CSC 573 Project Final Report
Spring, 2002

gmov-snmp v0.1
http://gmov-snmp.sourceforge.net

Team members
Gaurav Kataria (gkatari@unity.ncsu.edu)
Michael Cho (mcho@eos.ncsu.edu)
Omer Ansari (osansari@unity.ncsu.edu)
Vishal Bhargava (vbharga@unity.ncsu.edu)

Introduction:

The primary objective of this project was to write an snmp daemon to support some branches[1] of RFC1213 relevant to this course namely:
system,
ipNetToMediaTable,
icmp[2], and
tcpConnTable

The snmp daemon was broken up into several individual components discussed in the low level design. Furthermore, an snmp client suite was written to test the the daemon.The daemon was written to support the following snmp query types, namely,
snmpget, snmpgetnext, snmpset, and snmpwalk.

The snmp client suite comprised of a separate script each handing each of the above query type respectively. Each of the above was written in perl. They provided the means to test the agent (to follow later) but are not part of the primary goal (i.e. developing the agent) of the project.

The secondary motivation to write this software was to provide an alternative to the already existing popular snmp daemon (net-snmp). In this thrust, all our code is opensource, under the GNU Public License Agreement, and is available at:
http://gmov-snmp.sourceforge.net

[1]These branches can be looked up at: http://jaguar.ir.miami.edu/~marcus/snmptrans.html#
[2]Only 4 OIDs icmpInEchos, icmpInEchoReps, icmpOutEchos, icmpOutEchoReps

Background:

SNMP is the defacto protocol in the Networking Industry used to manage Network devices. In this project we wanted to explore the innards of an snmp server (agent) software and thus learn the functionality and design issues when writing an snmp agent.

SNMPv1, or Simple Network Management Protocol, is a standard defined by RFC 1155 and RFC 1157 for the remote management of network devices. There are two parts to a network management system – a manager (client) and an agent (server). The manager is the interface used by the network administrator to perform actions on the agent. An agent is a network device that has a predefined MIB, Management Information Base, which includes the objects/variables that are to be managed. The objects/variable can include, but are not limited to hardware information, statistics, and configuration parameters about a particular network device. SNMP has 5 types of messages that define the interaction between the manager and agent: get-request, get-next-request, set-request, get-response, and trap. For this interaction, UDP port 162 is used on the manager and UDP port 161 is used on the client. The encoding for the SNMP is Basic Encoding Rules, BER and the syntax used is Abstract Syntax Notation One, ASN.1.
The Managed Information Base, MIB, is the collection of information that can be queried and also set at the agent. Each object has a unique identifier referred to as the OID, or Object Identifier. For further information on the MIB, refer to the SMI given in RFC 1155.

High Level Design

The SnmpAgent:

Though there are internally, various facets to the agent, there really is only 1 binary used to invoke the snmp agent. The help can be invoked command line and it is self explanatory.

```
r2d3_knail1-> snmpd -?
[gmov-snmp v0.1] usage:
 snmpd [-h] [-p <portnumber>] [-v]
 -p which UDP port to listen on (default 161)
 -v to turn verbosity on
 -h to print (this) help msg
```

So an example to run the snmpd would be:

```
r2d3_knail1-> ./snmpd -p 160 -v
[gmov-snmpd v0.1] SNMP Daemon Started
Attempting to listen on port 160..Ready
Verbose Mode on..
...
```

SnmpAgent outputs:

Really, the ultimate output that is seen is the response that the snmp client receives..(that is discussed in the snmp client output a little further below. However, we have provided hooks to show and troubleshoot the agent functionality.

If the snmpd is run in the foreground (and not piped to /dev/null) you would also see one line stating the response string being sent back to the client.

```
r2d3_knail1-> ./snmpd -p 160
[Main] ResponseString:  0 .1.3.6.1.2.1.5.8.0 0 51216
```

In the above example, an snmpget on icmpInEchos is made (the data is 0, and the request_id of the snmp packet is 51216)

If the snmpd is run with the verbose flag (-1) you would see complete details of what was received, how each component of the snmp agent parsed the input, populated the requisited data structs, and sends the packet out.

```
r2d3_knail1-> ./snmpd -p 160 -v
[gmov-snmpd v0.1] SNMP Daemon Started
Attempting to listen on port 160..Ready
Verbose Mode on..
[Main] Decoded Fields: version:0, commstr:public, pdutype:0, reqid:8088, err:0, setval:-1
[Main] Decoded OID: 1.3.6.1.2.1.1.1.0.-1 -1.-1.-1.-1.-1 -1.-1.-1.-1.-1
```

the above is what the fields are of the incoming packet

```
[Main] CheckedQueryResult: error_status: 0 request_id: 8088 request_type: 0
snmpset_value: -1 oid_length: 9
[Main] CheckedQueryResult: Oid: 1.3.6.1.2.1.1.1.0.-1 -1.-1.-1.-1.-1 -1.-1.-1.-1.-1
```

the above is the output of checked query result (discussed in low level design)

```
[Main] AgentCore Output: (OID: data): 1.3.6.1.2.1.1.1.0.-1 -1.-1.-1.-1.-1 -1.-1.-1.-1.-1:
r2d3: Linux version 2.4.17
```

the above is what AgentCore pulls out from the kernel, after it receives the respective oid (i.e. SysDescr.0)

```
[Main] QueryResponse: version:0 commstr:public req_type:0, reqid:8088 err:0
[Main] QueryResponse: ResponseOID:.1.3.6.1.2.1.1.1.0, Data:r2d3: Linux version 2.4.17
```

the above is what the output packet struct looks like right before it is sent out.

```
[Main] ResponseString:  0 .1.3.6.1.2.1.1.1.0 r2d3: Linux version 2.4.17 8088
```

the above is the output string that is sent back to the snmp client

Snmp Agent Configuration:

This is in “etc/snmpd.cfg”

```
#this the config file for gmov-snmp
RO_Community:public
RW_Community:private
```

The two lines specify the Read and the Write community strings, usage specified in RFC1213.

For sysContact and sysLocation, separate .cfg files are maintained. This is to restrict any overwriting and file lock issues to that certain file.
These can be modified by hand on the server, or changed by issuing snmpset from client:
“etc/syscontact.cfg”

```
#this the sysContact file for gmov-snmp
sysContact:Michael.Langdon
```

“etc/syslocation.cfg”

```
#this the syslocation file for gmov-snmp
sysLocation:Heaven.Bound
```

The SnmpClient Api:

This comprises of 4 separate perl scripts.
snmpget.pl, snmpget.pl, snmpgetnext.pl and **snmpwalk.pl**
Each of them sends the respective type of query as specified in RFC1213.
Simply run just the script without arguments to see usage:
e.g.:

```
r2d4_knail1-> snmpgetnext.pl
snmpGetNext: [gmov-snmp v0.1]
usage: snmpgetnext.pl <host> <commstring> <oid>
<host>          : the snmp agent you want to poll
<commstring>    : the community string you want to use
<oid>           : the OID you want to poll, can be textual also
```

The output of each of the above commands is different. Here is the listing of each[3]:

[3]Note: the above snmpwalk output is limited to ip, we have the complete output in the FunctionalityTestCases submitted with the source code.

```
unity% snmpget.pl 66.26.37.246 public sysContact.0
*sysContact.0: Mike killah Cho
```

the above query was made on an snmpget and the response was likewise

```
unity% ./snmpgetnext.pl 66.26.37.246 public sysContact.0
*sysName.0: r2d3
```

the query was made on snmpgetnext and the response was the next supported OID

```
unity% ./snmpset.pl 66.26.37.246 private ipNetToMediaType.1.192.168.1.1 2
*ipNetToMediaType.1.192.168.1.1: 2
```

the query was made to delete the arp entry for the 192.168.1.1 ip address

```
unity% ./snmpwalk.pl 66.26.37.246 public ip
[snmpwalk client] Sending queries to 66.26.37.246:160
*ipNetToMediaIfIndex.1.192.168.1.1: 1
*ipNetToMediaPhysAddress.1.192.168.1.1: 0:4:5a:e1:2c:ed
*ipNetToMediaNetAddress.1.192.168.1.1: 192.168.1.1
*ipNetToMediaType.1.192.168.1.1: 1
```

the above query was made on snmpwalk on the ip table. (the supported tree in this table is ipNetToMediaTable)

Low Level Design

[4]Following the flow-chart below and the snmp_main.c file, basically you would see that each ovalshaped object above was a separate developed API, with distinct input arguments and return values.

The main() part of the code (in snmp_main.c) basically binds and listens on the specified UDP socket. And goes into an endless while (1) loop.

Recvfrom() is used due to the connectionless nature of the protocol. Thus recvfrom blocks until data is received on the port.

It is promptly passed to RequestDecoder[7] (in request_decoder.c) which populates the decoded_packet_struct (**dps**) data structure. The motivation behind this was so that the packet fields are all depicted in a data structure for ease of use.

This **dps** is passed to QuerySanityChecker (query_sanity_checker.c) which checks for and corelates the three things:

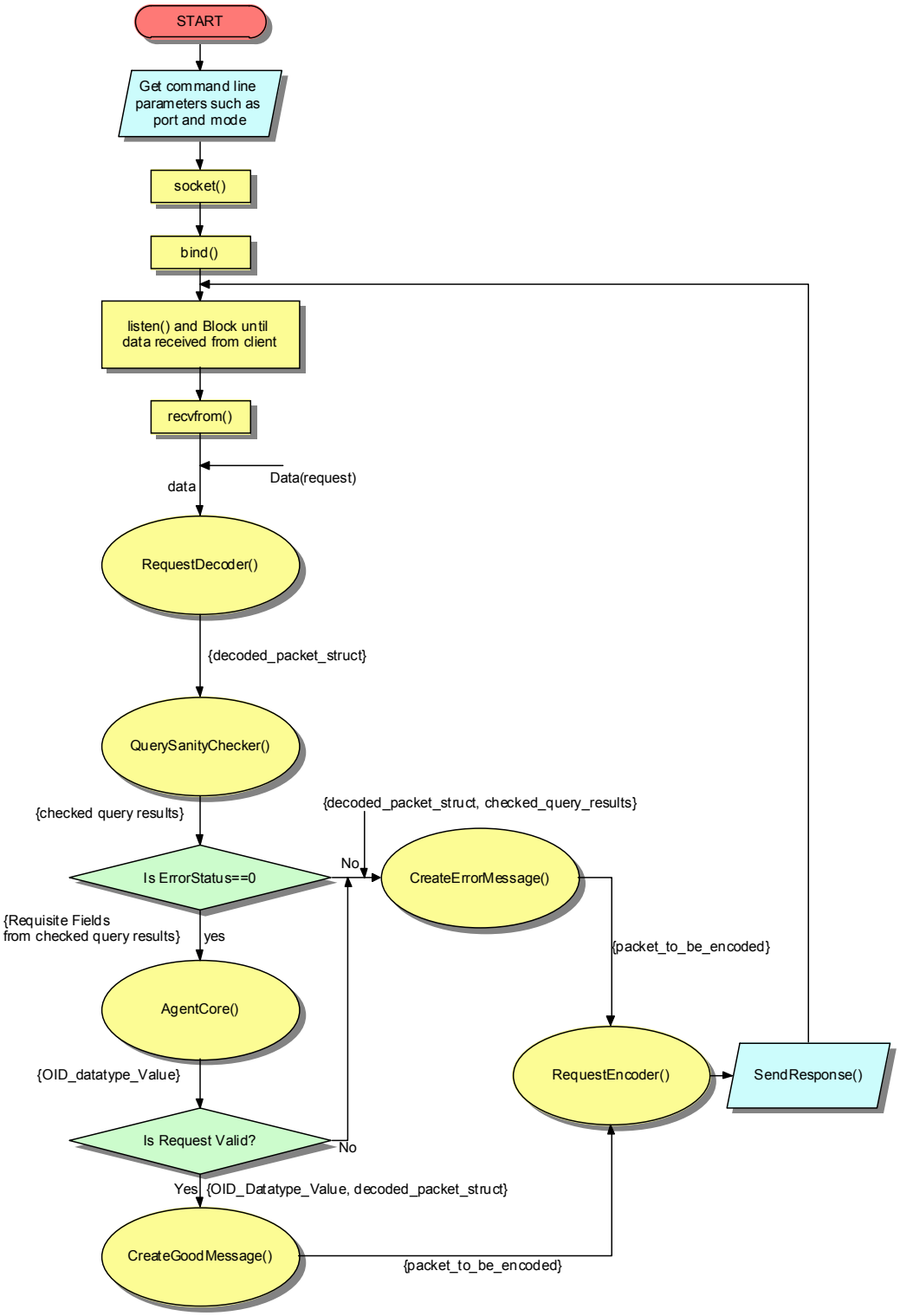
(a) version of snmp (VersionChecker())

(b) community strings and respective request type (get/set/getnext) Community_and_PDU_type_Checker())

(c) the sanity of the OID. (OidChecker())

[4] All references made to data structures here are explicitly defined in lib/snmpd_types.h

Flowchart for SNMP Agent



It in return populates the checked_query_result data struct. (**cqr)**

Depending on the outcome of the above, the error_status of the data struct is duly populated:
In (a) we are expecting SNMPV1 (0) and anything else would be an error
In (b) only the ReadWrite community string can be used for snmpset type queries,while the ReadOnly can be used for both the set and get type queries. Incorrect community string specified was be an error
In (c) various things are checked in the OID string to verify if it is the right OID. Further more for getnext type requests with very small oids, the next plausible OID is also filled in the data struct[5]

Depending on whether the error_status in **cqr** is zero or non-zero the flow proceeds. If there is an error (non-zero) the CreateErrorMessage (create_error_msg.c) fn is called, which uses **cqr** and **dps** to populate the data struct packet_to_be_encoded (**p2be)**.

If there is no error (error_status = 0), some pertinent fields from **cqr** are passed to AgentCore (agent_core.c). This is the heart of the agent.
Depending on the query, AgentCore in turn calls the following functions to carry out the get/set/getnext request:[6]
SystemFn() (systemfn.c) [to handle queries belonging to the system branch]
IpFn() (ipfn.c) [for the ipNetToMediaTable branch]
IcmpFn() (icmpfn.c) , and
TcpFn() (tcpfn.c) [for the tcpConnTable branch]

Though we encourage the reader to look at the code to understand the above functions completely, here is the jist. All the above functions either read from the special /proc file system. The above functions see what type of a query it is: get and set always have to specify the complete OID; the exception is made for getNext requests. For this, the agent needs to know in advance what the next request is at the reception of the query. For this, we extract the respective data from respective /proc file and populate a doubly linked list. This helps in traversing through the LL with ease to get to the right data values and returning it.
Since the /proc table is in memory, the above lookups cause negligible seek times.
Important: The idea of having a cache for getnext requests really did not gain anything as the /proc file system is already in memory, so taking data from one part of the memory and putting it

[5] this is reasonably documented in code and so we would not go into details here
[6] Each of the functions is fairly complex, and we have attempted to comment it and code it as cleanly as possible for the reader.

in another, did not necessarily improve performance. Also reading the proc file for each query helps in always serving the freshest data. Thus the cache idea was dropped. As you shall see during implementation, the performance on the snmp agent is pretty good with this implementation.

In return, AgentCore populate the oid_datatype_value struct (**oid_and_data**). This was then passed to CreateGoodMessage (create_good_msg.c) which populates the **p2be**. In the case of discrepancies found by AgentCore (snmpset attempted on wrong OID, or unsupported OID that slipped through QuerySanityChecker), the value in **oid_and_data** is returned as negative, and based on that CreateErrorMessage is called with **cqr** and **dps.**

Eventually, the p2be is then sent to RequestEncoder[7] which makes the data palatable for the client and then the data is sent to the client, and main goes back to the top of the while(1) loop and blocks again on recvfrom().

Testing Approach

The goal was to be able to query the snmp agent from an snmp client on a remote machine. Thus all the test cases revolve around being able to query the supported MIB branches on the server.
The testing was done with two machines. One (unity) was in NCSU network, and the other was behind a NAT firewall on the TimeWarner Network.[8]

The testing was broken down to three basic areas:
Functionality testing: Where all the supported request types (get/set/getnext/walk) were tested, on all the supported[9] MIB branches (as committed to, in milestone 1)[10]
Negative testing: Incorrect arguments, OIDs, community strings etc were tested here to see if the errors were gracefully handled by the SNMP server (as was in compliance to the error types specified in RFC 1157).
Furthermore, recovery of snmp walks was tested when the server was stopped during a walk and restarted...and also querying a non-running server was also tested.

[7] Ideally, p2be would have been passed to the BerEncoder()/BerDecoder() which would create the binary encoded bitstream/or decode the encoded packet respectively, but we haven't done that: see conclusion.
[8] See TestSetup.txt in the TestCases which is turned in with the source code and is in the same tar file as this .doc
[9] see conclusion on why ReadWrite was not supported on tcpConnState
[10] http://www4.ncsu.edu/~osansari/573/m1.html

Scalability testing: walks carried out with multiple snmp client invocations were timed and tested. It was done to see two things:

(a) whether server was able to handle multiple clients at the same time

(b) whether performance was affected or not

Owing to the bandwidth hogging nature of the UDP protocol, this test was done both over the internet as well as on the local LAN and marked differences in results were seen.

All the above test cases have been explicitly documented in the TestCases portion of the source code (in the same tarball)

Conclusions:

All in all, this was a great learning experience. Came to understand important SNMP related RFCs, learnt how to handle the lexicographical nature of OIDs, learnt how to query Linux platforms for system / network related real time data.

We are happy to come out supporting what we committed to right from the start (in milestone1) and being able to support all the OIDs in the specified branches (system, ip, icmp, tcp) of the RFC1213 MIB.

We discovered during development that there is no good way to hijack a TCP connection on Linux platform s as there is on BSD. Because of which the TCPConnState was unable to be configured as a ReadWrite variable. We didn't feel bad about this though, as even the renowned net-snmp software[11] is incapable of doing so.

In fact, compared to net-snmp we bring an extra feature which net-snmp does not support: Invalidating an arp entry in the arp cache.

Our strengths also lay in the efficient query processing on the agent side, especially our technique of using doubly link lists for each of our OID values, which enhances our software design.

[11] Http://net-snmp.sourceforge.net

From the network perspective, we saw the bandwidth hogging of UDP in action when the agent was tested over the internet. Tests on local segments were remarkably different[12]. This helped us learn how SNMP is really not a scalable protocol to be used over the internet, unless the server and client have enough intelligence built in the application layer to regulate the UDP traffic flow (which normally contemporary servers/clients dont)

The shortcoming of this project was being unable to implement BER encoding/decoding. The readeraware of the SNMP protocol would realize that implementing this is a project in itself. That is why it was decided early on that we would use the BER encoding/decoding methodology from the CMU code. We came clean about this in milestone2 also. It was seen during development, though that using the CMU code was impossible unless we were to drastically change our development structure to meet it, and thus it was dropped.

That said, we do however have our code under the GPL agreement and have all the hooks otherwise to support a BER interface, and thus we pave way for future CSC573 students to write the BER API and integrate it with gmov-snmp v0.1.

Appendix:

Contributions:

Gaurav –

Worked on query sanity checker,

Contributed to socket api for main

Mike –

Worked on snmpclient API

Contributed to agentcore, main()

Omer-

Worked on agentcore,

Contributed to all the other components

Vishal –

Worked on RequestEncoder/Decoder,

Contibuted to agentcore, main()

[12] see scalability testing

Appendix B:

Test Setup

```
Hardware/Networks required:
==========================
Note, for all the testing two machines were used:

- Unity (SunSolaris Machine hosted at NCSU)
- r2d3       (linux machine running at home)

Network Connectivity:
---------------------

Unity ---(internet) -----(e0)[nat_router](e1)---(wlan0)r2d3

nat_router_e0 host name: r2d4.gotdns.com (66.57.11.74)
nat_router_e1: 192.168.1.1
r2d3_wlan0: 192.168.1.26


.on r2d3 the snmp daemon was started in verbose mode on port 160:
the reason we could not run the daemon on port 161 was because
the internet service being used was from road-runner, and owing
to the latest snmp vulnerability, they have blocked port 161 completely.

.also port forwarding on the nat_router was enabled, so any queries
coming to port 160 were sent to r2d3_wlan0's ip address.

--
r2d3_knail1-> ./snmpd -v -p 160
[gmov-snmpd v0.1] SNMP Daemon Started
Attempting to listen on port 160..Ready
Verbose Mode on..

--
r2d3_knail1-> netstat -an |grep 160
udp        0      0 0.0.0.0:160             0.0.0.0:*
--



Also note that each of the snmp client software (snmpget, snmpset etc) were
modified on unity to query port 160, by changing the following line
in the scripts, e.g.:
snmpgetnext.pl:my $port            = '160';


IMPORTANT NOTE:
the server can ONLY run on linux platforms.
the client is a suite of perl scripts, and can run on any unix machine with
perl, and the Socket.pm installed.

We suggest that you use the scripts as is out of the box.
and for the server, use only "snmpd -v" and let it at the default port 161.

All snmp oid's supported by the snmp daemon abided to RFC 1213.
```

Functionality Testing

```
Hardware/Networks required:
===========================
Note, for all the testing two machines were used:

- Unity (SunSolaris Machine hosted at NCSU)
- r2d3       (linux machine running at home)

Network Connectivity:
---------------------

Unity ---(internet) -----(e0)[nat_router](e1)---(wlan0)r2d3

nat_router_e0 host name: r2d4.gotdns.com (66.57.11.74)
nat_router_e1: 192.168.1.1
r2d3_wlan0: 192.168.1.26


.on r2d3 the snmp daemon was started in verbose mode on port 160:
the reason we could not run the daemon on port 161 was because
the internet service being used was from road-runner, and owing
to the latest snmp vulnerability, they have blocked port 161 completely.

.also port forwarding on the nat_router was enabled, so any queries
coming to port 160 were sent to r2d3_wlan0's ip address.

--
r2d3_knail1-> ./snmpd -v -p 160
[gmov-snmpd v0.1] SNMP Daemon Started
Attempting to listen on port 160..Ready
Verbose Mode on..

--
r2d3_knail1-> netstat -an |grep 160
udp        0      0 0.0.0.0:160             0.0.0.0:*
--



Also note that each of the snmp client software (snmpget, snmpset etc) were
modified on unity to query port 160, by changing the following line
in the scripts, e.g.:
snmpgetnext.pl:my $port            = '160';


IMPORTANT NOTE:
the server can ONLY run on linux platforms.
the client is a suite of perl scripts, and can run on any unix machine with
perl, and the Socket.pm installed.

We suggest that you use the scripts as is out of the box.
and for the server, use only "snmpd -v" and let it at the default port 161.


All snmp oid's supported by the snmp daemon abided to RFC 1213.

Section 1: Functionality testing:
=================================
```

Note: (this pertains only to snmpget/snmpgetnext/snmpset)
in this test case we tested some basic functionality of snmpget on a sample OID in each of the four branches tested. As per our milestone 1 you can feel free to test anyone of OIDs supported.

Case#: Procedure
1. snmpget functionality:
---
run a simple snmpget query with the right arguments:

On an OID in system branch:
---
unity% snmpget.pl r2d4.gotdns.com public sysDescr.0 <--query
*sysDescr.0: r2d3: Linux version 2.4.17 <-----result
unity%

On the ICMP branch:
---
a spot check on another branch, the icmp branch was done:

another device in the internal network PINGed r2d3 four times:

cough_knail1-> ping r2d3
PING r2d3 (192.168.1.26): 56 data bytes
64 bytes from 192.168.1.26: icmp_seq=0 ttl=255 time=7.5 ms
64 bytes from 192.168.1.26: icmp_seq=1 ttl=255 time=4.0 ms
64 bytes from 192.168.1.26: icmp_seq=2 ttl=255 time=4.0 ms
64 bytes from 192.168.1.26: icmp_seq=3 ttl=255 time=3.9 ms

the server would now be polled for incoming icmp echos and outgoing icmp echo responses:

unity% snmpget.pl r2d4.gotdns.com public icmpInEchos.0
*icmpInEchos.0: 4
unity% snmpget.pl r2d4.gotdns.com public icmpOutEchoReps.0
*icmpOutEchoReps.0: 4

IMPORTANT NOTE: the linux kernel 2.4.17 has a bug where it does not populate the icmpOutEchos output in the /proc tables. Because of this, the value for icmpOutEchoReps is always zero.

on the IP branch

(note an snmpGET on this branch a previous knowledge of the ip address is needed so that the OID can be properly made out. We shall assume that we knew the oid)

unity% snmpget.pl r2d4.gotdns.com public ipNetToMediaPhysAddress.1.192.168.1.1
*ipNetToMediaPhysAddress.1.192.168.1.1: 0:4:5a:d2:90:45

this was verified by issuing "arp" on the local server:

r2d3_knail1-> arp

```
Address                  HWtype  HWaddress           Flags Mask
Iface
192.168.1.1              ether   00:04:5A:D2:90:45   C
wlan0

on the TCP branch
----
(this also required foreknowledge of the tcp socket connection)

unity% snmpget.pl r2d4.gotdns.com public
tcpConnState.192.168.1.26.32800.152.1.2.65.22
*tcpConnState.192.168.1.26.32800.152.1.2.65.22: 5
unity%

(the value 5 is "established" as per RFC1213)


2. snmpgetnext functionality:
-----------------------------

Basically issue a valid snmpgetnext query with the right args to
snmpgetnext.pl:

system:
unity% snmpget.pl r2d4.gotdns.com public sysContact.0
*sysContact.0: Mike killah Cho
unity% snmpgetnext.pl r2d4.gotdns.com public sysContact.0 <--test
*sysName.0: r2d3 <--response
unity%


(note how the next OID in the tree was returned)


Also testing on the edge of the system branch:
unity% snmpgetnext.pl r2d4.gotdns.com public sysServices.0
*ipNetToMediaIfIndex.1.192.168.1.1: 1


Note how our snmp daemon correctly rolls over to the next supported subtree
(i.e. the ip branch)


ip:
unity% snmpgetnext.pl r2d4.gotdns.com public
ipNetToMediaPhysAddress.1.192.168.1.25
*ipNetToMediaNetAddress.1.192.168.1.1: 192.168.1.1

unity% snmpgetnext.pl r2d4.gotdns.com public ipNetToMediaType.1.192.168.1.25
*icmpInEchos.0: 4


icmp:
unity% snmpgetnext.pl r2d4.gotdns.com public icmpOutEchos.0
*icmpOutEchoReps.0: 4
unity% snmpgetnext.pl r2d4.gotdns.com public icmpOutEchoReps.0
*tcpConnState.0.0.0.0.515.0.0.0.0.0: 2

tcp:
```

```
unity% snmpgetnext.pl r2d4.gotdns.com public
tcpConnState.192.168.1.26.32769.64.12.28.197.5190
*tcpConnState.192.168.1.26.32797.209.61.196.250.80: 8

(8 is close wait as per rfc1157),
also verifying on server:
r2d3_knail1-> netstat -an |grep 32797
tcp        1      0 192.168.1.26:32797      209.61.196.250:80       CLOSE_WAIT

polling last mib:
unity% snmpgetnext.pl r2d4.gotdns.com public
tcpConnRemPort.192.168.1.26.32800.152.1.2.65.22
*EndOfMIB

(note how we correctly identify that the end of mib has been reached)



3. snmpset functionality:
-------------------------

for the system branch, two OIDs are ReadWritable: sysContact, sysLocation

Testing sysContact for simple set:
unity% ./snmpget.pl r2d4.gotdns.com private sysContact.0
*sysContact.0: Brutus Badboy
unity% ./snmpset.pl r2d4.gotdns.com private sysContact.0 SnmpMaster
*sysContact.0: SnmpMaster
unity% ./snmpget.pl r2d4.gotdns.com private sysContact.0
*sysContact.0: SnmpMaster



Testing sysContact for multiworded set support:

unity% ./snmpget.pl r2d4.gotdns.com private sysContact.0
**sysContact.0: Julius Ceaser
unity% ./snmpset.pl r2d4.gotdns.com private sysContact.0 "Brutus Badboy"
*sysContact.0: Brutus Badboy
unity% ./snmpget.pl r2d4.gotdns.com private sysContact.0
*sysContact.0: Brutus Badboy

(note how "" are required for a value with space in it. This should be fine as
the
snmpset usage covers this)

Similarly testing sysLocation for single and then, multiworded supported set:
unity% ./snmpget.pl r2d4.gotdns.com private sysLocation.0
*sysLocation.0: Carnegie Hall
unity% ./snmpset.pl r2d4.gotdns.com private sysLocation.0 NCSU
*sysLocation.0: NCSU
unity% ./snmpget.pl r2d4.gotdns.com private sysLocation.0
*sysLocation.0: NCSU
unity% ./snmpset.pl r2d4.gotdns.com private sysLocation.0 "NCSU Campus"
*sysLocation.0: NCSU Campus
unity% ./snmpget.pl r2d4.gotdns.com private sysLocation.0
*sysLocation.0: NCSU Campus
```

it was seen that the corresponding etc/syscontact.cfg and etc/syslocation.cfg file
on the server side changed each time the above set happened.

for ip branch, the ipNetToMediaType is RW, using which you can invalidate an arp entry:

```
BEFORE (server):
r2d3_knail1-> arp -n
Address                  HWtype  HWaddress           Flags Mask
Iface
192.168.1.1              ether   00:04:5A:D2:90:45   C
wlan0
192.168.1.25             ether   00:01:02:91:86:78   C
wlan0

unity% snmpset.pl r2d4.gotdns.com private ipNetToMediaType.1.192.168.1.25 2
[snmpSet client] sending query to r2d4.gotdns.com:160
*ipNetToMediaType.1.192.168.1.25: 2
(2 means invalidate the corresponding arp entry)

AFTER (server):
r2d3_knail1-> arp -n
Address                  HWtype  HWaddress           Flags Mask
Iface
192.168.1.1              ether   00:04:5A:D2:90:45   C
wlan0
192.168.1.25                     (incomplete)
wlan0
```

A subsequent sniffer trace was carried out which showed that for r2d3 to communicate
to 192.168.1.25, it had to do an arp request for the IP address again, thus showing that the snmpset had succeeded.

(note, a query on an invalidated arp request on standard snmp daemons (net-snmp)
still returns the arp entry. In that vien, we followed suite in our design)

4. snmpwalk functionality:
---

below is the output of the whole walk of the supported tree:

```
unity% snmpwalk.pl r2d4.gotdns.com public .1 <----QUERY
[snmpwalk client] Sending queries to r2d4.gotdns.com:160
*sysDescr.0: r2d3: Linux version 2.4.17
*sysObjectID.0: 1
*sysUpTime.0: 3965
*sysContact.0: Mike killah Cho
*sysName.0: r2d3
*sysLocation.0: NCSUatech sd
*sysServices.0: 7
*ipNetToMediaIfIndex.1.192.168.1.1: 1
*ipNetToMediaIfIndex.1.192.168.1.25: 1
*ipNetToMediaPhysAddress.1.192.168.1.1: 0:4:5a:d2:90:45
*ipNetToMediaPhysAddress.1.192.168.1.25: 0:1:2:91:86:78
*ipNetToMediaNetAddress.1.192.168.1.1: 192.168.1.1
```

```
*ipNetToMediaNetAddress.1.192.168.1.25: 192.168.1.25
*ipNetToMediaType.1.192.168.1.1: 1
*ipNetToMediaType.1.192.168.1.25: 1
*icmpInEchos.0: 4
*icmpInEchoReps.0: 4
*icmpOutEchos.0: 0
*icmpOutEchoReps.0: 4
*tcpConnState.0.0.0.0.515.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.37.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.9.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.13.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.111.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.113.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.21.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.22.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.25.0.0.0.0.0: 2 (listen)
*tcpConnState.192.168.1.26.32801.192.168.1.25.22: 5 (established)
*tcpConnState.192.168.1.26.32796.193.130.68.195.80: 8 (close_wait)
*tcpConnState.192.168.1.26.32794.193.130.68.195.80: 8 (close_wait)
*tcpConnState.192.168.1.26.32771.161.44.11.166.22: 5 (established)
*tcpConnState.192.168.1.26.32769.64.12.28.197.5190: 5 (established)
*tcpConnState.192.168.1.26.32797.209.61.196.250.80: 8 (close_wait)
*tcpConnState.192.168.1.26.32799.209.61.196.250.80: 8 (close_wait)
*tcpConnState.192.168.1.26.32800.152.1.2.65.22: 5 (established)
*tcpConnLocalAddress.0.0.0.0.515.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.0.0.0.0.37.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.0.0.0.0.9.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.0.0.0.0.13.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.0.0.0.0.111.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.0.0.0.0.113.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.0.0.0.0.21.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.0.0.0.0.22.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.0.0.0.0.25.0.0.0.0.0: 0.0.0.0
*tcpConnLocalAddress.192.168.1.26.32801.192.168.1.25.22: 192.168.1.26
*tcpConnLocalAddress.192.168.1.26.32796.193.130.68.195.80: 192.168.1.26
*tcpConnLocalAddress.192.168.1.26.32794.193.130.68.195.80: 192.168.1.26
*tcpConnLocalAddress.192.168.1.26.32771.161.44.11.166.22: 192.168.1.26
*tcpConnLocalAddress.192.168.1.26.32769.64.12.28.197.5190: 192.168.1.26
*tcpConnLocalAddress.192.168.1.26.32797.209.61.196.250.80: 192.168.1.26
*tcpConnLocalAddress.192.168.1.26.32799.209.61.196.250.80: 192.168.1.26
*tcpConnLocalAddress.192.168.1.26.32800.152.1.2.65.22: 192.168.1.26
*tcpConnLocalPort.0.0.0.0.515.0.0.0.0.0: 515
*tcpConnLocalPort.0.0.0.0.37.0.0.0.0.0: 37
*tcpConnLocalPort.0.0.0.0.9.0.0.0.0.0: 9
*tcpConnLocalPort.0.0.0.0.13.0.0.0.0.0: 13
*tcpConnLocalPort.0.0.0.0.111.0.0.0.0.0: 111
*tcpConnLocalPort.0.0.0.0.113.0.0.0.0.0: 113
*tcpConnLocalPort.0.0.0.0.21.0.0.0.0.0: 21
*tcpConnLocalPort.0.0.0.0.22.0.0.0.0.0: 22
*tcpConnLocalPort.0.0.0.0.25.0.0.0.0.0: 25
*tcpConnLocalPort.192.168.1.26.32801.192.168.1.25.22: 32801
*tcpConnLocalPort.192.168.1.26.32796.193.130.68.195.80: 32796
*tcpConnLocalPort.192.168.1.26.32794.193.130.68.195.80: 32794
*tcpConnLocalPort.192.168.1.26.32771.161.44.11.166.22: 32771
*tcpConnLocalPort.192.168.1.26.32769.64.12.28.197.5190: 32769
*tcpConnLocalPort.192.168.1.26.32797.209.61.196.250.80: 32797
*tcpConnLocalPort.192.168.1.26.32799.209.61.196.250.80: 32799
*tcpConnLocalPort.192.168.1.26.32800.152.1.2.65.22: 32800
*tcpConnRemAddress.0.0.0.0.515.0.0.0.0.0: 0.0.0.0
*tcpConnRemAddress.0.0.0.0.37.0.0.0.0.0: 0.0.0.0
```

```
*tcpConnRemAddress.0.0.0.0.9.0.0.0.0.0: 0.0.0.0
*tcpConnRemAddress.0.0.0.0.13.0.0.0.0.0: 0.0.0.0
*tcpConnRemAddress.0.0.0.0.111.0.0.0.0.0: 0.0.0.0
*tcpConnRemAddress.0.0.0.0.113.0.0.0.0.0: 0.0.0.0
*tcpConnRemAddress.0.0.0.0.21.0.0.0.0.0: 0.0.0.0
*tcpConnRemAddress.0.0.0.0.22.0.0.0.0.0: 0.0.0.0
*tcpConnRemAddress.0.0.0.0.25.0.0.0.0.0: 0.0.0.0
*tcpConnRemAddress.192.168.1.26.32801.192.168.1.25.22: 192.168.1.25
*tcpConnRemAddress.192.168.1.26.32796.193.130.68.195.80: 193.130.68.195
*tcpConnRemAddress.192.168.1.26.32794.193.130.68.195.80: 193.130.68.195
*tcpConnRemAddress.192.168.1.26.32771.161.44.11.166.22: 161.44.11.166
*tcpConnRemAddress.192.168.1.26.32769.64.12.28.197.5190: 64.12.28.197
*tcpConnRemAddress.192.168.1.26.32797.209.61.196.250.80: 209.61.196.250
*tcpConnRemAddress.192.168.1.26.32799.209.61.196.250.80: 209.61.196.250
*tcpConnRemAddress.192.168.1.26.32800.152.1.2.65.22: 152.1.2.65
*tcpConnRemPort.0.0.0.0.515.0.0.0.0.0: 0
*tcpConnRemPort.0.0.0.0.37.0.0.0.0.0: 0
*tcpConnRemPort.0.0.0.0.9.0.0.0.0.0: 0
*tcpConnRemPort.0.0.0.0.13.0.0.0.0.0: 0
*tcpConnRemPort.0.0.0.0.111.0.0.0.0.0: 0
*tcpConnRemPort.0.0.0.0.113.0.0.0.0.0: 0
*tcpConnRemPort.0.0.0.0.21.0.0.0.0.0: 0
*tcpConnRemPort.0.0.0.0.22.0.0.0.0.0: 0
*tcpConnRemPort.0.0.0.0.25.0.0.0.0.0: 0
*tcpConnRemPort.192.168.1.26.32801.192.168.1.25.22: 22
*tcpConnRemPort.192.168.1.26.32796.193.130.68.195.80: 80
*tcpConnRemPort.192.168.1.26.32794.193.130.68.195.80: 80
*tcpConnRemPort.192.168.1.26.32771.161.44.11.166.22: 22
*tcpConnRemPort.192.168.1.26.32769.64.12.28.197.5190: 5190
*tcpConnRemPort.192.168.1.26.32797.209.61.196.250.80: 80
*tcpConnRemPort.192.168.1.26.32799.209.61.196.250.80: 80
*tcpConnRemPort.192.168.1.26.32800.152.1.2.65.22: 22
*EndOfMIB
```

also the netstat -an |grep tcp as well as "arp" was done on the server to verify the above:

```
r2d3_knail1-> netstat -an |grep tcp
tcp        0      0 0.0.0.0:515             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:37              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:9               0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:13              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:113             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:21              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:25              0.0.0.0:*               LISTEN
tcp        0      0 192.168.1.26:32801      192.168.1.25:22         ESTABLISHED
tcp        1      0 192.168.1.26:32796      193.130.68.195:80       CLOSE_WAIT
tcp        1      0 192.168.1.26:32794      193.130.68.195:80       CLOSE_WAIT
tcp        0      0 192.168.1.26:32771      161.44.11.166:22        ESTABLISHED
tcp        0      0 192.168.1.26:32769      64.12.28.197:5190       ESTABLISHED
tcp        1      0 192.168.1.26:32797      209.61.196.250:80       CLOSE_WAIT
tcp        1      0 192.168.1.26:32799      209.61.196.250:80       CLOSE_WAIT
tcp        0      0 192.168.1.26:32800      152.1.2.65:22           ESTABLISHED


r2d3_knail1-> arp
```

```
Address                  HWtype  HWaddress           Flags Mask
Iface
192.168.1.1              ether   00:04:5A:D2:90:45   C
wlan0
192.168.1.25             ether   00:01:02:91:86:78   C
wlan0
------
```

Negative Testing

This is to see if incorrect args / values are handled gracefully by the client and server.

1. Incorrect Arguments at Initialization:
---

client:
incorrect number of args entered on the client:
r2d4_knail1-> snmpget.pl ssf sds
snmpGet: [gmov-snmp v0.1]
usage: snmpget.pl <host> <commstring> <oid>
<host>   : the snmp agent you want to poll
<commstring>   : the community string you want to use
<oid>   : the OID you want to poll, can be textual also

(the same output is seen for all the other pl scripts)

server:
r2d3_knail1-> ./snmpd -d
./snmpd: invalid option -- d
[gmov-snmp v0.1] usage:
snmpd [-h] [-p <portnumber>] [-v]
 -p which UDP port to listen on (default 161)
 -v to turn verbosity on
 -h to print (this) help msg

2. Incorrect Community Strings used:

---incorrect commstring tried with get requests:

unity% snmpget.pl r2d4.gotdns.com publasdic sysDescr.0
*Error: No Such Name

(an error is correctly reported)
(note: both RW and RO comm strings would work for ReadOnly requests, as per RFC1157)

public Community string attempted while Setting:
client:
unity% snmpset.pl r2d4.gotdns.com public ipNetToMediaType.1.192.168.1.25 2
[snmpSet client] sending query to r2d4.gotdns.com:160
*Error: No Such Name

(this is the standard err message observed on other RFC compliant snmp agents also)

3. Incomplete OIDs polled
---
note: save for snmpwalk and snmpgetnext, the other queries (snmpget/snmpset) require a complete OID, or they would not work.

Thus incomplete oids are tried for snmpset and snmpget:

unity% snmpget.pl r2d4.gotdns.com publasdic sysDescr

```
*Error: No Such Name
unity% snmpget.pl r2d4.gotdns.com publasdic .1.3.6.1.2.1.1
*Error: No Such Name


(you can try this with snmpget and set for any supported oid and expect
the same results)

4. Incorrect OIDs polled:
-------------------------
unity% snmpget.pl r2d4.gotdns.com publasdic .1.45.76.53
*Error: No Such Name

unity% snmpget.pl r2d4.gotdns.com publasdic doodleyOID
Error: Bad OID


(note, for name OIDs, there is a local name->OID resolution on the client,
because of which we can have a more customized error message as seen above)
however in the case of syntactically correct OIDs as the first one, the query
is sent out and the error message from the server is received and computed.


5. Read Only oid attempted to be set:
-------------------------------------
unity% snmpset.pl r2d4.gotdns.com private sysDescr.0
[snmpSet client] sending query to r2d4.gotdns.com:160
*Error: Read Only

(any RO OID can be tested and the above result can be expected)


6. Bad value set on read write query:
-------------------------------------
unity% snmpset.pl r2d4.gotdns.com private ipNetToMediaType.1.192.168.1.25 1
[snmpSet client] sending query to r2d4.gotdns.com:160
*Error: Bad Value

(as per RFC 1213, ipNetToMediaType can only be set to "2")


7. polling a server which is not up:
------------------------------------

the server was shutdown.
then the client was used to poll the server which was no longer listening on
160.

cough_knail1-> ./snmpget.pl  r2d3 public sysDescr.0
*Connection refused at ./snmpget.pl line 85.

the above is a direct correlation when recv() returns -1. on the wire, it was
seen
that the only time this would happen is if an icmp port/host unreachable is
received shorly after the data is sent.

       die "$!"
      if ( !( recv( SOCK_SENDER, $query_response, 1024, 0 ) ) );

(this had to be tested internally, as my nat router did not send out the icmp
```

port unreachable packets sent out by my server)

8. Failure Testing:
---

this is to show how the if the server would become unreachable network wise (
even with the service up) what would happen:

the server was running...

the client started polling:
r2d4_knail1-> ./snmpwalk.pl r2d3 private tcp
[snmpwalk client] Sending queries to r2d3:161
*tcpConnState.0.0.0.0.515.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.37.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.9.0.0.0.0.0: 2 (listen)
(here the server's network cable was pulled out)
****Timed out

(each star denotes a retry. after 5 retries, the client gives up)

9. Failure Recovery Testing:
---

this is to show the connectionless nature of UDP.

an snmpwalk was started:
r2d4_knail1-> ./snmpwalk.pl r2d3 private tcp
[snmpwalk client] Sending queries to r2d3:161
*tcpConnState.0.0.0.0.515.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.37.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.9.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.13.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.111.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.113.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.21.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.22.0.0.0.0.0: 2 (listen)
*tcpConnState.0.0.0.0.25.0.0.0.0.0: 2 (listen)
*tcpConnState.192.168.1.26.32829.192.168.1.25.22: 11 (time_wait)
*tcpConnState.192.168.1.26.32830.192.168.1.25.22: 5 (established)
(here the server was stopped by issuing a ^Z, but within 5 seconds
was restarted)
****tcpConnState.192.168.1.26.32771.161.44.11.166.22: 5 (established)
*tcpConnState.192.168.1.26.32827.216.239.51.100.80: 8 (close_wait)
*tcpConnState.192.168.1.26.32826.216.239.51.100.80: 8 (close_wait)
*tcpConnState.192.168.1.26.32828.216.239.51.100.80: 8 (close_wait)
....
(the walk then completed gracefully.)

The client attempted to send the request to the server (and kept retrying
as it never received any icmp unreachable, during the time the server was
stopped)

And it was seen that the moment the server was restarted, the client continued
where it left off.

Scalability Testing

This was just one test case.

Multiple sessions (4) were opened on unity.

on each shell, an snmpwalk on the whole tree was issued, the output was piped to
a file, and the time noted.

then the output was compared for data integrity.

also the time it took for a query on a busy server was compared to when only 1 session
was querying the server and it was compared:

```
unity% cat sn.sc
snmpwalk.pl r2d4.gotdns.com public tcp

-rwxr-xr-x   1 osansari ncsu          38 Apr 23 22:15 sn.sc
```

two walks were run simultaneously:

```
unity% time sn.sc > walk.2
0.25u 0.13s 4:28.39 0.1%

unity% time sn.sc > walk.1
0.22u 0.16s 4:38.30 0.1%

unity% diff walk.1 walk.2
24d23
< tcpConnLocalAddress.0.0.0.0.9.0.0.0.0.0: 0.0.0.0
```

a solo walk revealed:

```
unity% time sn.sc > walk.solo
0.29u 0.16s 1:51.13 0.4%
```

(all data was sane)

It was seen that indeed walk.1 had one repeated line (above)

A little thought can clarify this:

udp is a bandwidth hogging protocol.

As far as the time lag goes, it is more likely packet drops than the server response time which have caused the time differences.

As far as the extra line goes, that is perfectly understandable. The client sent a request, which hit the server and the response got queued in the path the client timed out and resent the request, which was also responded to byt the server. after a little time, both these response landed with the client. since there are no seq/ack numbers here, both were taken and displayed.

Another test done on the local segment:

from source: 192.168.1.25--> r2d3 (.26)

three clients were fired up and queried the whole table:

the time for all of them was very close:

```
r2d4_knail1-> time sn.sc > output/walk.1
   12.25s real     0.08s user     0.02s system

r2d4_knail1->  time sn.sc > output/walk.3
   14.54s real     0.08s user     0.03s system

r2d4_knail1->  time sn.sc > output/walk.2
   12.69s real     0.06s user     0.06s system
```

also the three outputs were identical.

```
r2d4_knail1-> ls -ltr
total 16
-rw-r--r--    1 root     root          6159 Apr 23 22:55 walk.1
-rw-r--r--    1 root     root          6159 Apr 23 22:56 walk.2
-rw-r--r--    1 root     root          6159 Apr 23 22:56 walk.3
```