SNMP Traffic Measurement and Analysis
Seminar Report for Networks and Distributed Systems Seminar

Matúš Harvan

Spring Semester 2006

Abstract

The Simple Network Management Protocol (SNMP) is widely deployed to monitor, control, and configure network elements. Even though the SNMP technology is well documented and understood, it remains relatively unclear how SNMP is used in practice and what the typical SNMP usage patterns are. This report describes tools developed to analyze SNMP traces in order to develop a better understanding of how SNMP is used in real world production networks.

1 Introduction

The Simple Network Management Protocol (SNMP) is widely deployed to monitor, control, and configure network elements. However, it is not clear which features are being used, how SNMP usage differs in different types of networks or organizations, which information is frequently queried, and what typical SNMP interaction patterns are in real world production networks. Several publications deal with the performance of SNMP in general [7], the impact of SNMPv3 security [4, 3], or the relative performance of SNMP compared to Web Services [9, 8]. Although these papers are useful to better understand the impact of various design decisions, some of them lack experimental evidence or comparisons and are based only on assumed SNMP interaction patterns. There are many speculations about the usage and performance of SNMP in real world production networks, but to date no systematic measurements have been published.

Many authors use the `ifTable` of the `IF-MIB` [6] or the `tcpConnTable` of the `TCP-MIB` [12] as a starting point for their analysis and comparison. Despite the fact that there is no evidence that operations on these tables dominate SNMP traffic, it is even more unclear how these tables are read and which optimizations are done (or not done) by real world applications. It is also unclear what the actual traffic trade-off between periodic polling and more aperiodic data retrieval is. Furthermore, we do not generally understand how much traffic is devoted to standardized MIB objects and how much traffic deals with proprietary MIB objects and whether the operation mix differs between these object classes or between different operational environments.

An effort to collect SNMP traffic traces in order to find answers to some of these questions, tools developed for that purpose and preliminary results are described in [10]. This report deals with some of the more specialized tools developed to further analyze SNMP with focus on functionality developed and work done within the Networks and Distributed Systems seminar.

2 snmpdump

SNMP traces are usually captured using `tcpdump` and stored in `pcap` format. In order to analyze them, a conversion tool called `snmpdump` has already earlier been developed by Jürgen Schönwälder. It reads raw `pcap` files and converts them into XML and CSV formats. These formats are more convenient for further analysis, where simpler programs or scripts can be used. The CSV and XML formats are described in more details in [13]. The formats may change as development progresses. Output in pcap format is not possible with `snmpdump`. Some work on the `snmpdump` program has been done within the scope of the seminar and will be secribed in mre detail.

2.1 XML input

As the XML format retains all information contained in the raw `pcap` trace, it is possible to use also XML format as input to `snmpdump` instead of the `pcap` format. The module for reading XML as input has been developed as part of the seminar.

The `xmlreader` API from the `libxml` library has been used. Compared to other possibilities, this approach resulted in a $O(1)$ memory consumption. Other approaches had memory consumption dependent on the length of the input and hence did not scale well to large traces. It should be noted that traces in the order of 100GB have been collected at one site and as `snmpdump` is expected to cope with such large traces, the `xmlreader` approach was chosen.

The `xmlreader` offers basic XML syntax checking. A user-defined function is called on every XML node (opening, closing, value inside), so the higher level work requiring knowledge of the XML schema for the particular document has to be done outside the `libxml` library. This, however, does not require to keep track of much state information as the names used in the XML schema are unique, allowing to determine where within the document a node or element belongs. Due to not keeping the state information, checking of well-formedness of the document has not been implemented. In other words, the input XML is not verified against the XML schema and hence the behavior of `snmpdump` on non-well-formed input is undefined. However, it is assumed that mostly XML produced by `snmpdump` itself would be used as input, so it should be well-formed. It should be noted that the checking of well-formedness could be implemented without increasing the memory requirements, but has not been considered significant enough to be worth major implementation efforts. The CSV format contains less information than the original `pcap` trace and hence cannot be used as input for `snmpdump`.

2.2 Invalid checksums

It has been noticed that in some traces used for testing during implementation, only response packets have been present in the output of `snmpdump`, in both XML and CSV formats. The `pcap` trace used as input, however, contained both, request and response packets. The reason for leaving out the request packets turned out to be broken UDP checksums on all request packets. It should be noted that `snmpdump` uses the `libnids` library for reassembly of fragmented packets. This library is ignoring packets with incorrect checksums, which explains why these packets have not shown up in the output of `snmpdump`.

The problem with invalid checksums was further investigated. If traffic traces have been captured on Linux systems with checksum offloading on the network card, where traces were captured on one of the communication endpoints (i.e. hosts on which traffic originates) then outgoing UDP packets have been observed to contain invalid checksums. This was caused by the fact that the packet was sent from the kernel to the networking card without calculating the checksum, as the networking card was taking care of calculating the right checksum. The packets, however, have been captured in the kernel rather than after leaving the networking card and hence contained incorrect checksums. A possible

solution is to turn off checksum offloading using the `ethtool` command. This, however, may noticeably increase system load on fast networks. The problem with invalid checksums was not observed on another Linux host using a networking card not capable of checksum offloading.

Another possible solution could be correcting the checksums directly in the `pcap` file. A rather simple C program could be used for that purpose. The disadvantage would be that if there really have been corrupted packets on the network, one would correct also their checksums.

A similar problem with checksums has been observed if the communication takes place completely within a single host, i.e. on the local interface. For this case, no workaround has been found.

2.3 libanon

One of the problems involved with analyzing SNMP is obtaining real world production network traces a researcher could analyze. Traces produced by the researcher himself can be useful for testing the implementation, but do not yield an insight into the usage and behavior of SNMP in large production networks. Management traffic naturally contains sensitive information and hence operators may be reluctant to provide traces. A possible argument in persuasion of a hesitant network operator might be the availability of trace anonymization. To address this issue, `snmpdump` supports filtering out specified values, i.e. removing sensitive information and anonymization, i.e. transformation of values to other values, but not complete removal of values. For filtering out, regular expressions are used to select message fields. For anonymization, a library `libanon` has previously been developed. This library is capable of lexicographical-order-preserving anonymization of the various data types involved in SNMP traces such as integers, octet strings, MAC addresses or IP addresses. For IP addresses, a prefix- and lexicographical-order-preserving anonymization is used [5]. Due to the way SNMP logical operations, such as table retrievals (table walks), were designed to retrieve data in lexicographical order, the anonymization of a trace needs to preserve the lexicographical ordering to enable analysis of logical operations in the anonymized trace. Although the library has already been developed previously, some work has still been done on it within the seminar. It should be noted that the anonymization implemented in `snmpdump` is not yet complete.

The `libanon` library is based on *Crypto-pAN*, a prefix-preserving IP address anonymization scheme. From there, the representation of an IPv4 address as a 32-bit integer was inherited. While the `libanon` library has been integrated into the `snmpdump` tool, the 32-bit integer representing an IPv4 address was replaced by a more standard representation, `in_addr_t`. This, however, is stored in network byte order, whereas the previous representation relied on host byte order. During the initial change this significant difference was overlooked. Only later was the library found to produce incorrect results and it was time consuming to pinpoint the cause. This was fixed and should bring better portability across architectures with different byte ordering. A similar problem occurred and was fixed for IPv6 addresses, where the initial representation was an array of integers in host byte order.

This was replaced by `struct in6_addr`, which uses network byte order.

The problems with the byte ordering led to creation of test cases and their inclusion into the repository. To motivate testing after changes are done to `libanon`, a shell script was written to check the library's compliance with the test cases for IP address anonymization. This is done by comparing output for known input with expected output that is known to be correct. Unfortunately, such approach, is not directly possible for other data types as their anonymization algorithm involves randomness and hence different runs of the algorithm produce different output even for the same input. Testing for these data types would require setting up a more sophisticated testing framework than simply comparing output with known good output.

Another issue that turned out during integration with `snmpdump` was the need for paraphrase support. The IP anonymization algorithm uses the AES encryption algorithm, where an encryption key needs to be set. As no randomness is involved, knowledge of the key could be used to weaken the anonymization. The previous approach was to hard code the key in the source code. As the software is open-source and distributed in source form, every operator or trace provider wishing to anonymize a trace before giving it away would be expected to modify the key in the source. A probable outcome would be that the key would not be changed, effectively defeating the purpose of anonymization. Therefore, `libanon` was changed to provide support for specifying a passphrase instead of the key directly in the source code. The encryption key is then produced from the passphrase using the SHA-1 hashing algorithm. The support for passphrase was used in `snmpdump` to enable the user easily supply his own passphrase on the command line. If no passphrase is supplied, a random key is used.

Further work on the library included writing a man page for the MAC address anonymization functions. Functions for some of the other data types still remain without a man page.

3 Analysis scripts

The `snmpdump` tool takes care of converting traces from mainly `pcap` format to XML or CSV formats. Optionally, sensitive information can be filtered out or anonymized. After the traces are converted, they can be analyzed. Several Perl scripts have been developed for this purpose and will be discussed in more detail. All of these scripts operate on the CSV format. They have been used to analyze the SNMP traces from networking lab at the International University Bremen. Results from this analysis are presented as well.

3.1 Basic statistics

Previously, a Perl script `snmpstats.pl` has been written for basic statistics. It reports how often which SNMP versions, protocol operations have been used, the number of varbinds in packets and message size distributions.

3.2 Object Identifier statistics

The `snmpstats.pl` script was modified to count which OIDs are used in varbinds and how often. While parsing the input, a hash table for each operation is used to relate OIDs to the number of times they have occurred (separately for different protocol operations). After the input parsing is finished, information from hash tables is summarized and used to create a table showing how often was which OID used. Statistics are displayed separately for each protocol operation.

The OIDs, as seen in packets, consist of an OID prefix and a suffix. It makes much more sense to consider only the OID prefixes and aggregate by them. For this purpose, a script called `snmpobjectstat.pl` was written. It uses the OID table from `snmpstats.pl`'s output to produce an aggregation by OID prefixes. In order to gain MIB knowledge, the `smidump` utility with option `-f identifiers` was used to create a file with known object identifiers from various MIB modules. This file is read in by the `snmpobjectstat.pl` script and stored in a hash table. Only following types are used:

- scalar
- column
- notification

The script then looks up the longest matching prefix for each OID from `snmpstats.pl`'s output in the information supplied by `smidump`. The lookup is performed by cutting away the last number from the OID and trying to match a key in the hash table of known OIDs (from `smidump`). This is repeated until either a prefix is matched or the OID only consists of one numeral. The latter means no matching prefix was found for given OID. Afterwards, aggregation is done using the prefixes only. The output is enriched by names of matched OID, to be more useful to humans reading the reports. OIDs, for which no prefix was found are reported as well. One could then try find MIBs for these OIDs and rerun the scripts.

Initially, the OID prefix lookup was performed in `snmpstat.pl` whenever an OID was read from the input. Hence, the same OID was looked up several times. This indeed turned out to slow down the processing significantly. Therefore, complete OIDs are stored in a hash table while `snmpstat.pl` is running and the lookup is then done only once for each OID when `snmpobjectstats.pl` is run.

For the netlab traces, the most used OIDs have been ifType, ifOperStatus, ifDescr, ifAdminStatus, ifInOctets, ifOutOctets, ifHCInOctets, ifHCOutOctets and sysUpTime.

3.3 General purpose OID lookup

The lookup of names for OIDa, which was a side effect of the `snmpobjectstat.pl` script turned out to be very convenient for analysis of OID usage. This has inspired a general purpose script for doing just OID name lookup, the `snmpoidlookup.pl` script. It uses the information from `smidump`,but includes also rows and tables. The script looks for OIDs

in standard input (using the regular expression `/^\d(\.\d+)+$/`). For each OID it finds, the longest prefix matching it is looked up, using the same strategy as implemented in `snmpobjectstat.pl`. Then the matched numerical prefix value is replaced with the name, while the suffix is kept. In this way no information is lost, but for OIDs where at least some MIB knowledge exists, the name is looked up. The modified input is then sent to standard output. Text not looking as OID is passed through unchanged. This is useful for reports of other scripts, where only numerical OID values are contained. These reports can simply be piped through this script to produce more meaningful reports.

One particular problem with the simple replacing is if the string length of the numerical OID prefix and the corresponding name do not match. This leads to white space formatting problems in reports from other scripts. A possible improvement would be to consume/add white space as possible, but so far the formatting problem has not been severe enough to incite implementation of such an improvement.

3.4 Walks

Rather than looking at protocol operations, it may bring more insight to study logical operations. One such operation is table retrieval (aka table traversal, aka table walk). It is described in detail and with examples in [11]. In this report table traversal will be referred to as a (table) walk. A walk is defined as a sequence of get-next/response or get-bulk/response operations. A mixture of get-next and get-bulk requests is not considered a walk. OIDs in the requests for the same varbind index (corresponding to the same table column) are increasing lexicographically and have the same OID prefix. This prefix is obtained from the first request within the walk. A request may have OID lexicographically lower than the one in previous response in case of table holes. A walk is ended if a response contains OID prefix different from the one in the initial request for all varbinds. Some columns may be shorter than others, but the walk ends when we reach the end of the longest column rather than the shortest one. Therefore, the prefix has to be different for all columns.

For SNMP version 1 get-next requests are used, while from version 2c get-bulk requests can be used. With a get-bulk request, a maximum number of repetitions is set in the request. In this way, it may be possible to retrieve several elements from a different table or column. These elements would have a different OID prefix. With max-repetitions greater than one, there could be a significant portion of the reply consisting of such not desired OIDs. These will be referred to as overshoot. Information about these overshoot elements is not desired by the command generator and hence one could argue transmitting such information is useless. As the walk has to be ended by a response with OID out of the prefix, there always is overshoot of at least 1 for properly ended walks. Please note that for get-next walks the overshoot is always trivially 1.

With the presented definition of a walk in mind, the `snmpwalks.pl` scripts was designed. It detects walks and collects statistics about them, such as

- length of the walk in terms of packets

- number of interactions within the walks (number of requests)
- OID prefix(es) starting the walk
- number of non-repeaters and repeaters – these are heuristically determined also for get-next walks
- number of repetitions in responses
- number of varbinds in responses
- overshoot

The sum of number of varbinds in all responses in a walk provides a measure for the amount of information retrieved via a walk. Similarly, the number of repetitions could be used.

Due to incomplete trace files, it may be possible to observe a walk being started, but not being finished. The response packet with the overshoot OIDs could simply be missing. Such a walk would then not be properly closed. As the script tries to match any new packet to an open walk, having too many open walks would hurt performance. Therefore, using a command line switch, timeout value can be set after which open walks for which there was no packet within the specified period would be timed-out, i.e. closed forcibly and put into a separate bin of timed-out walks.

A walk with one iteration only (1 request, 1 response) is considered a degenerated walk. Such walks seem to be produced by scli, where get-next requests are used for retrieving single values. Although this approach may have advantages in the scli case, it is not considered a proper walk in the context of the `snmpwalks.pl` script. Besides general statistics on the number of packets, number of open, closed and timed-out walks,... the script produces several tables. For all tables, only properly closed walks are considered. The first three tables will be referred to as Table 1, 2 and 3. Table 1 shows for each closed walk the type (get-next or get-bulk), number of interactions, repetitors, non-repetitors, cumulative number of repetitions and response varbinds, duration of the walk (time of last packet minus time of first packet) and the overshoot. Table 2 is similar to Table 1, but shows information based on OIDs starting the walks. Each row in the table corresponds to one OID. Table 3 shows information for groups of OIDS as they have been used to start walks (e.g. in case multiple columns of a table are retrieved in parallel). From this table, it is possible to infer which pieces of information were retrieved together, i.e. as a table. The information in each table is sorted by the cumulative number of repetitions so that the "largest" walks would come first. After the tables a histogram of overshoots is produced.

Using a command line switch it is possible to obtain more detailed information about each walk. Furthermore, it is possible to write packets (in CSV format) belonging to each walk into a separate file. Walks are assigned unique names, which make it possible to use information from the summary report produced by the script to look for details within a particular walk, i.e. an example would the walk with the longest duration or the walk with the largest overshoot.

It has been observed that net-snmp's `snmptable` utility starts a get-bulk walk by appending a .0 to the table OID when using protocol version 2c. In order to detect such walks properly and make them fit our definition, the script strips off the trailing .0 (and prints a warning to standard error).

For the netlab trace, 54% of all the packets (2 539 574 out of 4 699 906) belong to properly closed walks. One could hence assume that walks constitute a significant portion of the SNMP traffic, at least in our networking lab. The trace contains a negligible number of open walks. These are probably the result of stopping the capturing process while some walks were still in progress. The largest walk contains 139 iterations, which corresponds to 278 response varbinds for the whole walk. As could be guessed from the numbers, there have been 2 repetitors. The groups of OIDs starting walks with largest number of repetitions (summed up for all walks with this group of OIDs) have been

- ifDescr
- ifType
- ifOperStatus
- ifAdminStatus

As no get-bulk walks are taking place in the networking lab, only trivial overshoots of 1 have been observed.

As further work, the walks script could be able to distinguish between various walking strategies. Such strategies, however, need to be defined first.

3.5 Frequency analysis

It is assumed that SNMP is often used for regular polling of devices. Therefore, the question of looking for regular interval polling has been approached. Clearly, it does not make sense to look at the arrival times of single packets. A logical operation like a table walk performed every five minutes would have a lot of packets exchanged in millisecond intervals and then for 5 minutes there would be quiet. Instead, arrival times of packets starting logical operations would be more useful. For this reason the `snmpwalks.pl` script is capable of putting packets starting logical operations into a separate file. The arrival times of these packets can then be used as timestamps for the beginning of logical operations. Packets starting logical operations are considered to be

- packets starting walks (get-next, get-bulk)
- request packets not belonging to walks (get-request)

The result of the this is basically a timeseries, in which regular patterns are seeked. The first approach was to inspect the time series in frequency domain. As outlined in [1], simple FFT on packet arrival times should already give hints on regularity. However, in

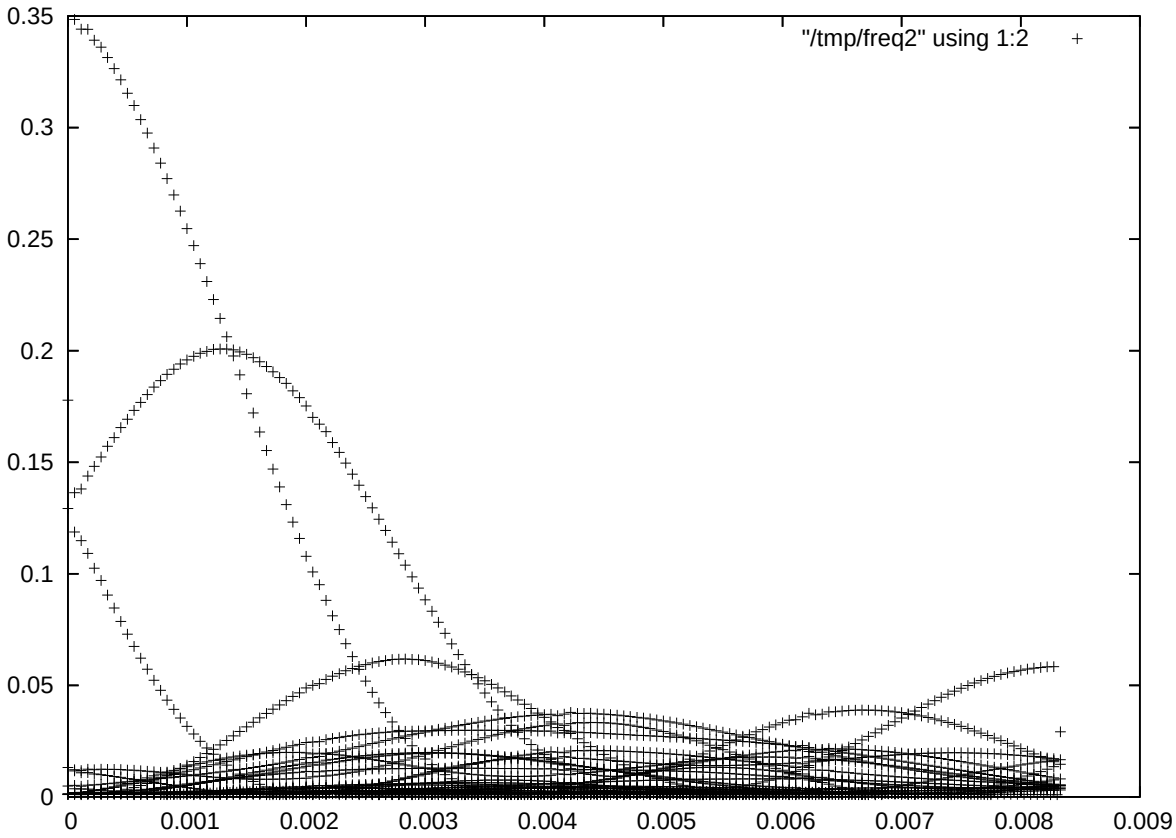


Figure 1: FFT of the start times of logical operations in the trace xen-br-e0

order to be able to do FFT, the time series has to first be sampled at regular time intervals. Afterwards, the GNU Octave was used to compute FFT of the trace. This, however, did not produce useful results. A plot of the FFT can be found in Figure 1. The values for the peaks correspond to approximately

Frequency	Period
0.0013	19.3m
0.00285	8.8m
0.00440	5.7m
0.00665	3.8m

However, there are doubts to the correctness of the frequency values on the x-axis. Furthermore, a simple test case with a packet every 5 seconds produced a completely useless frequency plot. This is probably because taking Fourier Transform of an impulse train is not such a good idea. The values obtained with the FFT as regular intervals for the netlab trace would seem reasonable. Checking the configuration of nagios and cacti in the lab, where this trace was captured, it turned out that the intervals should be 5 or 3 minutes. Nevertheless, the FFT method was not producing reliable results and the reasonably look-

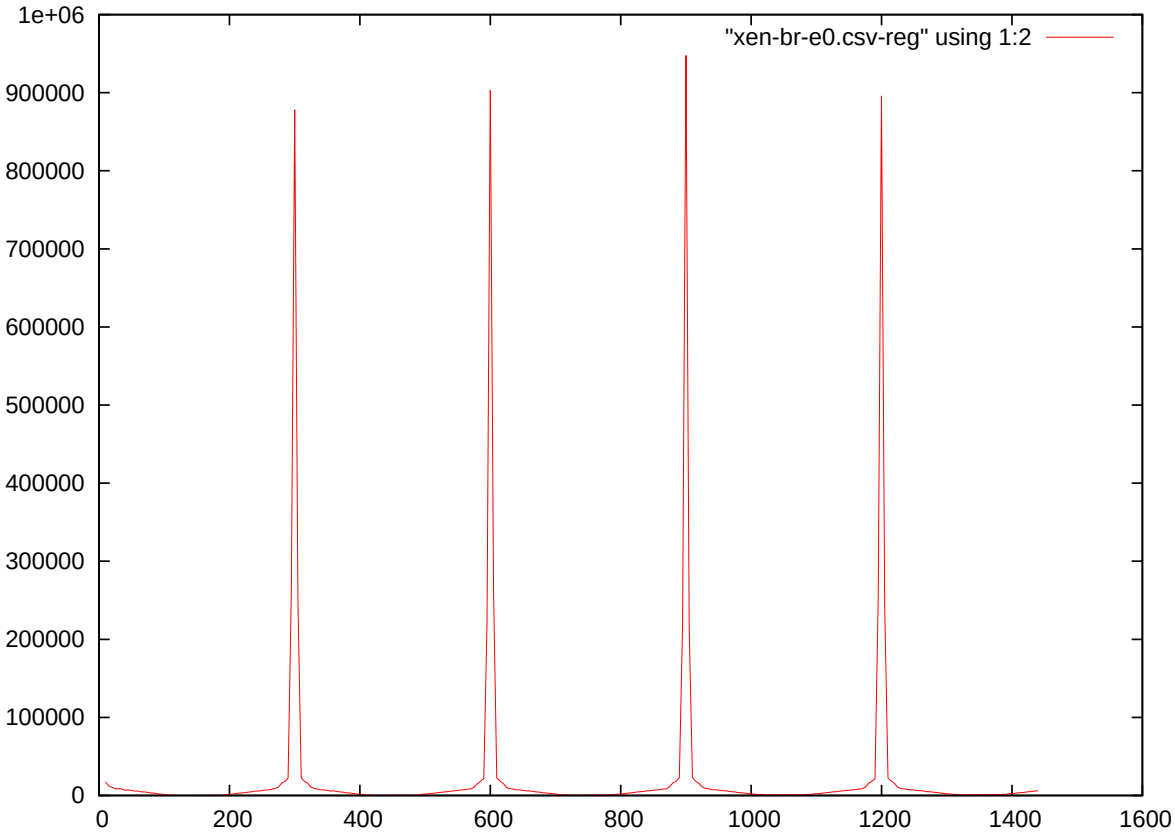
ing values are attributed to coincidence. Besides that, computing the FFT seemed to be rather computationally intensive.

Another paper suggests using Discrete Fourier Transform of the autocorrelation [2]. Therefore, autocorrelation was attempted with GNU Octave. It worked correctly for the simple test case and already seemed to suggest some regular interval pattern for the trace.However, the problem with both methods was that the processing time in GNU Octave was very long. For the autocorrelation, only very small lags were requested to be computed. Otherwise, the computation would run for hours, until it would be killed as probably not leading anywhere in reasonable time. As the autocorrelation results seem reasonable in general and worked for a simple test case with a few packets (as opposed to FFT), the idea of autocorrelation was implemented in yet another Perl script called `snmpreg.pl`. The time series data is sampled with an interval of 5 seconds. A hash table mapping time to number of packets is used. Empty time slots with no packets are not put into the hash table and hence do not occupy memory. For each time slot with packets, it's distance to other time slots with packets is added to another hash table. This hash table maps the distance between time slots with packets to a measure of number of packets in these time slots. At the moment, for every two timeslots distance t apart, the value t maps to is increased by the minimum number of packets in the two time slots. The distance corresponds to a candidate time interval for regular polling. Only distances within the range of 5 seconds to 12 minutes are considered in the analysis script. The reason is that smaller or larger values were not expected for the traces from the lab and limiting the distance enbaled to decrease runtime. The script runs significantly faster than the autocorrelation or FFT approaches with GNU Octave. Furthermore, it seems to produce correct results. In the trace it has found a regular interval of 5 minutes, as can be seen in Figure 2. This corresponds to the configuration of SNMP managers in the networking lab, where the analyzed trace was captured.

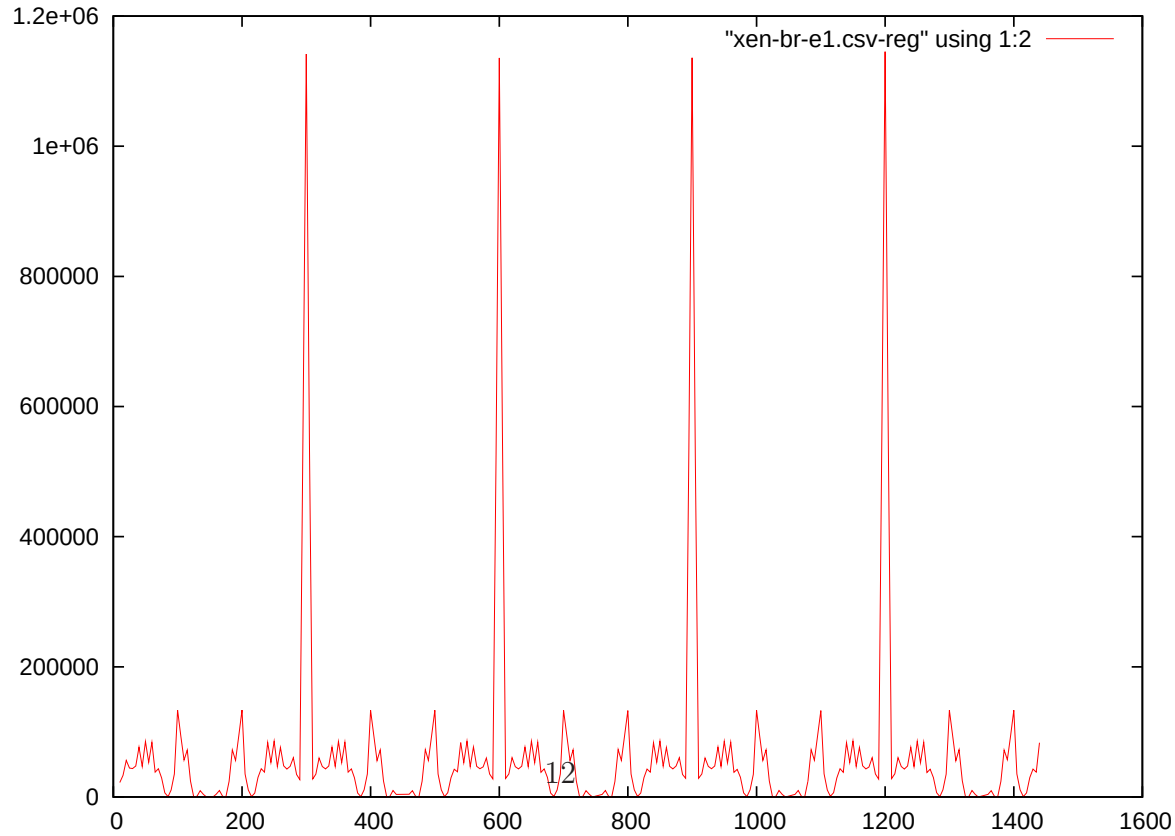
The script, as it is so far, shows that this approach is feasible to analyze regular SNMP traffic. The metric for measuring amount of regularity could be improved to indicate how much of the traffic follows each of the regular intervals. Furthermore, multiples of the fundamental interval could maybe be removed. If the script could detect peaks rather than just produce plots and rely on a human to find the relevant peaks, it could be used for several traces and then summarize results with another script. It may be easier to analyze single flows [10] between managers and agents rather than having them mixed together in one trace. As there could be a huge number of flows in a trace, automating the peak detection would be crucial.

3.6 Runtime

All the Perl scripts presented in the section run in less than 10 minutes on the largest netlab trace, which contains 4 699 906 packets and takes 466 MB in pcap format. Machine on which this was tested has an Intel Xeon 3GHz CPU. The runtime and memory consumption seem acceptable and the scripts are expected to scale well to larger traces.



(a) xen-br-e0



(b) xen-br-e1

Figure 2: Detection of regular intervals between logical operations. The x-axis is the length in seconds of the interval between logical operations. Y'axis measures of how regular this

4 Related Work

Similar analysis scripts, but in Java rather than Perl, are being written by Jorrit Schippers [10].

The performance of SNMP is discussed in several recent papers, like [7, 4, 3, 9, 8]. This work is complementary as it aims at providing empirical data about the usage of SNMP in production networks. Such data is needed to design realistic scenarios and models for evaluating SNMP performance and comparing it to other network management protocols.

5 Conclusion

Several tools for analysis of SNMP traces have been developed or new functionality has been added to existing tools. These improvement include reading XML format as input with the `snmpdump` tool besides the `pcap` format. Furthermore, statistics collection about OIDs, detection and analysis of table walks and regularities in SNMP traces has been enabled. The improvements and new analysis scripts have been tested on traces from the netowrk lab at International University Bremen. Contrary to the conclusion of other reserchers, FFT was not found useful in detection of regularities.

References

[1] Yu Chen, Kai Hwang, and Yu-Kwong Kwok. Filtering of shrew ddos attacks in frequency domain. In *LCN*, pages 786–793, 2005.

[2] Chen-Mou Cheng, H.T. Kung, and Koan-Sin Tan. Use of spectral analysis in defense against dos attacks. In *Proceedings of the IEEE GLOBECOM*, 2002.

[3] A. Corrente and L. Tura. Security performance analysis of snmpv3 with respect to snmpv2c. In *Proc. 2004 IEEE/IFIP Network Operations and Management Symposium*, pages 729–742, Apr. 2004.

[4] X. Du, M. Shayman, and M. Rozenblit. Implementation and performance analysis of snmp on a tls/tcp base. In *Proc. 7th IFIP/IEEE International Symposium on Integrated Network Management*, pages 453–466, May 2001.

[5] Matúš Harvan and Jürgen Schönwaelder. Prefix- and lexicographical-order-preserving ip address anonymization. In *IEEE/IFIP Network Operations and Management Symposium NOMS 2006*, April 2006.

[6] K. McCloghrie and F. Kastenholz. The Interfaces Group MIB. RFC 2863 (Draft Standard), June 2000.

[7] C. Pattinson. A study of the behaviour of the simple network management protocol. In *Proc. 12th IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, Oct. 2001.

[8] G. Pavlou, P. Flegkas, S. Gouveris, and A. Liotta. On management technologies and the potential of web services. 42:58–66, July 2004.

[9] A. Pras, T. Drevers, R. van de Meent, and D. Quartel. Comparing the performance of SNMP and web services based management. 1, Nov. 2004.

[10] Aiko Pras and Jürgen Schönwälder. Snmp traffic analysis. submitted to INM, May 2006.

[11] R. Presuhn. Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP). RFC 3416 (Standard), December 2002.

[12] R. Raghunarayan. Management Information Base for the Transmission Control Protocol (TCP). RFC 4022 (Proposed Standard), March 2005.

[13] J. Schönwälder. SNMP Traffic Measurements. Internet Draft <draft-schoenw-nrmg-snmp-measure-00.txt>, International University Bremen, May 2006.