SNMP GetPrev: An Efficient Way To Browse Large MIB Tables

D. Breitgand[1]
The Hebrew University,
Jerusalem 91904, Israel
{davb}@cs.huji.ac.il

D. Raz[2], *Y. Shavitt*[3]
Bell Labs, 101 Crawfords Corner Road,
Holmdel, NJ 0773, USA
{raz, shavitt}@research.bell-labs.com

Abstract

For some important network management applications, *e.g.*, MIB browsing, there is a need to traverse portions of a MIB tree, especially tables, in both directions. While the GetNext request defined by SNMP standard allows an easy and fast access to the next columnar object instance or the next scalar object, there is no corresponding operator in the SNMP framework for retrieving the previous MIB object instance. This allows an efficient MIB traversal only in one direction and makes the search in the reverse direction problematic.

This paper presents SNMP *GetPrev*, a tool that substantially optimizes retrieval of the previous instances of a columnar objects or scalar MIB objects. Our GetPrev application uses only standard SNMP GetNext and Get requests to carry on a fast and bandwidth efficient search for the required object instance. As we show, our application is two orders of magnitude faster and two to three orders of magnitude less bandwidth consuming when compared to the more traditional approaches.

Keywords:

SNMP, MIB browsing, GetPrev

1 Introduction and Motivation

SNMP is a widely used standard for remote management of devices in IP networks [9, 11, 13, 15]. To retrieve information from the remote SNMP daemon's

[1]This work had been done while the author was a summer intern at Bell Labs.

[2]Danny Raz is currently also with the Dept. of Computer Science, Technion — Israel Institute of Technology, Haifa, Israel.

[3]Yuval Shavitt is currently also with the Dept. of Electrical Engineering – Systems, Tel-Aviv University, Tel Aviv, Israel.

Management Information Base (MIB) a *manager* can use two basic SNMP requests: Get and GetNext. A Get(x) request retrieves the value of the object instance identified by the given OID x. GetNext(x) request retrieves a value of the object instance whose OID is immediate lexicographical successor of the given OID approximation x in the MIB tree.

While Get and GetNext are perfectly adequate for retrieving scalars, using them to retrieve large tables may be cumbersome and inefficient (see a discussion in [14]). To this end a special GetBulk request was introduced in SNMPv2 [3]. GetBulk facilitates the retrieval of large blocks of data and is used primarily to retrieve large tables. In a sense it is a generalization of the GetNext request. Conceptually, GetBulk can be viewed as n GetNext requests packed into a single PDU, where n being a parameter controlled by a manager.

In many cases one does not need to retrieve a whole portion of a MIB tree, *e.g.,* a table, sequentially, but rather is interested in a "random" access to the objects in the MIB tree. This case is examplified by the MIB browsers. A MIB browser is an indispensable management tool that allows the interactive browsing of the content of a MIB using an advanced GUI. Many MIB browser implementations are available today [1, 10, 7]. In order to implement such a browser, one has to be able to efficiently respond to the user requests and retrieve the ever changing information regarding the next and previous objects instances with respect to the current position of a user in the MIB tree. In other words, the MIB tree should be efficiently traversed in both directions in real time. Most browsers today will download a "chunk" of the MIB tree and then perform a local browsing. This creates two problems: (1) if the information in the MIB is changing at a relatively high rate, the information displayed to the user may be out of date, and (2) in many cases the browser may download unnecessary information (*e.g.*, when GetBulk is used), which both wastes bandwidth and slows down the browser.

The following options are available in order to support the retrieval of the previous MIB object instances.

- One may add a GetPrev request to the SNMP standard. Once the agent supports such a request, the retrieval of the previous object will become fast and efficient. The problem, of course, is that changing the standard is highly undesirable. Even if this would happen the multitude of the legacy agents would not support the new version.

- One can redefine large tables in such a way that every object instance contains a pointer to the previous object. Although such mechanism could be used when new MIB modules are designed, it is inapplicable to the well established MIB definitions *e.g.*, those of MIB II [8].

- One may "walk" the table, using a sequence of the GetNext requests until the required OID is retrieved. If we are required to find the value of the last but one entry in the table of size n, this method loads the device with

n queries and takes a time of n round trip delays plus n query processing delays. RFC 1187 [12] suggests to divide the table among a number of threads each one walking its part of a table concurrently with the other threads. This scheme potentialy reduces the retrieval time by a constant factor at the price of overloading the SNMP agent and the management station. The issue of evenly partitioninig the table among the threads is not addressed there.

- One may use GetBulk for greater efficiency. Alas, here we are faced with a problem of choosing the right block size for the GetBulk request. If the block size is too large, one may retrieve information which is outside of the table, thus wasting bandwidth. If the block size is too small we will only improve the previously mentioned "walking" method by a small factor. Last, but not the least, this method is not applicable to the legacy SNMP daemons that are still running SNMPv1.

In the latter two options, we may be required to retrieve the entire table although we are interested only in a single object instance. This is inefficient in several aspects: it takes a longer time, it consumes more bandwidth, and it overloads both the managed device and the manager. For example, retrieving a forwarding table with 3000 entries from a Cisco 7500 router over the 100BaseT Ethernet can take 5-7 minutes using *snmptable* application included with the UCD SNMP library [10], which is definitely too time consuming.

Our goal is to address this problem and allow efficient MIB browsing in real time. To do so, we introduce a simple *GetPrev* application, a tool that substantially optimizes retrieval of the previous MIB object instances. The presented GetPrev application uses only standard SNMP GetNext and Get requests (thus being applicable to any SNMP version) to carry on a fast and bandwidth efficient search for the required object instance.

The main novel idea proposed by this paper is to use binary search rather than linear search in order to make a fast progress in a table starting with the first instance of the columnar object. Of course, this basic idea had to be modified in order to build an efficient tool. This resulted in a number of algorithm variants with different time/bandwidth tradeoffs. Note that our algorithm is an interactive process and may be affected by changes in the table during its execution. As a result, the algorithm may return a stale value. This problem exists in all the other alternative methods, *e.g.,* due to changes that may occur immediatedly after the table was dumped.

As our experiments show, GetPrev retrieves a value of the last columnar object instance in a large forwarding table (ipForwardTable) containing about 3000 entries in less than a second, which is 500 times faster than the straightforward approach that would use a sequence of GetNextnessages. It also achieves a factor of about 400 in the amount of data transmitted over the network during the search. As mentioned above the usage of GetBulk does not reduce the total bandwidth consumption dramatically (in some cases it can even make it larger). However, one would expect less request messages sent over the network and less processing time. We show that even when compared to GetBulk, the performance of our GetPrev application is orders of magnitude superior not only in terms of bandwidth, but also in terms of time and a number of IP datagrams.

The rest of the paper is organized as follows. In the next section we define the problem in more rigorous terms. Section 3 discusses the algorithms used for GetPrev implementation. We evaluate the performance of the new tool in Section 4, and provide some concluding remarks in Section 5.

2 Problem Definition and Model

In this section we define the problem using a rigorous model. We abstract the MIB objects and the communication between a manager and an SNMP agent by the following data base model. All management variables are organized into a labeled tree representing a MIB. A labeled tree $\mathcal{T}$, is a tree in which each node has a label. Two different children of a node must have distinct labels. Following the naming model adopted by SNMP, these labels are non-negative Integer numbers. Each node of the labeled tree is uniquely identified by the ordered sequence of labels of the internal nodes belonging to the path from the root to this node.

Definition 2.1 Identity (or path label) of node n, $nl = l_1.l_2.\ldots.l_n$, is the ordered sequence of labels uniquely identifying the path from the root of $\mathcal{T}$ to this node.

Leaves in the labeled tree have *values* associated with them. The value of node $l_1.l_2.\ldots.l_n$ is given by $\mathcal{V}(l_1.l_2.\ldots.l_n)$. In SNMP terminology, nl is termed the Object Identifier (OID) of a MIB variable, and $\mathcal{V}(ln)$ is referred to as the value of the variable. Each of the labels comprising nl is termed Object Identifier Component (OIC).

Definition 2.2 *SNMP-like data base* is an object comprised of a labeled tree $\mathcal{T}$, and the following methods:

- $Get(l_1.l_2.\ldots.l_n) = \begin{cases} \mathcal{V}(l_1.l_2.\ldots.l_n) & \text{if } l_1.l_2.\ldots.l_n \text{ is a leaf} \\ \text{NULL} & \text{otherwise} \end{cases}$
- $GetNext(l_1.l_2.\ldots.l_n) = (l'_1.l'_2.\ldots.l'_m, \mathcal{V}(l'_1.l'_2.\ldots.l'_m))$, where $l'_1.l'_2.\ldots.l'_m$ is the smallest (in lexicographic ordering) leaf in $\mathcal{T}$ that is lexicographically greater than $l_1.l_2.\ldots.l_n$. If such a leaf does not exist then $l_1.l_2.\ldots.l_n \equiv EndOfTree$ and $GetNext(l_1.l_2.\ldots.l_n) = NULL$.

Note that the methods can be called with any value, that is $l_1.l_2.\ldots.l_n$ can be any label sequence and does not need to represent an existing node in the tree. In this case $l_1.l_2.\ldots.l_n$ is termed *identity approximation.*

The above formalism can also represent a table with a lexicographically sorted key list. In this case the identity of the leaves represent the values of the keys of the table entries. *Get* retrieves the value of a table entry given the key values, and *GetNext* retrieve the key values of the next element in the table along with its value.

Given a label sequence (OID), identifying a leaf in the tree, our aim is to find the previous (in the lexicographic ordering) leaf in this tree. In other words, we want to implement the following operator:

Definition 2.3 $GetPrev(l_1.l_2.\ldots.l_n) = (l'_1.l'_2.\ldots.l'_m, \mathcal{V}(l'_1.l'_2.\ldots.l'_m))$, where $l'_1.l'_2.\ldots.l'_m$ is the largest (in lexicographic ordering) leaf in $\mathcal{T}$ that is smaller than $l_1.l_2.\ldots.l_n$. If such a leaf does not exist then $l_1.l_2.\ldots.l_n \equiv beginningOfTree$ and $GetPrev(l_1.l_2.\ldots.l_n) = NULL$.

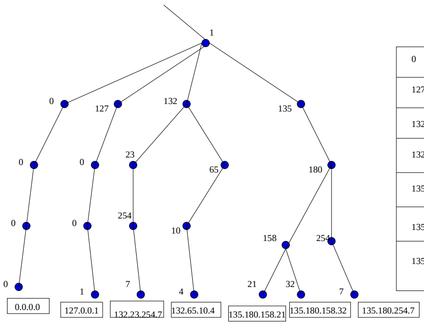
Given the above definitions, the problem is expressed as follows. We need to implement the *GetPrev* method using the least number of calls to the *Get* and *GetNext* SNMP-like data base operators.

Let us call the identity returned by *GetPrev* the *target leaf.* In order to implement the *GetPrev* operator we need to find the target leaf first. In other words, we have to find the path label lexicographically preceding the path label specified as *GetPrev*'s input. Once found, target leaf uniquely identifies the previous element, and its value is retrieved using *Get*.

The first step in finding the target leaf is to find the common ancestor of the given path label and the target leaf. The common ancestor is the last node (from the root) on the path from the root to the given node that has a descendant leaf whose label is lexicographically smaller than that of the input path label. In other words, we are looking for the first node from the input node upward, for which the label of the *GetNext* operator is strictly smaller than the input. We call this node the *maximal common prefix.* For example, in Figure 1 the maximal common prefix of the leaf `.1.135.180.158.21` is the node `.1`, and the maximal common prefix for the label `.1.132.65.10.4` is `.1.132.`

3 Implementations of the GetPrev Algorithm

This section presents the algorithms used by the GetPrev application and reports the status of its implementation. All of the algorithms utilize two basic building blocks: a variant of the well-known binary search algorithm, and an *upper bound* algorithm based on well-known techniques for competitive algorithms [2]). The latter is used to obtain an upper bound to the searched value when one is not known.



0	0	0	0	0.0.0.0
127	0	0	1	127.0.0.1
132	23	254	7	132.23.254.7
132	65	10	4	132.65.10.4
135	180	158	21	135.180.158.21
135	180	158	32	135.180.158.32
135	180	254	7	135.180.254.7

Figure 1: An example for the tree structure of a table

As explained in Section 2, we are looking for an identity of an instance of the columnar object that lexicographically precedes the identity of the given OID. If this OID does not correspond to an existing instance of the columnar object (validated using Get), we replace the given OID by the identity of the instance returned by a GetNext request.

In order to illustrate the problem and the difficulties associated with solving it, consider the table presented in the right hand side of Figure 1. This is a part of the ipRoutingTable from MIB II.[4] The corresponding labeled tree is presented in the left hand side of Figure 1. Leaves of the tree are ordered (from left to right) in the ascending order of the values of their indices. Suppose that we want to find the value of the entry preceding `135.180.158.21.` The full OID of this columnar object instance is: `1.3.6.1.2.1.4.21.1.1.135.180.158.21` (or `ip.ipRoutingTable.1.1.135.180.158.21`). By parsing the MIB one can find the table OID and limit the search to the indices that follow the table prefix (these key fields are shown in Figure 1). The first stage is to find the first component of the preceding entry (132 in our case). We know that the component's value is an Integer number between 0 and 135, and we can carry on a binary search to find it (see Section 3.1.1). After the first component has been determined we turn to finding the second one. In this case we do not have an upper bound on the possible value of this component. Since we know that any OID component is of Integer type, we can use the maximal Integer value as an upper bound.[5] However, a better approach is to try to find a tighter bound on the actual value of the component. We present a simple and fast algorithm that finds an upper bound that is guaranteed to be no greater than twice the actual value of this component. Once the bound is found we use the binary search to determine the wanted component value. In a similar way we find the

[4] Notice that this table is obsolete and is replaced by other tables. However, in many legacy systems this table is still in use, and therefore we used it in our experiments.

[5] In fact, in this specific case we could use additional knowledge about IP addresses, and bound this number by 255. We discuss this point further in Section 3.1.2.

values of the next two components and then perform an SnmpGet request to retrieve the desired value of the columnar object instance.

Next we describe the building blocks mentioned above and the GetPrev algorithm. Then we describe modifications made to the building blocks in order to improve the application's performance. Finally, we report the GetPrev implementation status and the application's availability.

3.1 Building blocks

3.1.1 Binary search

We use binary search in two ways: a classic divide-and-conquer and a search in a sparse domain [5]. As explained before, the first step in finding the predecessor of the given OID is to find the maximal common prefix between this OID and its lexicographic predecessor. In order to do this we use the classic binary search on the length of the prefix. Later we refer to this procedure as $Maximal\_Common\_Prefix$.

Another use of the binary search is to determine values of the *object identifier components* (OICs), following the maximal common prefix. The pseudo-code of the binary search algorithm adapted for this purpose is shown in Figure 2. Here, and later on, we make usage of the two additional functions: *length(oid)* that receives an OID and returns the number of its OICs; and $comp_i(oid)$ that returns the value of i-th component of the OID specified as its argument.

Binary_Search (see Figure 2) receives the following parameters: OID whose lexicographic predecessor should be found, currently known prefix of the predecessor, lower bound of the binary search, upper bound of the binary search. It performs a binary search based on the fact that the wanted value of the component is the largest value v such that $SnmpGetNext(prefix.v)$ is lexicographically smaller than *oid*. Note that the if statement in Line 9 is redundant and is present there just for the sake of clarity.

3.1.2 Upper Bound Algorithm

In many cases the upper bound for the binary search is unknown. As an example, consider tcpConnTable (MIB-II). An instance of a columnar object in this table is identified using four indices: two IP addresses (source and destination), and two ports (source and destination).

Using semantic knowledge of the OICs (in this case, about IP addresses and ports), we would be able to set 255 as the upper bound for the OICs that are part of the IP address and 65536 for the OICs that stand for ports. However, this requires the client to be aware of the semantics of all MIB variables. We prefer to keep the client side as simple as possible and, thus, do not use semantic knowledge.

Algorithm 1 (Binary_Search(*oid*, *prefix*, *low*, *high*))

```
i ← length(prefix) + 1;
while (low < high) {
    mid ← ⌊(low + high)/2⌋;
    if (mid = low)
            mid ← mid + 1;
    response ← SnmpGetNext(prefix.mid);
    if (response ≡ oid)
            high ← mid − 1;
    else if (response < oid)  // lexicographic comparison of OIDs
            low ← comp_i(response);
} // while
return low;
```

Figure 2: Binary Search: determines a value of the OID in the $length(prefix)+1$ position of the OID's predecessor

In absence of the additional knowledge, we find an upper bound for the binary search using the upper bound algorithm. The algorithm (see Figure 3) receives the OID whose predecessor should be found, and a currently known prefix of the predecessor.

It sequentially tries candidates for an upper bound until the *SnmpGetNext* request returns the OID equivalent to that specified as the input. To achieve a tight bound we select as our next candidate value twice the value we received in the response to the *SnmpGetNext*. This promises a bound which is at most twice the actual value present in the table for this OIC. In Section 3.3 we show how to make the bound tighter.

Starting from second OIC after the common prefix, the upper bound algorithm is executed prior to the binary search for every OIC.

3.2 Basic Search Algorithm

Figure 4 presents the pseudocode of the main algorithm used to implement the GetPrev application. The algorithm is comprised of four steps. In the first step (line 1) the algorithm finds the maximal prefix common to the input OID and its predecessor using classic divide-and-conquer search on the length of the prefix. In the next step (line 3) the value of the first OIC following the maximal common prefix is found using binary search algorithm presented in Section 3.1.1. In the third step (line 4) subsequent OICs are determined

Algorithm 2 (Upper_Bound(oid, $prefix$))

1. $i \leftarrow length(prefix) + 1$;
2. $high \leftarrow 1$;
3. $response \leftarrow prefix$;
4. while ($response < oid$) {
5. $\quad response \leftarrow$ SnmpGetNext($prefix.high$);
6. $\quad high \leftarrow 2 * comp_i(response)$;
7. }
8. return $high$;

Figure 3: Upper Bound Algorithm: finding an upper bound for the binary search

using a two phase procedure. First, a tight upper bound for the OIC value is found using the upper bound algorithm (see Section 3.1.2). This bound is fed to the binary search which comprises the second phase. In the fourth step, the algorithm retrieves the value of the variable identified by the found OID of the predecessor.

In the following subsections we present some modifications made to the building blocks used by the main algorithm in order to improve its performance.

3.3 Improved Bounds Algorithm

The Upper Bound algorithm finds an upper bound which is, in the worst case, twice the actual OIC value of the predecessor. It would be beneficial to obtain a tighter upper bound to minimize the number of iterations performed later by the binary search, and thus to conserve both bandwidth and time. In addition, since the upper bound algorithm sequentially tests increasing OIC values until it finds an upper bound, each of the responses to the queries issued by this algorithm constitutes a better lower bound on the OIC value.

Figure 5 presents the pseudo-code of the Improved Bounds Algorithm. It computes two values: a lower bound and an upper bound.

Improved_Bounds works in two phases: *gross bound computation* and *fine bound computation*. The gross bound computation phase is almost identical to the original Upper Bound Algorithm. The only difference is that now the algorithm keeps updating the lower bound. The lower bound is the last response to a $SnmpGetNext$ request being lexicographically less than the input OID.

During the second phase, we start a new series of $SnmpGetNext$ queries at exponentially increasing distances from the lower bound found in the previous phase. For example, if the OIC we are looking for is 3214, and the gross bound

Algorithm 3 (GetPrev(oid))

1. $prefix \leftarrow Maximal\_Common\_Prefix(oid)$;
2. $i \leftarrow length(prefix) + 1$;
3. $first \leftarrow Binary\_Search(oid, prefix, 0, comp_i(oid))$;
4. $prefix \leftarrow prefix.first$;
5. while ($SnmpGetNext(prefix) < oid$) {
6. $\quad high \leftarrow Upper\_Bound(oid, prefix)$;
7. $\quad comp \leftarrow Binary\_Search(oid, prefix, 0, high)$;
8. $\quad prefix \leftarrow prefix.comp$;
9. };
10. return SnmpGet($prefix$);

Figure 4: Basic GetPrev Algorithm: finds the value of a lexicographic predecessor of the given OID

phase finished with the bounds [3200, 6400], the fine bounds phase would test the values 3201, 3203, 3207, 3215, and return [3208,3215] as the bounds for the binary search.[6]

The main motivation for the improvement described in this section is the observation that OICs are not uniformly distributed in the Integer space, but rather tend to concentrate in several blocks of large size.

3.4 Search Algorithm with time-bandwidth tradeoffs

In many cases getting the right answer fast is more important than the bandwidth overhead of the algorithm. Thus one may wish to trade bandwidth for time. For example, in cases where the communication delay dwarfs the message processing time, we may want to reduce the number of messages sent back and forth between the SNMP daemon and GetPrev even further. This can be achieved by asking about more than one OID in every GetNext message.

The number of OIDs to include in a message is a parameter we can tune. If this number is very large, we are again faced with the problems discussed for usage of GetBulk request. We found that using two or three variable bindings per $SnmpGetNext$ message offers a considerably reduceds the number of messages and only slightly increases the bandwidth consumption. For simplicity we continue with a description of the algorithm where three variable bindings are used, but in the same spirit we can use any (small) number of bindings per message.

[6] For simplicity we assume that all the values between 3200 and 3216 indeed exist.

Algorithm 4 (Improved Bounds Algorithm)

```
Improved_Bounds(oid, prefix)

bounds[0] ← 0;
bounds[1] ← 1/2;

phase I: gross bound computation phase:
  while (true) {
      comp ← bounds[1] * 2;
      approx ← prefix.comp;
       response ← SnmpGetNext(approx);
       if (response ≡ oid)
          break;
       bounds[0] ← comp_i(response);
  };
phase II: fine bound computation phase:
  delta = 1;
  bounds[1] = bounds[0] + delta;
  while (true) {
       bounds[1] = bounds[1] + delta;
       approx ← prefix.bounds[1];
       response ← SnmpGetNext(approx);
       if (response ≡ oid) lexicographic comparison of OIDs
          break;
       delta ← delta * 2;
  };
return bounds;
```

Figure 5: Improved Bounds Algorithm: tighter upper and lower bounds for the binary search

In general, the variable bindings are evenly spaced between the current bounds. For the case where the number of bindings per message is three, the known prefix is concatenated with $(low + high)/4$, $(low + high)/2$, and $3 * (low + high)/4$, respectively, where low is the lower bound of an OIC value and $high$ is the upper one. All three approximations are packed into a single SNMP GetNext message PDU and sent to the daemon.

For the case of three variable bindings per message, the modified binary search provides the effect of performing two iterations of the original binary search in just one iteration. Therefore 50% less messages are sent by Binary Search. However, one may notice that exactly one out of every three bindings sent over the network in every $SnmpGetNext$ message would not have been used by the original binary search algorithm. Therefore, the tradeoff between using this modified algorithm and other versions is the tradeoff between sending 50% less GetNext messages where each message is three times larger than in the regular case, and sending more messages where each message is smaller.

3.5 Implementation Status

The GetPrev application has been implemented in the FreeBSD OS environment using the C programming language and ucd-snmp-3.6.2 SNMP library [10]. The application, called *snmpgetprev* (in analogy to *snmpgetnext*) admits a number of options that allow to choose different tradeoffs explained above. As we go to press, the application is in the process of being released by Lucent Technologies, and we plan to make it publicly available in the near future.

4 Performance Evaluation

In this section we study the efficiency of our GetPrev tool. All experiments have been conducted on a 166MHz PentiumII running the FreeBSD operating system. The machine was connected through a Fast Ethernet to a Cisco 7500 router running IOS Version 11.2(21)P, and to a Sun Ultra 10 running Sun SNMP Agent, Ultra-250. Our application was written using ucd-snmp-3.6.2 SNMP library.

We performed two sets of experiments on two different large tables found in MIB-II [8]: tcpConnTable (residing on the Sun machine), and ipRoutingTable (residing on the Cisco router). TcpConnTable had around 500 entries in it throughout the experiment. It took *snmpwalk* application (included with the ucd-3.6.2 library) about 2.5 seconds to download the entire table. In order to evaluate the performance of the algorithm we first downloaded the entire table, then we retrieved the OID's of the elements in position 5,10,15 etc. For each of these OIDs we found the previous value using the following four methods:

- **TableWalk:** A sequence of GetNext requests from the beginning of the table. This is the traditional way to find the previous object.

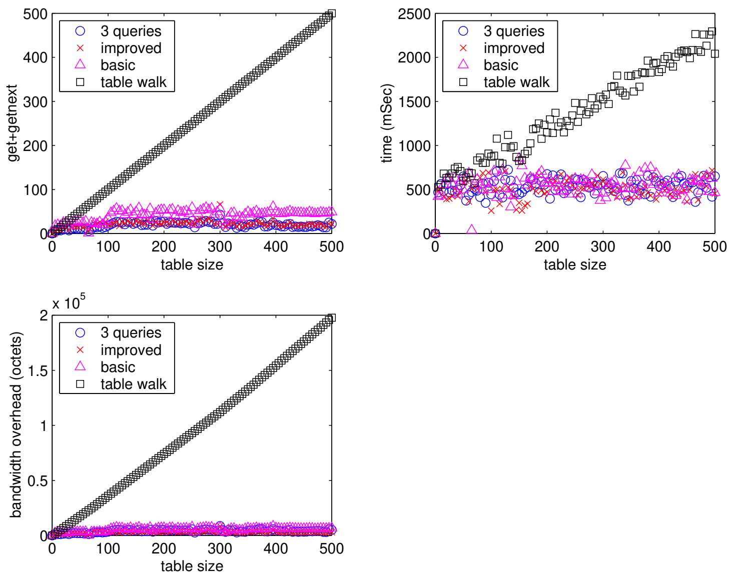


Figure 6: The performance of GetPrev on a TCPConnectionTable

- **GetPrev (Basic):** This is our basic algorithm without the improvements discussed in Section 3.1.2 and Section 3.4.
- **GetPrev (Improved Bounds):** This is our basic algorithm with the improvement discussed in Section 3.1.2.
- **GetPrev (Multi-Query):** This is our algorithm with the time/bandwidth tradeoff improvement discussed in Section 3.4.

For every invocation of each of the above algorithms we have measured the number of SnmpGet and SnmpGetNext messages , the overall time (including all of the local computations, the communication delay , and the agent's response time), and the number of payload octets sent over the network.

The results presented in Figure 6 reflect an average of 10 runs of the above test for each of the four algorithms. It is clear from the figure that the number of Get/GetNext requests, the amount of network traffic, and the time required for the traditional retrieval of the previous value increase linearly with the distance of the object from the beginning of the table. However, the amount of resources used by our tool does not depend on the position of the entry in the table. It stays fairly constant for all elements Moreover, the amount of time it took our tool to get the previous element of the 500th entry in the table was about .65 sec while it took $TableWalk$ method more than 2 seconds to accomplish the same task. In terms of bandwidth usage the result is even more dramatic. Our

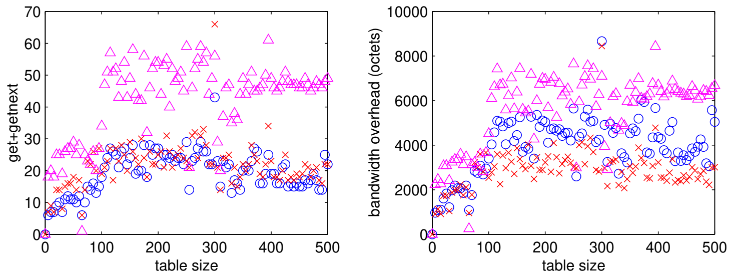


Figure 7: The performance of the different variants of GetPrev on a TCPConnectionTable

tool used 7000 octets while the traditional method used 200000 octets (a factor of 30).

Figure 7 shows a detailed picture of the resources used by the different variants of GetPrev (the legend is the same as in Figure 6). One can see the clear tradeoff between time and bandwidth achieved by the variant with multiple variable bindings per message (blue circles in the figure). This variant uses the least number of Get/GetNext requests, but, in general, it uses more bandwidth compared to other two variants of our GetPrev algorithm. This is hardly surprising, since each packet contains more octets. It is also clear from this figure that the number of Get/GetNext requests, and the total bandwidth usage does not depend on the location of an entry in the table.

An even more impressive improvement has been achieved when we tested the GetPrev tool on Cisco 7500 router's ipForwardTable. The number of entries in this table was around 3000 throughout the experiment, which is not a very large number for such a router. The resource consumption of the different tests is presented in Figure 8. The gap between the linear behavior of the traditional retrieval method and constant requirements of our GetPrev is very clear. In fact, due to this large gap it is impossible to see the data in a linear scale. Thus, we selected a logarithmic scale on the y axis. In fact, if we consider the time parameter for the 2625-th element, we can see that while more than 8 minutes were required for the traditional snmpwalk, our GetPrev accomplished the same task in less than a second, a factor of more than 500.

5 Conclusion

We have presented GetPrev, a new tool that facilitates fast and bandwidth efficient traversal of large SNMP tables in reverse direction. This application may be used as a building block for implementing more efficient and less resource consuming MIB browsers that are widely used by network managers. We have presented the algorithms used to implement GetPrev, and compared

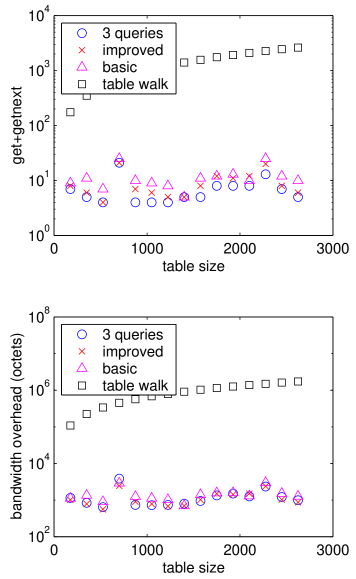


Figure 8: The performance of GetPrev on a ipRoutingTable

the performance of our implementation to other alternatives.

Some other alternatives for implementing an efficient retrieval of the previous instances of MIB objects have not been discussed and have been deferred for future work. In particular, we are interested in investigating the possibility of using AgentX [4] and MIB Script Protocol [6] for implementing GetPrev locally to the SNMP daemon.

Acknowledgments

We would like to thank Subrata Mazumdar for his valuable comments during our work on GetPrev.

References

[1] AdventNet. Adventnet web nms. `http://www.vembu.com/products/snmpbeans/snmpv1/help/uicomponents_mibbro%w`.

[2] Alan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[3] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Protocol operatoins for version 2 of the simple network management protocol (SNMPv2), January 1996. RFC 1905.

[4] M. Daniele, B. Wijnen, M. Ellison, and D. Francisco. IETF RFC 2741: Agent extensibility (AgentX) protocol version 1, jan 2000.

[5] Donald Ervin Knuth. *The Art of Computer Programming: Sorting and Searching (vol 3, Second Edition)*. Addison-Wesley, 1998.

[6] D. Levi and J. Schoenwaelder. IETF RFC 2592: Definitions of managed objects for the delegation of management script, may 1999.

[7] Subrata Mazumdar and Kenneth Swenson. Corba/snmp gateway project. `http://www.bell-labs.com/project/CorbaSnmp/JavaORBImpl/Demo/MibBrowser.%html`.

[8] K. McCloghrie and M. Rose. Management information base for network management of TCP/IP-based internets: MIB-II, March 1991. Internet RFC 1213.

[9] P. E. Miller and Mark A. Miller. *Managing Internetworks with SNMP, Second Edition*. M&T Books, nov 1999.

[10] University of California Davis. Ucd-snmp project home page. `http://net-snmp.sourceforge.net`.

[11] David Perkins and Evan McGinnis. *Understanding MIBs*. Prentice Hall PTR, 1997.

[12] M. Rose, K. McCloghrie, and J. Devin. Bulk table retrieval with SNMP, October 1990. RFC 1187.

[13] Marshall T. Rose and Keith McCloghrie. *How to Manage Your Network using SNMP*. Prentice Hall, January 1995.

[14] Ron Sprenkels and Jean-Philippe Martin-Flatin. Bulk transfers of MIB data. *The Simple Times, The Quarterly Newsletter of SNMP Technology, Comment, and Events*, 7(1):1–8, March 1999.

[15] William Stallings. *SNMP, SNMPV2, SNMPV3, and RMON 1 and 2*. Addison-Wesley, January 1999.

[16] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall Software Series, 1990.