pointer to the struct of type oid_datatype_value (defined in lib/snmpd_types.h)

This would contain:
.the requested OID (could be the next OID containing valid data if the request_type == SNMPGETNEXT)
.the value in string format,
if the request_type == SNMPSET, then the new set value is returned.
if the OID which was tried to be SET is ReadOnly, oid_datatype_value->value = -1
(note: badValue attempted to be set would be caught on the client side)

.If the requested OID yields no value, oid_datatype_value->value = '\0' would be returned.

DETAILS:
========

Read/Write request types (generic/specific)

The read/write requests would be of two kinds:
specific (GET / SET / subsequent GET-NEXTs)
generic (initial GET-NEXT)

Specific queries are the exact OIDs down to the specific instance that is being queried. for example.
1.3.6.1.2.1.6.13.1.5.192.168.1.26.32884.152.1.2.66.22
(human readable form:
tcp.tcpConnTable.tcpConnEntry.tcpConnRemPort.192.168.1.26.32884.152.1.2.66.22)

If such an OID is received by the agent, it would know exactly what type of data to be fetched.

Generic queries are queries of valid format but which are not complete.
Only the initial GET-NEXT requests are allowed to make such queries.

for example:
.1.3.6.1.2.1.6.13.1.1
(human readable form:
tcp.tcpConnTable.tcpConnEntry.tcpConnState)

Here the OID is not complete, though it is of valid format.

note that 1, 3, 6 and so on would be referred to sub-identifiers.

Such a query would only be entertained by the agent core if the request type is GET-NEXT.

The SNMP AGENT is entertaining the following _major_ branches:

system
ipNetToMediaTable
icmp
tcpConnTable

A list of supported OIDs within the above branches can be obtained from <lib/oids_supported.h>

Using the different sub-intentifiers within the OIDs, the agentcore would decipher which major branch this query is for, and pass the OID to the relevant function:

struct oid_datatype_value *system_core(short int request_type, oid *request_oid)
struct oid_datatype_value *ip_core(short int request_type, oid *request_oid)
struct oid_datatype_value *icmp_core(short int request_type, oid *request_oid)
struct oid_datatype_value *tcp_core(short int request_type, oid *request_oid)

Tree types: (static/dynamic)

As you can notice there are two types of trees.
Static
---
system and icmp trees are static trees:

taking the example of sysDescr,
the OID branch is:
.1.3.6.1.2.1.1.1 = sysDescr

the leaf is: 0
making the grand OID to be: sysDescr.0
if you notice in the OID listings above, all the leaves for system and icmp are "0"
These are easier to cater for as each of the OID can be mapped to a specific function.

Dynamic:
---
ipNetToMediaTable and tcpConnTable are dynamic trees.

Taking ipNetToMediaIfIndex as an example,

the OID branch is:
.1.3.6.1.2.1.4.22.1.1 = ipNetToMediaIfIndex

the leaf is:
X.A.B.C.D

where X is the ifIndex of the interface, and
A.B.C.D is the IP address learnt from that interface (into the arpcache)
making the grand OID : ipNetToMediaIfIndex.X.A.B.C.D

every time there is an addition to the arp cache of the system, a new leaf would be created.

As you can see, this makes the tree dynamic.

Owing to the varying complexities of the major branches, it makes sense to break them up code wise also, each having its own separate functions.

Now let's look at each function separately:
NOTE:

Each function has different OIDs dataypes it needs to cater request for. But each OID value is treated as (char *) until the info gets to ber_encoder, where the datatypes defined in oids_supported.h are used.

system_core:
---
struct oid_datatype_value *system_core(short int request_type, oid *request_oid)

Referring to the OID listings above, each the OID within the system branch would map a function:

char *sysDescr()
(this would churn out the hostname of the device)

int sysObjectId -
(set as zero, as we havent registered our daemon as public domain)

char *sysUpTime() -
this would serve the sysUptime of the snmp daemon, utilizing:
/proc/uptime, and the time when the daemon would be started

char *sysContact(request_type) -
this would serve the system contact info from a locally saved file:
/etc/snmpd.cfg

depending on the request type, this function should also be able to set the sysContact variable.

char *sysName(request_type) -
the assigned name of this device from a locally saved file:
/etc/snmpd.cfg

depending on the request type, this function should also be able to set the sysName variable.

char *sysLocation(request_type) -
the location information on this system, as saved in /etc/snmpd.cfg

depending on the request type, this function should also be able to set the sysLocation variable.

int *sysServices() -
(set as 7, as this device is hosted on a unix machine hosting application (layer7) services

ip_core:
---
struct oid_datatype_value *ip_core(short int request_type, oid *request_oid)

The primary resource for the data for the ipNetToMediaTable would be :
/proc/net/arp

This special file exists on all linux platforms and is dynamically updated each time the arp table changes.

Arp entries can also be invalidated by read/write mechanism on the OID:
.1.3.6.1.2.1.4.22.1.4
ipNetToMediaType OBJECT-TYPE

This functionality would be achieved by using the SIOCDARP request on the respective arp entry using ioctl() (2)

icmp_core:

---
struct oid_datatype_value *icmp_core(short int request_type, oid *request_oid)

The primary resource for the data for the icmp table would be :
/proc/net/snmp

This special file exists on all linux platforms and is dynamically updated whenever there is ICMP type transaction.

tcp_core:
---
struct oid_datatype_value *tcp_core(short int request_type, oid *request_oid)

The primary resource for the data for the tcpConnTable would be :
/proc/net/tcp

This special file exists on all linux platforms and is dynamically updated each time any tcp connections are attempted/established/torn down on the device.

IMPORTANT NOTE:
we would not be demo'ing a tcp connection reset.
The inpcb (Internet Protocol Control Blocks) structures do not exist on linux platforms, as they do on BSD platforms (ref: TCP/IP Illustrated v2, W. Richard Stevens).
Because of which, there exists no mechanism on linux platforms to "hijack" a tcp connection and cause a RST packet to be sent on it to break the connection. There are crude mechanisms like killing the forked child process of the service which is catering for the tcp connection, but we have decided not to use these crude means to implement a tcp conn-reset.

Alleviating Performance Bottlenecks: proactive caching for get-next requests
==========

As get-next requests are almost always implemented by snmpwalks, it is certain that one get next request would have another getnext request for the next OID following it.

Because of this, it would deem multiple fopen calls on the /proc files inefficeint.

Thus, it would make sense to read the incoming get-next request, and depending on its depth, cache the results for all the prospective get-next requests that can follow based on the depth of that get next request.

e.g.

a get-next request of .1.3.6.1.2.1.5 is made.

The agent identifies this as:
.1.3.6.1.2.1.5 icmp

there are five branches under it:

.1.3.6.1.2.1.5.8.0 icmpInEchos.0 <---1
.1.3.6.1.2.1.5.9.0 icmpInEchoReps.0 <-2
.1.3.6.1.2.1.5.21.0 icmpOutEchos.0 <-3
.1.3.6.1.2.1.5.22.0 icmpOutEchoReps.0 <-4

the next branch being:
.1.3.6.1.2.1.6.13.1.1 ... tcpConnState

now, note: the client is only requesting (1) for now, but our agent would need to have enough intelligence in it so that it collects (2), (3), and (4) also, knowing fully well that this data would be requested by the client also.

thus (1) would be served to from the agent_core api, but (2), (3), (4) would be cached with a cache_timer of GN_CACHE_TIME seconds (as defined in lib/snmpd_types.h)

At each subsequent get-next request which hits the cache, the timer is checked, and if it has not expired, the value is served from the cache.

If during the walk the cache timer expires, another fopen is made on the proc table and the above process is repeated.

This way, the fopens on each proc file are throttled by a max rate of
1 fopen/GN_CACHE_TIME seconds
and we can see how this would greatly improve performance on the agent side.

At the cost of improving performance, we do acknowledge that we introduce the limitation of having a static cache, meaning that a change in the table during the time it is being walked might not be relayed if the cache is still fresh.
The client would get the snapshot of the data at the time his first getnext request for the walk hits the server and would be served data from this snapshot until the cache timer expires.

However, the probability of such a change happening during a walk is not that high, so the above algorithm is quite reasonable.

The query cache:
========

The query cache would be in the form of linked lists of structs.
Each struct node within the linked list would look something like this:

```
struct query_cache_node         {
            struct query_cache_node *next_node;
            oid *oid_enum;
            char *oid_data_value;
                                                                    }
```

at the time of an fopen of a /proc file, the file is traversed and its data is massaged.
at the same time, the query_cache_node is malloc'd
also at this time, a snapshot of the system time (t1) is taken.
The extracted data and the respective OID which points this data is then put in this node.

As more data is extracted, more nodes are malloc'd, their pointer put in the previous query_cache_node thus creating the linked list.

When the eof if reached on the /proc file, the last cache_node's *next_node is marked as null.

The data from the first node is then served to the client.

whenever it is ascertained that any subsequent getnext requests hit the same major branch's cache, the system time is checked (t2) and:

if t2 - t1 < GN_CACHE_TIME then,
the linked list is traversed until the queried oid equals
query_cache_node->oid_enum at which time query_cache_node->oid_data_value is served.

if t2 - t1 > GN_CACHE_TIME then,
another fopen on the /proc file is done and the process of creating another cache is instantiated again.

BER encoder/decoder

NAME
=====
BER encoder/decoder - a suite of functions providing the agent and the client
the API's to encode and decode snmp packet data using ASN.1/BER.

SYNOPSIS
========

struct *decoded_packet_struct ber_decoder(char *raw_msg_payload);

char *bit_stream_response ber_encoder(struct *packet_to_be_encoded);

these structs are defined in the lib/snmpd_types.h header.

DESCRIPTION
===========

The BER decoder is invoked by main() to convert the raw snmp-requests coming from the client to a structure of type ‘decoded_packet_struct ‘ defined in the description of snmpd_types.h
The BER encoder is invoked by main() to BER encode a packet of type ‘packet_to_be_encoded’ to a bitstream, which is then passed over on the network, by either the client or the agent.

DETAILS
=======

We would be using the ASN.1 modules from the Linux CMU SNMP project for the BER encoding and decoding. We would be writing a wrapper for a neat interface between our agent/client and the CMU BER encoding/decoding module. In the case of the encoder, we would be parsing the data from the structure ‘packet_to_be_encoded’ to a format which could be used by the modules provided by the CMU code, and pass back the bitstring to be sent back to the main program. In the decoder, the buffer is passed onto the BER decoder, which returns an object of the structure data type ‘decoded_packet_struct’

Test Plan.

*Mechanism***:** We shall be writing a suite of snmp client tools (see Snmp_Client portion of doc) which will enable us to test our agent.

The following functionality would be tested:

1. simple snmpget on variables.
(a)Simulating the real life environment, users would do snmpget queries on static trees which they know the complete OID of. Thus the test would incorporate querying variables from the system and the icmp branch.
(b) Failure tests
A failed query would be shown when a get request on an
.INCOMPLETE OID would be issued.
. INVALID OID would be issued.

2. snmpset on certain variables.
(a) An snmpset on the some read write variables would be attempted using the RW community string.
Upon a non error type return, an snmpget would be issued on the same oid to see if the value has changed.
(b) Failure tests
a failed query would be attempted
. using the readonly community string.
. attempting to set a ReadOnly variable (the variable would be shown to the group via the internet, using the snmp translate/search tool)

In both cases, it would be then verified using snmpget that the value has not changed.

3. snmpwalk on all trees.
(a) snmpwalk would be attempted with the right community string on all trees.
It would be shown how the walk would stop when the tree being walked ends.
(b) a sniffer trace output would be shown showing the sequence of snmpgetnexts as the walk proceeds.

Allocation of Work :

The work has been divided among the members as

Mike -
suite of snmp client software(perl) (primary)
query_sanity_checker (secondary),
response_msg creation api & snmp_main overall agent integration (primary)

Vishal -
ber/encoding decoding, (primary)
socket api on agent, (solo)
response_msg creation api & snmp_main overall integration (secondary)

Gaurav -
agent core (secondary)
ber/encoding decoding, (secondary)
query_sanity_checker (primary)

Oansari -
agent core (primary) ,
snmp_main integration (secondary)
suite of snmp client software(perl) (secondary)

Appendix A The main() Algorithm

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <lib/snmpd_types.h>

// refer to lib/snmpd_types.h for struct definition for all nonstandard
struct definitions


int passivesock(int *port)
   // argument:
   //    int *port = port to allocate and bind the server socket
   // passivesock - allocate and bind a server socket the *port
specified
   // return value = if the return value is <0, an error creating the
socket has occurred
   //                else, the socket decriptor value is returned


main()
   int socket_value = 0;                // socket descriptor returned
by passivesock()
   struct sockaddr_in fsin;             // the from address of a client
   unsigned int alen;                   // from address length
   char *buf;                           // "input" buffer; any size > 0
   char *return_buf;                    // "output" buffer; any size >
0
   unsigned int return_len;             // to address length


   socket_value = passivesock(161))
   // allocate and bind the server socket to udp port 161




   While(1)   {
   // infinite loop to accept requests

      if (recvfrom(socket_value, buf, sizeof(buf), 0, (struct sockaddr
*)&fsin, *&alen) < 0)
          errexit("recrfrom: %s\n", strerror(errno));
      //  buf will be sent to BER for decoding
      //  endpoint address will be used by sendto();

      decoded_packet_struct *decoded_output=ber_decoder(buf);
      // buf will be sent to BER decoder which will decode each field
of snmp pdu and populate decoded_output
      // and  return a pointer to the struct decoded_packet_struct
```

```
      checked_query_result *query_result =
query_sanity_checker(decoded_output);
      // *decoded_output will be passed to query_sanity_checker()
      // query_sanity_checker() will return a pointer of type struct
checked_query_result


      if (*query_result.error_status != 0)   {
         // create error message
         packet_to_be_encoded *response =
create_error_msg(decoded_output, query_result);
         // create_error_msg will create a snmp struct of type
packet_to_be_encoded upon an unsuccessful query
      }

      else   {

         oid_datatype_value *oid_result =
agent_core(query_result.requested_oid,query_result.request_type,
query_result.snmpset_value)
         if (oid_result.value == -1)   {
         // if oid_result.value == -1, then a set was attempted on a
read-only oid and an error has occurred
            *query_result.error_status = ERR_READONLY;
            packet_to_be_encoded  *response=
create_error_msg(decoded_output, query_result);
         }
         else   {
            packet_to_be_encoded
*response=create_good_msg(decoded_output,oid_result);
            // this is a valid query and an snmp struct of type
packet_to_be_encoded will be created
         }

      }

      return_buf = ber_encoder(response);
      //  reponse will be sent to BER for encoding



      sendto(socket_value, (char*)&return_buf, sizeof(return_buf), 0,
(struct sockaddr *)&fsin, sizeof(fsin);
      //  encoded output is sent to client using the fields specified
in recvfrom()




   }

}
```

Appendix B Snmp Client

NAME:

A suite of snmp commands to get/set data from any snmp agent.

SYNOPSIS
========

snmpget <IP> <community_string> <OID> [retries] [timeout]
snmpset <IP> <community_string> <OID> <type> <value> [retries] [timeout]
snmpgetnext <IP> <community_string> <OID> [retries] [timeout]
snmpwalk <IP> <community_string> <OID> [retries] [timeout]

DESCRIPTION
===========

The Snmp Client is essentially responsible for querying any snmp agent(daemon)

It has four different interface types for the enduser.

snmpget takes in three arguments, the name or IP address of the agent, the community string, and the <OID> that needs to be polled. this is to retrieve data for the specified oid.
snmpgetnext also takes the exact three args. The difference here is that is requests for the _next_ OID right after the <OID> specified for which the agent has data.
snmpwalk uses the same output, but has the ability to walk a certain tree depending on the depth of the <OID> specified.
snmpset takes the same args also, but also requires the type and the value fields to be set. The type specifying what type of the value is (string/integer etc) and the value specifying the actual value that is attempted to be set.
Furthermore, the <community_string> specified in this case needs to be the RW commstring.

DETAILS
========

The Snmp Client is going to be written in perl, using the Net::SNMP perl module. This has been bundled as part of the project so that it can be used as a test tool to test the snmp agent.
Note: any snmpclient can be used to test/demo the snmp agent (daemon) but with this tool, we decouple the need of having to install bulky snmp software on the machine for test/demo purposes.

The Net::SNMP module already has an API defined for the above commands except snmpwalk:
In order to show transparency in our work, we will even quote the APIs:
http://search.cpan.org/doc/DTOWN/Net-SNMP-4.0.1/lib/Net/SNMP.pm

Note though that we would need to use the snmpgetnext api and write a wrapper for snmpwalk.

Basically, it would be following Section 4.1.3.1 in RFC 1157:
http://www.faqs.org/rfcs/rfc1157.html

essentially the idea behind it is this:

A user uses the snmpwalk command to walk on the icmp branch .1.3.6.1.2.1.5 .
This translates into a get-next request for the same oid. thus get-next request of .1.3.6.1.2.1.5 is made to the specified agent.

there are five branches under icmp that we are supporting:

.1.3.6.1.2.1.5.8.0 icmpInEchos.0 <---1
.1.3.6.1.2.1.5.9.0 icmpInEchoReps.0 <-2
.1.3.6.1.2.1.5.21.0 icmpOutEchos.0 <-3
.1.3.6.1.2.1.5.22.0 icmpOutEchoReps.0 <-4

the next branch being:
.1.3.6.1.2.1.6.13.1.1 ... tcpConnState <--5

The data
The agent would return the value for (1), along with the OID for this value.
The client would display this oid/data and then do a get-next for this newly returned OID, and the process would continue, until the agent returns the OID in (5).
At this point, the snmpwalk script would realize that an OID has been returned which is of a different branch than the original request, and would stop sending the getnext requests.

Timeouts and retries:
==========

Though SNMP uses a stateless transport mechanism (UDP), there would be intelligence within the snmp client software to attempt [retries] number of retries for the same oid if the response does not come back from the agent within [timeout] time.

These are optional values though, and if they are not entered the default values of
retries = 1
timeout = 5 seconds
would be assumed.

Appendix C SNMP Message Types and Format

Snmp Message Types:

There are four SNMP message Types we are investigating in this project:
1. snmp get
2. snmp set
3. snmp get-next
4. snmp response

1-3 are requests made from the snmp client to the snmp agent (or snmp daemon).
4 is the response from the agent back to the client.
The functionality of the message types is discussed in detail in RFC 1157, and we also discuss these in the Agent_Core and the Snmp_Client portions of this document.
Snmp was designed so that all the above message types can be sent in one packet format.

Snmp Message Format:

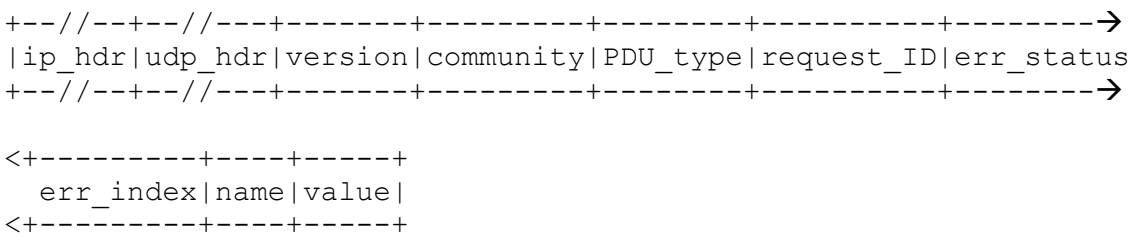

fig.a

(note in these details, only the IPhdr and the UDPhdr's length is mentioned.
This is because the the encoding used by snmp (ASN.1/BER) varies in the eventual bit length depending on the type of variable and its value.
However, the we do specify the maxval for the decoded fields)

Details on the fields

ip_hdr = IP header (normally) 20 bytes

udp_hdr = UDP header 8 bytes

version = version of snmp. in our project this would always be 0 (defined in lib/snmpd_types.h) (snmpv1 = 0, snmpv2 = 1 ..)

community = character string used as a cleartext password for authentication purposes before the request is entertained.

PDU_type = specifies the type of request (get/set/getnext/response) : (defined in lib/snmpd_types.h)

request_ID = set by the client in the query, and returned by the agent .
This is used to keep track of the UDP request.

err_status = error status, an integer value returned by the agent to specify the error (as laid down in RFC 1157, section 4.1.1) (defined in lib/snmpd_types.h)

err_index = error index, set by the agent to specify which value was in error. This is used when an snmp request was made with multiple OIDs in the same request packet.
length = short int

name = the OID for which the query is being made (inserted by the client)
in a get request the agent returns the same OID back in this field. In a getnext request, the agent would return the next valid OID after the OID specified in the request.

value = the value of this OID instance. This is ignored by the client when it is sending the request, and is populated by the agent if there is a valid value present for the OID being requested.

APPENDIX D: Header and Configuration Files

lib/snmpd_types.h

```
/* this file defines the various typedefs and constant enumerations
used by various subsystem components of the engine */

typedef u_long oid;
/* used as:
 * oid sysDescr[9] = {1 3 6 1 2 1 1 1 0};
 * as sysDescr.0 is : .1.3.6.1.2.1.1.1.0
 */

#define VERSION 0
#define SNMPGET 0
#define SNMPGETNEXT 1
#define SNMPGETRESPONSE 2
#define SNMPSET 3
#define GN_CACHE_TIME 5
#define ERR_NOERROR 0
#define ERR_TOOBIG     1
#define ERR_NOSUCHNAME 2
#define ERR_BADVALUE   3
#define ERR_READONLY   4
#define ERR_GENERR     5

// definition of datatypes.
//typedef char DisplayString [255]
//typedef int  TimeTicks
//typedef int          Integer
//commented out as this agent doesnt really use datatypes.
//the definitions for each oid are defined in oids_supported.h
//which would only be useful while ber encoding.
typedef struct checked_query_result {
        /*struct returned by query_sanity_checker to main() */
        short int error_status;
        // short int error_index;
        int request_id;
        // int *array_of_pointer_to_oids; /*where an oid is an array of
integers*/
        oid requested_oid[]; //array of u_long
        short int request_type;
        char *snmpset_value;
}

typedef struct decoded_packet_struct  {
        /* this is the struct returned by the ber_decoder.
         * as you can see, all the fields in this struct match the
         * incoming request packet's fields */
        short int version;
        char *community;
        short int pdu_type;
        int request_id;
        short int error_status;
        // short int error_index;
        // int *array_of_pointer_to_oids; /*where an oid is a string */
        char *OID; //anticipating an oid of type string from ber_decoder
```

```
        char *snmpset_value; // the value to set if this is an snmpset
request
                                                                    }


typedef struct packet_to_be_encoded {
    short int version;
    char *community;
    short int pdu_type;
    int request_id;
    short int error_status;
        const short int error_index = 0;
        /*if error_status != 0, then
         * packet_to_be_encoded->oid_data_type->value = NULL
         */
    /*int *array_of_oid_data_ *
         * //short int error_index; */
        struct oid_datatype_value *oid_data_type;
         /* Each struct would have the OID, its value and the type of
the value.
         * this would make the life of ber_decoder() easier */
        //int *array_of_oid_data_structs;
        /*this would be an array of struct oid_datatype_value.*/
}

typedef struct oid_datatype_value     {
        oid requested_oid[];
        //char *datatype;
        /*these types would be typedef'd in lib/snmpd_types.h */
        char *value;
        /*note: this value is returned as a string.
         * the ber_encoder would have to cast this value as
         * specified in the datatype */
}
```

lib/oids_supported.h

```
/* this header file has a list of all the fixed portions of the i
* supported OIDs: would be used by query_sanity_checker to verify
incoming
 * oid request */

#include

const oid c_sysDescr0[] = {1 3 6 1 2 1 1 1 0};
const oid c_sysObjectID0[] = {1 3 6 1 2 1 1 2 0};
const oid c_sysUpTime0[] = {1 3 6 1 2 1 1 3 0};
const oid c_sysContact0[] = {1 3 6 1 2 1 1 4 0};
const oid c_sysName0[] = {1 3 6 1 2 1 1 5 0};
const oid c_sysLocation0[] = {1 3 6 1 2 1 1 6 0};
const oid c_sysServices0[] = {1 3 6 1 2 1 1 7 0};

// note how we have included the instance 0 in the constant part of the
oid
// this way query_sanity_checker would throw an error msg by verifying
the
// oid and not sending something like .1.3.6.1.2.1.1.1.10 to agent_core

const oid c_ipNetToMediaIfIndex[] = {1 3 6 1 2 1 4 22 1 1};
const oid c_ipNetToMediaPhysAddress[] = {1 3 6 1 2 1 4 22 1 2};
const oid c_ipNetToMediaNetAddress[] = {1 3 6 1 2 1 4 22 1 3};
const oid c_ipNetToMediaType[] = {1 3 6 1 2 1 4 22 1 4};

const oid c_icmpInEchos0[] = {1 3 6 1 2 1 5 8 0}
const oid c_icmpInEchoReps[] = {1 3 6 1 2 1 5 9 0}
const oid c_icmpOutEchos[] = {1 3 6 1 2 1 5 21 0}
const oid c_icmpOutEchoReps[] = {1 3 6 1 2 1 5 22 0}

//note how we are checking the FULL oid for the icmp table above also,
similar to the system table above

const oid c_tcpConnState[] = {1 3 6 1 2 1 6 13 1 1};
const oid c_tcpConnLocalAddress[] = {1 3 6 1 2 1 6 13 1 2};
const oid c_tcpConnLocalPort[] = {1 3 6 1 2 1 6 13 1 3};
const oid c_tcpConnRemAddress[] = {1 3 6 1 2 1 6 13 1 4};
const oid c_tcpConnRemPort[] = {1 3 6 1 2 1 6 13 1 5};


// Below we define the datatypes for each of the supported OIDs. This
is
// to help the ber_encoder in encoding the values
const char dt_sysDescr0[] = "DisplayString"
const char dt_sysObjectID0[] = "OBJECT ID";
const char dt_sysUpTime0[] = "Timeticks";
const char dt_sysContact0[] = "DisplayString";
const char dt_sysName0[] = "DisplayString";
const char dt_sysLocation0[] = "DisplayString";
const char dt_sysServices0[] = "Integer";

const char dt_ipNetToMediaIfIndex[] = "Integer";
const char dt_ipNetToMediaPhysAddress[] = "PhysAddress";
const char dt_ipNetToMediaNetAddress[] = "IpAddress";
```

```
const char dt_ipNetToMediaType[] = "Integer";

const char dt_icmpInEchos0[] = "Counter";
const char dt_icmpInEchoReps[] = "Counter";
const char dt_icmpOutEchos[] = "Counter";
const char dt_icmpOutEchoReps[] = "Counter";


const char dt_tcpConnState[] = "Integer";
const char dt_tcpConnLocalAddress[] = "IpAddress";
const char dt_tcpConnLocalPort[] = "Integer";
const char dt_tcpConnRemAddress[] = "IpAddress";
const char dt_tcpConnRemPort[] = "Integer";
```

/etc/snmpd.cfg

```
RO public // readonly community string
RW private // readwrite community string
sysContact snmp_boy //sysContact field
sysName snmp_toy //sysName field
sysLocation snmp_ahoy //sysLocation field
```