SNMP GetPrev: An Efficient Way to Browse Large MIB Tables

David Breitgand, *Associate Member, IEEE*, Danny Raz, *Member, IEEE*, and Yuval Shavitt, *Senior Member, IEEE*

Invited Paper

***Abstract*—The simple network management protocol (SNMP) is a widely used standard for management of devices in Internet protocol networks. Part of the protocol great success is due to its simplicity; all the managed information is kept in a management information base (MIB) that can be accessed using SNMP queries to a software agent. In this paper, we develop a general model that abstract the data retrieval process in SNMP. In particular, we study the amount of queries (communication) and time needed to randomly access an element in this model. It turns out that this question has practical importance.**

For some 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 next scalar object, there is no corresponding operator defined in the SNMP framework for retrieving the previous MIB object instance. This, in effect, allows an efficient MIB traversal only in one direction and makes the search in the reverse direction problematic.

This paper presents and analyzes the GetPrev application, a tool that enables the 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. For example, as predicted by our analysis and shown by our experiments, retrieving a value of the last columnar object instance in a large forwarding table (ipForwardTable) containing about 3000 entries can take several minutes using a sequence of the GetNext requests (the straightforward approach used, e.g., by widely deployed *snmpwalk* and *snmptable* applications). The GetPrev application presented in this paper retrieves this value using no more than 20 GetNext requests (in most cases about seven requests), taking no more than a second (i.e., it is two orders of magnitude faster and two to three orders of magnitude less bandwidth consuming).

***Index Terms*—GetPrev, MIB browsing, network management, SNMP.**

Manuscript received May 8, 2001; revised December 20, 2001. The work of D. Breitgand, D. Raz and Y. Shavitt was done while they were at Bell Labs, Lucent Technologies, Holmdel, NJ 07733 USA.

D. Breitgand was with Bell Labs, Lucent Technologies, Holmdel, NJ 07733 USA. He is now with the Hebrew University, School of Engineering and Computer Science, Jerusalem, Israel (e-mail: davb@cs.huji.ac.il).

D. Raz was with Bell Labs, Lucent Technologies, Holmdel, NJ 07733 USA. He is now with the Technion, Department of Computer Science, Haifa, Israel (e-mail: danny@cs.technion.ac.il).

Y. Shavitt was with Bell Labs, Lucent Technologies, Holmdel, NJ 07733 USA. He is now with Tel-Aviv University, Department of Electrical Engineering—Systems, Tel-Aviv, Israel (e-mail: shavitt@eng.tau.ac.il).

Publisher Item Identifier S 0733-8716(02)05078-3.

I. Introduction and Motivation

SNMP (simple network management protocol) is a widely used standard for management of devices in Internet protocol (IP) networks [1]–[4]. Part of the protocol great success is due to its simplicity. All the managed information is kept in a management information base (MIB) that can be accessed using SNMP queries to a software agent that is executed on the managed device.[1]

To retrieve information from the remote SNMP daemon's MIB *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 object identifier (OID) x. If the leaf in the MIB tree identified by x does not exist Get returns with an error indicating this. A GetNext(x) request receives an *OID approximation* as its input. The word *approximation* is used in the literature to indicate the fact that the input component sequence x should not necessarily identify an existing leaf in the MIB tree. GetNext always retrieves a value of the object instance whose OID is immediate lexicographical successor of the given OID approximation x in the MIB tree. In other words, GetNext retrieves the value of an object instance with the smallest (in the lexicographic order) OID that is still greater than the given OID approximation. If the given OID approximation identifies the last leaf (in ascending lexicographic order) of the MIB tree, then GetNext returns with an error indicating this condition.

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 [5]). To this end a special GetBulk request was introduced in SNMPv2 [6]. 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 protocol data unit (PDU), where n being a parameter controlled by a manager.

In many cases, one does not need to retrieve an entire table but is interested only in a specific entry. For example, a manager may wish to find out what is the largest port number currently used by transmission control protocol (TCP) at a specific host. Another case that may be considered arises when a manager wishes to see which entry in a routing table precedes a given entry. A more general case is introduced by the MIB browsers.

[1]In order to make our presentation self-contained and clear even to the SNMP novice, we found it appropriate to reiterate some of the SNMP basics in the Appendix.

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 [7]–[9]. 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 $\mathrm{GetBulk}$ 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 $\mathrm{GetPrev}$ 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. It takes a long period of time, and it is not clear whether such a change is unavoidable. 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 (and maybe should) be used when new MIB modules are designed, it is inapplicable to the well established MIB definitions e.g., those of MIB II [10]. Therefore, this option is as troublesome as updating the SNMP standard to support a new type of request.
- One may "walk" the table, using a sequence of the $\mathrm{GetNext}$ 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 [11] 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 potentially reduces the retrieval time by a constant factor at the price of overloading the SNMP agent and the management station. The issue of evenly partition the table among the threads is not addressed there.
- One may use $\mathrm{GetBulk}$ for greater efficiency. Alas, here we are faced with a problem of choosing the right block size for the $\mathrm{GetBulk}$ request. If the block size is too large, one may retrieve information which is outside of the table. If the block size is too small we will only improve the previously mentioned "walking" method by a small factor. It should be noted also that $\mathrm{GetBulk}$ does not reduce the overall amount of data transmitted over the network, but rather reduces the overall number of IP packets sent between the manager and a daemon. 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 five to seven minutes using $snmptable$ application included with the UCD SNMP library [8], 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 $\mathrm{GetPrev}$ application, a tool that substantially optimizes retrieval of the previous MIB object instances. The presented $\mathrm{GetPrev}$ application uses only standard SNMP $\mathrm{GetNext}$ 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. That is, instead of retrieving the lexicographically next object we try to "jump" forward to a larger OID and perform a $\mathrm{GetNext}$ request on it. If we do not end up with a too large OID then we continue, otherwise, we know that the guessed value is too large and we modify the previous guess. 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.

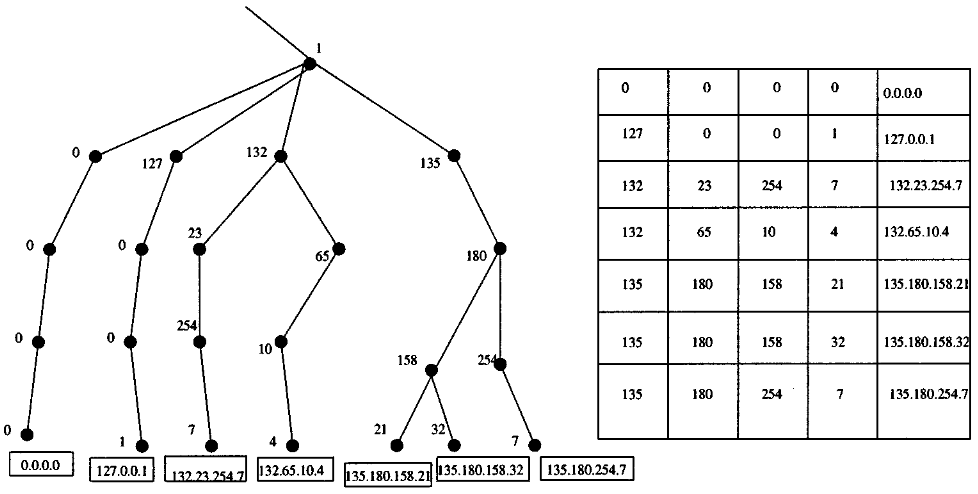
Note that our algorithm is an interactive process and may be effected by changes in the table during its execution. As a result, the algorithm may return a stale value. This problem exists in all the other alternative methods, e.g., due to changes that may occur immediately after the table was dumped.

As our experiments show, $\mathrm{GetPrev}$ 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 used by $snmptable$. 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 $\mathrm{GetBulk}$ does not reduce the total bandwidth consumption (in some cases it even makes it larger). However, one would expect less request messages sent over the network and less processing time. We show that even when compared with $\mathrm{GetBulk}$, the performance of our $\mathrm{GetPrev}$ 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 following. In Section II, we define the problem in more rigorous terms. Section III discusses the algorithms used for $\mathrm{GetPrev}$ implementation. We evaluate the performance of the new tool in Section IV, analyze the performance results in Section V, and provide some concluding remarks in Section VI.

II. Problem Definition and Model

In order to perform a rigorous study of the efficiency of the different possible solutions we turn now to define our model formally. We abstract the MIB objects and the communication



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

Fig. 1. An example for the tree structure of a table.

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 nonnegative 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 1: Identity (or path label) of node n, $nl = l_1.l_2.\dots l_n$, is the ordered sequence of labels uniquely identifying the path from the root of $\mathcal{T}$ to this node. The sequence is interpreted as a path that starts at the root (.) goes to the node labeled l_1, then to the child labeled l_2 and so on.

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

Definition 2: SNMP-like data base is an object comprised of a labeled tree $\mathcal{T}$, and the following methods:

$$\mathrm{Get}(l_1.l_2.\dots l_n) = \begin{cases} \mathcal{V}(l_1.l_2.\dots l_n) & \text{if } l_1.l_2.\dots l_n \text{ is a leaf} \\ \mathrm{NULL} & \text{otherwise} \end{cases}$$

$$\mathrm{GetNext}(l_1.l_2.\dots l_n) = (l'_1.l'_2.\dots l'_m,\ \mathcal{V}(l'_1.l'_2.\dots l'_m))$$

where $l'_1.l'_2.\dots l'_m$ is the smallest (in lexicographic ordering) leaf in $\mathcal{T}$ that is lexicographically greater than $l_1.l_2.\dots l_n$. If such a leaf does not exist then $l_1.l_2.\dots l_n \equiv \mathrm{EndOfTree}$ and $\mathrm{GetNext}(l_1.l_2.\dots l_n) = \mathrm{NULL}$.

Note that the methods can be called with any value, that is $l_1.l_2.\dots l_n$ can be any label sequence and does not need to represent an existing node in the tree. In this case $l_1.l_2.\dots l_n$ 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 $\mathrm{GetNext}$ 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 3: $\mathrm{GetPrev}(l_1.l_2.\dots l_n) = (l'_1.l'_2.\dots l'_m, \mathcal{V}(l'_1.l'_2.\dots l'_m))$, where $l'_1.l'_2.\dots l'_m$ is the largest (in lexicographic ordering) leaf in $\mathcal{T}$ that is smaller than $l_1.l_2.\dots l_n$. If such a leaf does not exist then $l_1.l_2.\dots l_n \equiv \mathrm{beginningOfTree}$ and $\mathrm{GetPrev}(l_1.l_2.\dots l_n) = \mathrm{NULL}$.

Given the above definitions, the problem is expressed as following. We need to implement the $\mathrm{GetPrev}$ method using the least number of calls to the Get and $\mathrm{GetNext}$ SNMP-like data base operators.

Note that by the above definitions if $l_1.l_2.\dots l_n$ is a leaf then $\mathrm{GetNext}(\mathrm{GetPrev}(l_1.l_2.\dots l_n)) = (l_1.l_2.\dots l_n, \mathcal{V}(l_1.l_2.\dots l_n))$. If $l_1.l_2.\dots l_n$ is an internal node then $\mathrm{GetNext}(\mathrm{GetPrev}(l_1.l_2.\dots l_n)) = \mathrm{GetNext}(l_1.l_2.\dots l_n)$.

Let us call the identity returned by $\mathrm{GetPrev}$ the *target leaf.* In order to implement the $\mathrm{GetPrev}$ 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 $\mathrm{GetPrevs}$ 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 $\mathrm{GetNext}$ operator is strictly smaller than the input. We call this node the *maximal common prefix.* For example, in Fig. 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.`

The correspondence between the labeled tree $\mathcal{T}$ and the scalar objects in the MIB is obvious. Instances of columnar objects are lexicographically ordered by the ascending values of their indices. Usually, as explained in the introduction one would apply the GetPrev operator to an instance of a columnar object, looking for the value of the previous element in the table. In such cases either MIB parsing or Syntax parsing of the OID can retrieve the table's OID. The search for the maximal common prefix can then start from this OID, since the table is an existing object.

III. 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 [12]). The latter is used to obtain an upper bound to the searched value when one is not known.

As explained in Section II, we are looking for an identity of an instance of the columnar object that lexicographically precedes the identity of the given OID. In other words, if the given OID identifies a valid instance of a columnar object then we are looking for the instance of the columnar object for which the return value of a GetNext request is the value of the given instance. However, if the given OID does not identify a valid instance of a columnar object then we are looking for the instance of the columnar object for which the returned value of a GetNextrequest is the returned value of a GetNext request for the given instance approximation. Thus, before starting the search algorithm we verify that the given OID identifies an existing instance of a columnar object (using Get). If this is not the case, 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 Fig. 1. This is a part of the ipRoutingTable from MIB II.[2] The corresponding labeled tree is presented in the left hand side of Fig. 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 Fig. 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 III-A-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.[3] 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 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.

```
Binary_Search($oid$, $prefix$, $low$, $high$)
1.  $i \leftarrow length(prefix) + 1$;
2.  while ($low < high$) {
3.      $mid \leftarrow \lfloor (low + high)/2 \rfloor$;
4.      if ($mid = low$)
5.          $mid \leftarrow mid + 1$;
6.      $response \leftarrow$ SnmpGetNext($prefix.mid$);
7.      if ($response \equiv oid$)
8.          $high \leftarrow mid - 1$;
9.      else if ($response < oid$) // lexicographic comparison of OIDs
10.         $low \leftarrow comp_i(response)$;
11. } // while
12. return $low$;
```

Fig. 2. Binary search: determines a value of the OIC in the $\mathrm{length(prefix)} + 1$ position of the OIDs predecessor.

A. Building Blocks

1) Binary Search: We use binary search in two ways: a classic divide-and-conquer and a search in a sparse domain [13]. 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 Fig. 2. Here, and later on, we make usage of the two additional functions: $\mathrm{length(oid)}$ that receives an OID and returns the number of its OICs; and $\mathrm{comp}_i(\mathrm{oid})$ that returns the value of ith component of the OID specified as its argument.

Binary_Search (see Fig. 2) receives the following parameters:

- oid: OID whose lexicographic predecessor should be found
- prefix: currently known prefix of the predecessor
- low: lower bound of the binary search
- high: upper bound of the binary search.

[2]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.

[3]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 III-A-2.

tcpConnState	tcpConnLocalAddress	tcpConnLocalPort	tcpConnRemAddress	tcpConnRemPort
closed	0.0.0.0	0	0.0.0.0	0
listen	0.0.0.0	7	0.0.0.0	0
listen	0.0.0.0	9	0.0.0.0	0
listen	0.0.0.0	13	0.0.0.0	0
listen	0.0.0.0	111	0.0.0.0	0
listen	0.0.0.0	513	0.0.0.0	0
....	....	....	....	...
listen	0.0.0.0	515	0.0.0.0	0
listen	0.0.0.0	13722	0.0.0.0	0
listen	0.0.0.0	13782	0.0.0.0	0
listen	0.0.0.0	32780	0.0.0.0	0
....	....	....	....	...
listen	0.0.0.0	32870	0.0.0.0	0
listen	0.0.0.0	34810	0.0.0.0	0
established	127.0.0.1	32796	127.0.0.1	32872
established	127.0.0.1	32870	127.0.0.1	32875
established	127.0.0.1	32870	127.0.0.1	32881
established	127.0.0.1	32870	127.0.0.1	32887
established	127.0.0.1	32870	127.0.0.1	32893
established	127.0.0.1	32870	127.0.0.1	33187
....	....	....	....	...
established	127.0.0.1	32890	127.0.0.1	32889
established	127.0.0.1	32893	127.0.0.1	32870
established	127.0.0.1	32895	127.0.0.1	32896
established	127.0.0.1	32896	127.0.0.1	32895
established	127.0.0.1	33187	127.0.0.1	32870
established	127.0.0.1	33189	127.0.0.1	33190
established	135.180.161.15	23	135.180.142.50	1125
established	135.180.161.15	23	135.180.142.50	1126

Fig. 3. An example of a TCP connection table.

It preforms a binary search based on the fact that the wanted value of the component is the largest value v such that $\mathrm{SnmpGetNext}(\mathrm{prefix}.v)$ 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.

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). A sample table is shown by Fig. 3.

If we want to find the value of an OID that lexicographically precedes `135.180.161.15.23.135.180.142.50.1125`, then, obviously, we can use the regular binary search in order to determine the value of the first OIC of the predecessor by having zero as the low and 135 as the high bounds of the search. However, we cannot use this simple scheme for determining a value of the second OIC of the predecessor.

Using semantic knowledge of the OIC (in this case, about IP addresses), we would be able to set 255 as the upper bound for the label's values. However, this require 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.

In absence of the additional knowledge, we find an upper bound for the binary search using the upper bound algorithm. The algorithm (see Fig. 4) 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 III-C, we show how to make the bound tighter.

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

Fig. 4. Upper bound algorithm: finding an upper bound for the binary search.

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

B. Basic Search Algorithm

Fig. 5 presents the pseudocode of the main algorithm used to implement the GetPrev application. The algorithm is comprised of four main 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 III-A-1. In the third step (line 4), subsequent OICs are determined using a two phase procedure. First, a tight upper bound for the OIC value is found using the upper bound algorithm (see

```
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.     $high \leftarrow Upper\_Bound(oid, prefix)$;
7.     $comp \leftarrow Binary\_Search(oid, prefix, 0, high)$;
8.     $prefix \leftarrow prefix.comp$;
9. }
10. return SnmpGet($prefix$);
```

Fig. 5. Basic GetPrev algorithm: finds the value of a lexicographic predecessor of the given OID.

Section III-A-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.

C. 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. For example, suppose we want to find the predecessor of `135.180.161.15.23.135.180.142.50.1125` in tcpConnTable shown in Fig. 3. Upper_Bound will return a value not greater than 66 378 as the upper bound for the fifth OIC of the predecessor (In fact, the upper bound returned for this example will be $32\,780 * 2 = 65\,560$).

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. Not using this information later in the binary search is wasteful.

Fig. 6 presents the pseudo-code of the Improved Bounds Algorithm. The Improved_Bounds function receives the same parameters as Upper_Bound. However, it is modified to return 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 expansion phase finished with the bounds [3200, 6400], the contraction phase would test the values 3201, 3203, 3207, 3215, and return [3208,3215] as the bounds for the binary search.[4]

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

```
Improved_Bounds($oid$, $prefix$)
1. $bounds[0] \leftarrow 0$;
2. $bounds[1] \leftarrow \frac{1}{2}$;
3.
4. phase I: gross bound computation phase:
5.   while ($true$) {
6.       $comp \leftarrow bounds[1] * 2$;
7.       $approx \leftarrow prefix.comp$;
8.       $response \leftarrow$ SnmpGetNext($approx$);
9.       if ($response \equiv oid$)
10.          break;
11.      $bounds[0] \leftarrow comp_i(response)$;
12.  }
13. phase II: fine bound computation phase:
14.   $delta = 1$;
15.   $bounds[1] = bounds[0] + delta$;
16.   while ($true$) {
17.       $bounds[1] = bounds[1] + delta$;
18.       $approx \leftarrow prefix.bounds[1]$;
19.       $response \leftarrow$ SnmpGetNext($approx$);
20.       if ($response \equiv oid$) lexicographic comparison of OIDs
21.          break;
22.       $delta \leftarrow delta * 2$;
23.   }
24. return $bounds$;
```

Fig. 6. Improved bounds algorithm: Tighter upper and low bounds for the binary search.

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. This can be easily seen in the table presented in Fig. 3. The improvement's performance is more rigorously assessed in Section V using the model defined in Section II.

D. Search Algorithm With Time-bandwidth Tradeoffs

In many cases, getting the right answer fast is more important than the 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. As a result, the number of messages is reduced but their size and processing time become larger. This requires to modify the binary search as described below.

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 (see Section IV-A. We found that using two or three variable bindings per SnmpGetNext message offers a considerable improvement in 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.

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 $(\mathrm{low} + \mathrm{high})/4$, $(\mathrm{low} + \mathrm{high})/2$, and $3 * (\mathrm{low} + \mathrm{high})/4$, respectively, where low is the lower bound of an OIC value and

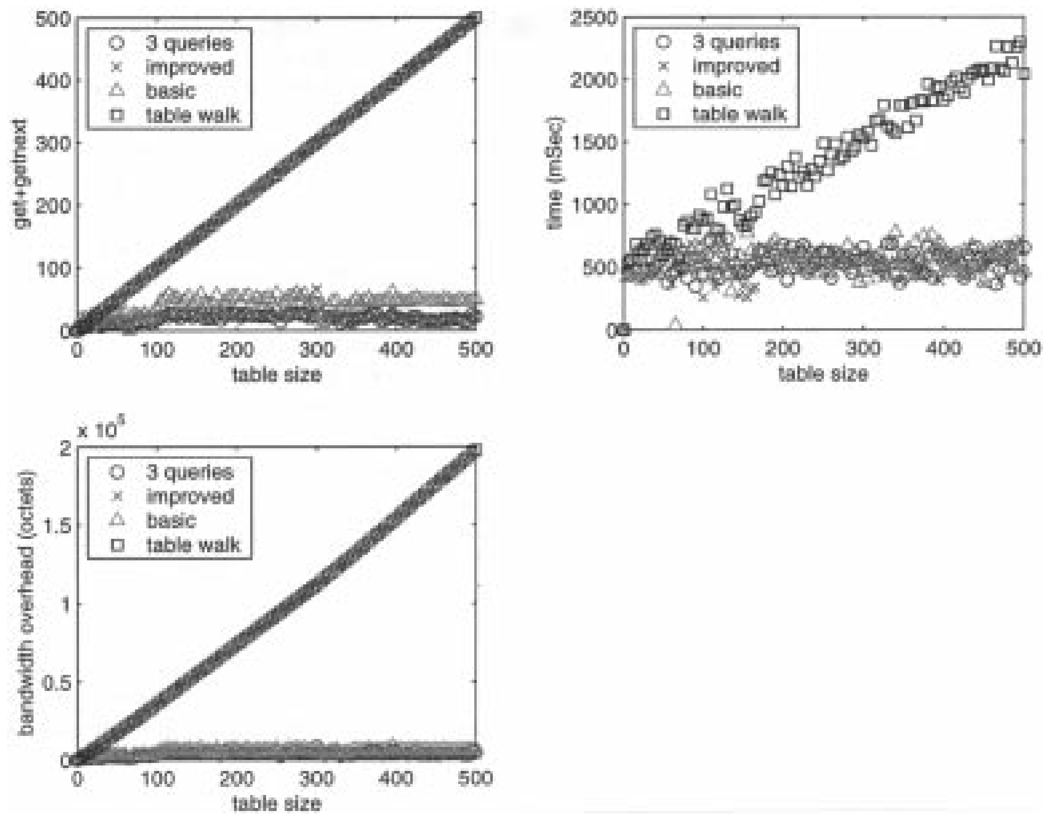


Fig. 7. The performance of GetPrev on a TCPConnectionTable.

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 be 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.

Similar modification can be made to the improved bounds algorithm, by changing it to try more than one upper and lower bound candidate in every GetNext message.

In principle, multiple variable bindings may also be used for the search of the maximal common prefix. However, in practice, this would not provide any considerable benefit since the maximal number of OICs in an OID according to the SNMP standard is limited to 128, and usually OIDs are much shorter than this. Therefore, in the worst case, maximal common prefix can always be found using just seven GetNextmessages.

E. 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 [8]. The application, called snmpgetprev (in analogy to snmpgetnext) admits a number of options that allow to choose different tradeoffs explained above. As we write this paper, the application is in the process of being released by Lucent Technologies, and we plan to make it publicly available in the near future.

IV. Performance Evaluation

In this section, we study the efficiency of our GetPrev tool. All experiments have been conducted on a 166 MHz 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 [10]: 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 s to download the entire table. In order to evaluate the performance of the algorithm, we first downloaded the entire table, then we retrieved the OIDs 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.
- **GetPrev (Basic):** This is our basic algorithm without the improvements discussed in Sections III-A-2 and III-D.
- **GetPrev (Improved Bounds):** This is our basic algorithm with the improvement discussed in Section III-A-2.

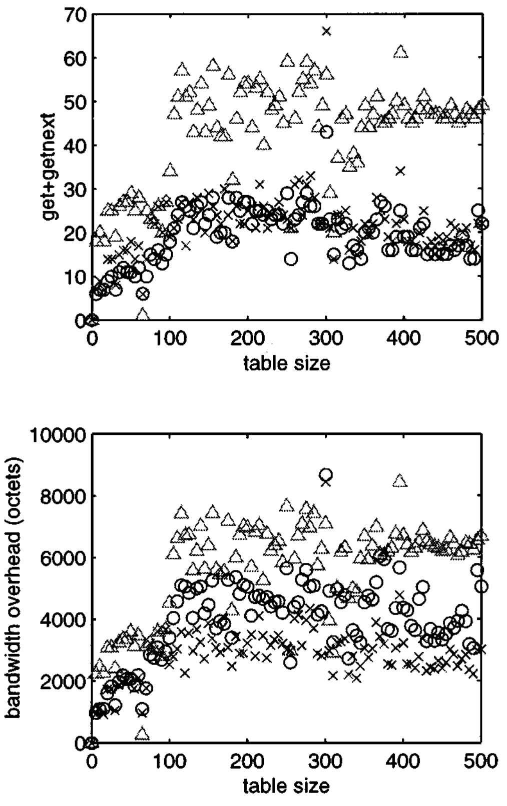


Fig. 8. The performance of the different variants of GetPrev on a TCPConnectionTable.

- **GetPrev (Multiquery):** This is our algorithm with the time/bandwidth tradeoff improvement discussed in Section III-D.

For every invocation of each of the above algorithms we have measured the following parameters:

- the number of SnmpGet and SnmpGetNext messages used to retrieve the value of the lexicographically previous OID;
- the overall time it took to retrieve a value of the previous OID. This time accounts for all of the local computations, the communication between the agent and the application, and the agent's response time;
- the total bandwidth used. This is the number of payload octets sent over the network in the process of retrieving the value of the previous OID.

The results presented in Fig. 7 reflect an average of ten 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 (the reasons for this are discussed in the following section). Moreover, the amount of time it took our tool to get the previous element of the 500th entry in the table was about 0.65 s while it took snmpwalk more than two seconds to accomplish the same task. In terms of bandwidth usage the result is even more dramatic. Our tool used 7000 octets while the traditional snmpwalk used 200 000 octets (a factor of 30).

Fig. 8 shows a detailed picture of the resources used by the different variants of GetPrev (the legend is the same as in Fig. 7). 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 with 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 Fig. 9. 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 2625th element, we can see that while more than eight minutes were required for the traditional snmpwalk , our GetPrev accomplished the same task in less than a second, a factor of more than 500.

A. GetBulk

SnmpGetBulk is a new type of request introduced in version 2 of the SNMP protocol [10] as an optimization to SnmpGetNext in order to improve the performance of downloading large tables [2]. Basically, instead of retrieving only a single next object (like in SnmpGetNext) it retrieves a bulk of data following the next object. The size of the retrieved data bulk is specified by the user. Note that while GetBulk can reduce the number of requests needed to download a table, and can also reduce the number of IP packets used, it will not decrease the total amount of octet transmitted except for the headers. In some cases, it may even increase this number as the last data bulk may contain data that is not part of the table.

Unfortunately, SnmpGetBulks applicability is limited to SNMPv2 and SNMPv3 versions only. In particular, in our experiments we could not test it with the SNMPv1 agent deployed by the Cisco router. Thus, we computed the possible effect of GetBulk instead of measuring it. The amount of resources required by GetBulk depends on the bulk size parameter. Since most SNMP implementation are done on top of UDP, the maximal size of a UDP packet (65 000 octets) is a clear upper bound on the size of the possible bulk size. Because of the ASN.1 encoding schemes about 163 instances of the columnar object like ipRoutingTable can fit into 65 000 octets. Thus, in a very optimistic scenario, a retrieval of the table up to the ith element will work 163 times faster than snmpwalk (compare to the factor of 500 for our GetPrev).

The MTU (which is the maximal size of an IP packet) in our Fast Ethernet network is 1500 octets and, therefore, the number of IP packets used by GetBulk is only a factor of five times

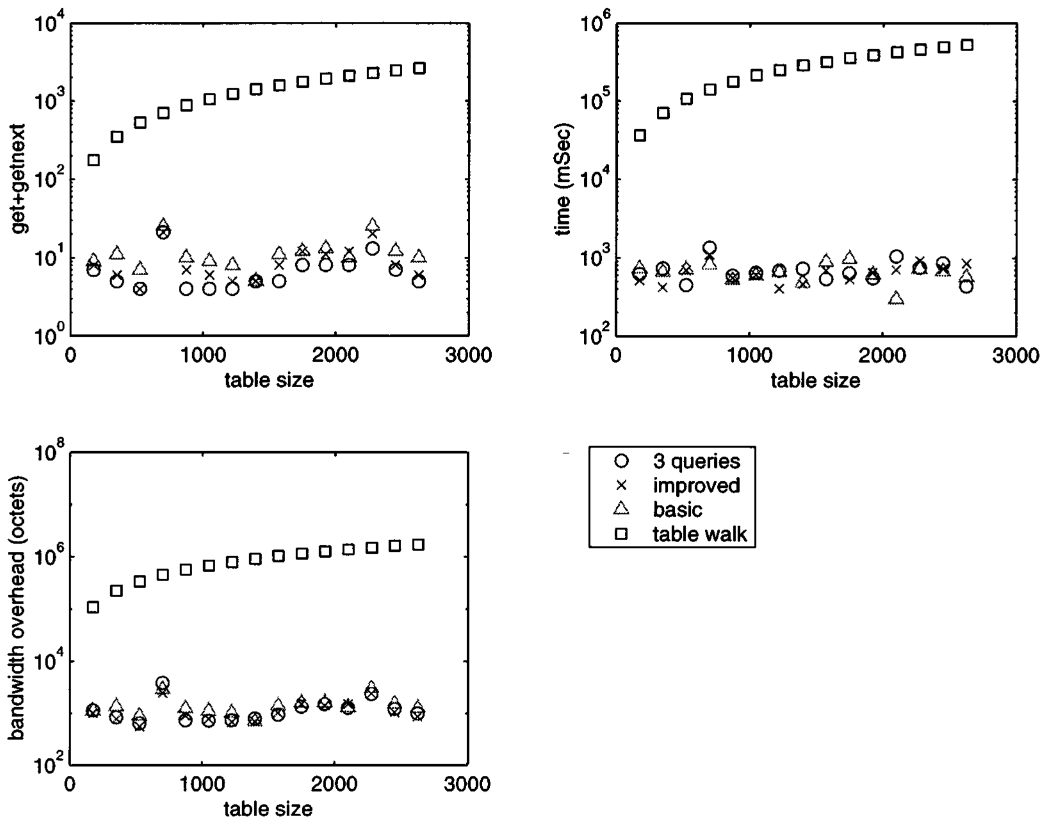


Fig. 9. The performance of GetPrev on a ipRoutingTable.

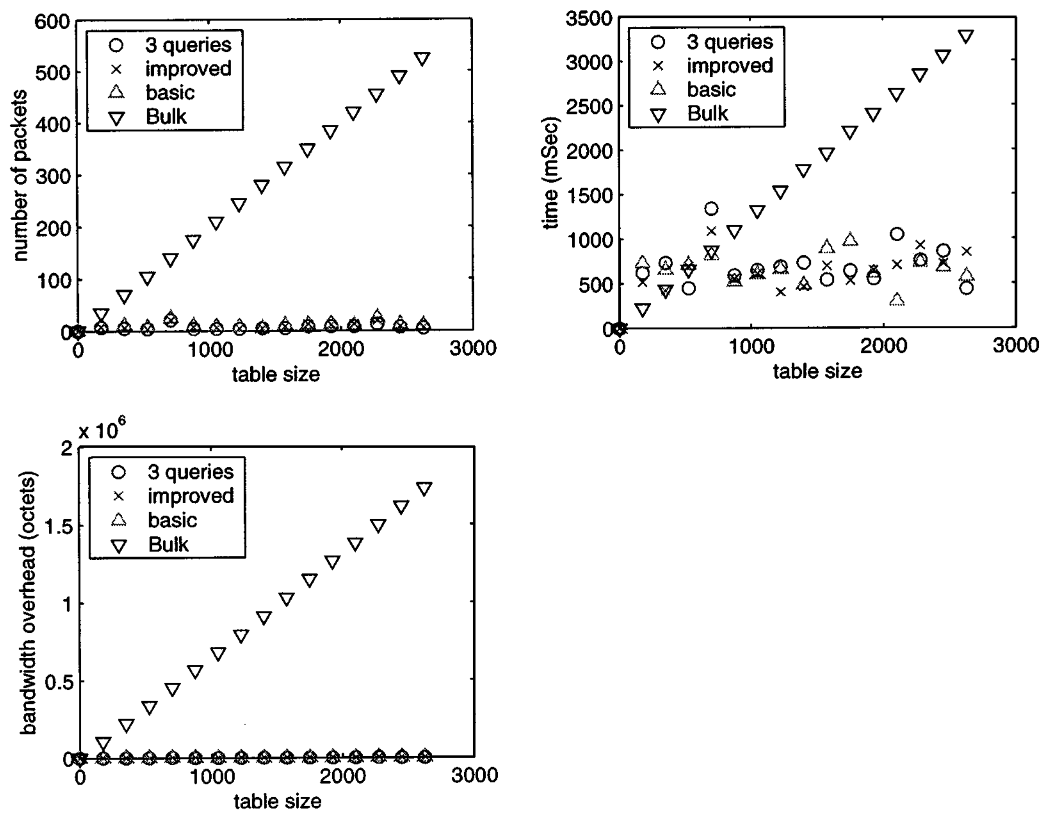


Fig. 10. The performance of GetBulk on a ipRoutingTable.

less than the one used by *snmpwalk*. Note that each large UDP packet is fragmented into smaller MTU size IP packets [14].

As mentioned before, the total number of bandwidth usage of GetBulk is at least the same as that of snmpwalk. Fig. 10

presents the computed resource consumption of the previous object instance retrieval using $\mathrm{GetBulk}$, and compares it to that of $\mathrm{GetPrev}$. As one can observe, the gap between our $\mathrm{GetPrev}$ and this application is smaller with respect to the number of IP packets and processing time. However, our tool still offers a clear advantage for all resource types (packets, time, and overall bandwidth usage) while being compatible with all versions of SNMP.

V. Performance Analysis

In this section, we analyze the expected performance of our $\mathrm{GetPrev}$ tool using the model defined in Section II. Since one can find the beginning of the table using MIB parsing or syntax parsing (simply looking for the last occurrence of the pattern .1.1 in the OID) we assume for this analysis that the table begins at the tree's root. Also, unless specified otherwise, we use $\log(x)$ to denote $\log_2(x)$.

Consider first the basic variant of $\mathrm{GetPrev}$, and let us assume that we want to find the predecessor of $l_1.l_2.\ \ldots\ .l_n$. Recall that the first step of the algorithm is to find the maximal common prefix. This is done using a binary search on the length of the label path (the number of components in the input OID) using no more than $\log(n)$ $\mathrm{Get/GetNext}$ messages. The second step is a bounded binary search that uses a value of the component (in the input OID) that follows the maximal common prefix as the upper bound and zero as the lower bound. Then, for every remaining component, until we find the wanted object, we carry on both a computation of the upper bound and the bounded binary search (see Fig. 5). Note that the number of these steps is bounded by the length of the result, i.e., by the number of components in the OID of the object preceding $l_1.l_2.\ \ldots\ .l_n$. This OID may have more than n components. The number of components is effectively bounded by 128 which is the maximal number of components allowed in a name of SNMP object [1]. However, if both $l_1.l_2.\ \ldots\ .l_n$ and its predecessor are instances of the same columnar object, then they have the same number of components in their OIDs and it equals the number of table indices. Both the upper bound computation and the binary search algorithm take $O(\log(MAXINT))$ $\mathrm{Get/GetNext}$ requests where $MAXINT$ being the maximal value of an Integer index. Finally, there is an additional Get request that retrieves the wanted value.

Putting this all together, if we are searching in a table indexed by n integer keys, the total number of $\mathrm{Get/GetNext}$ request messages is bounded by:

$$O(\log(n)+\log(MAXINT)+2*(n-1)\log(MAXINT)+1).$$

Note that this bound does not depend on a position of the object in the table, or, even more important, on the table size. This is the reason for the superior performance of $\mathrm{GetPrev}$ that we observed in the previous section.

It is important to mention that $\log(n) + \log(MAXINT) + 2 * (n - 1)\log(MAXINT) + 1$ is only an upper bound, for a specific object, the number of $\mathrm{Get/GetNext}$ requests used by GetPrev depends on the actual values of the index components of the wanted object. For example, consider the routing table. It is indexed by an IP address. This means that $\mathrm{GetPrev}$ algorithm should perform the search for four components of the predecessor's OID. However, since all of them belong to IP addresses their value never exceeds 255. Therefore, the number of the $\mathrm{Get/GetNext}$ requests used is, in fact, bounded by $2 + 8 + 2 * 3 * 8 + 1 = 59$. The actual number obtained in the experiments, though, varies depending on the actual values of the fields. This explains the noisy picture in Fig. 8.

This also explains why the search in the TCP connections table demands more requests. In this table, entries have ten components of their OIDs belonging to the indices (two four-component IP addresses of the source and destination and two integer numbers for the source's and destination's ports). The port numbers, however, can be as large as 64 000. Thus, the upper bound on the number of requests in this case is much worse, namely $4+8+2*7*8+2*2*16+1 = 4+8+112+64+1 = 189$.

Now, let us analyze the theoretical performance of the improvements introduced in Sections III-C and III-D. Clearly the upper bound for finding the maximal common prefix does not change as a result of the improvement. Analogously we still need the final Get request to be performed in order to retrieve the desired value. It is easily observed that the minimal lower bound that can be attained by the first phase of the improved bounds algorithm (see Fig. 6), namely the lower bound computation phase, is $MAXINT/2$. Also, as explained above this phase's message complexity is bounded by $\log(MAXINT)$. The upper bound computation starts from the lower bound computed in the previous phase. As one observes, its message complexity is bounded by $\log(MAXINT/2)$. Finally, the binary search algorithm message complexity is also bounded by $\log(MAXINT/2)$.

Putting it all together, we obtain that the message complexity of the improved bounds variant of the $\mathrm{GetPrev}$ algorithm is $O(\log(n)+\log(MAXINT)+(n-1)*[\log(MAXINT)+2*\log(MAXINT/2)]+1)$. Turning back to our ipForwardTable example, we obtain the upper bound of $2 + 8 + 3 * (8 + 2 * 7) + 1 = 77$. As one may notice, the predicted performance of the improved algorithm is worse than that of the basic one. Yet, as our experiments show, the performance of the improved bounds variant of the algorithm is almost always superior to that of the basic variant. Namely, the improved bounds variant uses 10%–15% less $\mathrm{Get/GetNext}$ requests.

The reason for this is simple. As was mentioned in Section III-C. In the real tables, the integer values of the indices of the table entries are not uniformly distributed. Thus, in most cases after the lower bound is found in the first phase of the improved bounds algorithm, it takes only a few messages (usually only one or two) to compute the upper bound and not $\log(MAXINT/2)$ as predicted by the theoretical analysis. When this is not the case, we observe the rare fluctuations of the Improved Bounds performance rendering it worse than the basic algorithm (see Fig. 8).

Finally, we analyze the multiquery variant of the basic algorithm. As was explained in Section III-D, a single packet of the multiquery algorithm contains k different values (in our experiments we used $k = 3$). This means that we partition the search range into $k + 1$ equal intervals, and use a response to the $\mathrm{GetNext}$ request to choose the correct interval. The search in this case takes $\log_{k+1}(MAXINT)$ iterations, instead of the $\log_2(MAXINT)$ required by our basic algorithm. If $\log n + 1$ is small enough when compared with $2n \log(MAXINT)$ the Multiquery variant saves a factor of

$$\frac{\log_2(MAXINT)}{\log_{k+1}(MAXINT)} = \log_2(k+1)$$

in the number of messages. Assuming that the local computation time is negligible, and most of the search time is due to the communication delay and the daemon's response time, a similar factor is saved for the overall search time. As explained above, we pay for this speedup by increasing the size of each packet. We can assume that each packet is k times larger than the basic version's packet. Therefore, the total bandwidth used is a factor of

$$\frac{k}{\log_2(k+1)}$$

larger than that of the basic variant. Indeed, for $k = 3$ we use half the messages ($\log_2(3+1) = 2$) and $3/\log_2(3+1) = 1.5$ times more bandwidth. Note that this is a theoretical analysis, in practice, due to factors such as the MTU size, one cannot use k that is larger than say five.

VI. Conclusion

We have presented $\mathrm{GetPrev}$, 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 $\mathrm{GetPrev}$, compared the performance of our implementation to other alternatives, and also provided the rigorous theoretical analysis of $\mathrm{GetPrev}$ performance.

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 [15] and MIB Script Protocol [16] for implementing $\mathrm{GetPrev}$ locally to the SNMP daemon.

Appendix

A. SNMP Basics

Unarguably, one of the main advantages of SNMP is its simplicity. All management information pertaining to a specific device is made accessible to a manger through a universal interface referred to as *MIB.* MIB defines *classes* (or *types*) of management information called *objects* (or *object types*). Thus, the actual management information is viewed as a collection of *object instances* (also called *variables*) and their values. The set of MIB variables is structured as a rooted tree where leaves correspond to the variables. Each variable has a unique name referred to as its *OID.* SNMP uses the naming model borrowed from Abstract Syntax Notation One (ASN.1) [17]. Accordingly, the OID of a MIB object is an ordered sequence of *components* (called *subidentifiers* in SNMP SMI) written from left to right being separated by "..'' Each component value of OID is a nonnegative Integer number and the number of components in the OID should be at least two. SNMP limits the OID length to 128 components while ASN.1 does not. An interpretation of the OID component values is as following. Starting with the root of the OID tree every component value identifies an arc in the tree. Thus, an OID of the MIB object type which is a node in the tree is, in fact, a unique path from the root of the tree to this node. To illustrate this, node `1.5.2` is accessed using the first child of the root, then the fifth child of the first child, and then the second child of the second level child.

An object may be defined as a *scalar object* meaning that it has exactly one instance, or as a *columnar object.* Columnar objects may have multiple instances where each instance is uniquely identified using some *indexing scheme.* Together with the indexing scheme, a columnar object defines a *conceptual table* (later referred simply as *table*). Entries in the table are accessed using the values of the indices.

An access to the management information items exported by the MIB interface is instrumented by the software agent, called *SNMP daemon* that executes locally on the managed device and possesses the means for retrieval or modification of the values of variables present in the MIB it controls.

Acknowledgment

The authors would like to thank S. Mazumdar for his comments during their work on GetPrev.

References

[1] P. E. Miller and M. A. Miller, *Managing Internetworks with SNMP*, 3rd ed: M&T Books, Nov. 1999.
[2] D. Perkins and E. McGinnis, *Understanding MIBs*. Englewood Cliffs, NJ: Prentice-Hall, 1997.
[3] M. T. Rose and K. McCloghrie, *How to Manage Your Network Using SNMP*. Englewood Cliffs, NJ: Prentice-Hall, Jan. 1995.
[4] W. Stallings, *SNMP, SNMPV2, SNMPV3, and RMON 1 and 2*: Addison-Wesley, Jan. 1999.
[5] R. Sprenkels and J.-P. Martin-Flatin, "Bulk transfers of MIB data," *The Simple Times, The Quarterly Newsletter of SNMP Technology, Comment, and Events*, vol. 7, no. 1, pp. 1–8, Mar. 1999.
[6] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser, "Protocol operations for version 2 of the simple network management protocol (SNMPv2)," *RFC 1905*, Jan. 1996.
[7] AdventNet. Adventnet Web nms. [Online]. Available: http://www.vembu.com/products/snmpbeans/snmpv1/help/uicomponent_mibbro%w.
[8] University of California Davis. Ucd-snmp Project Home Page. [Online]. Available: http://net-snmp.sourceforge.net.
[9] S. Mazumdar and K. Swenson. Corba/snmp Gateway Project. [Online]. Available: http://www.bell-labs.com/project/CorbaSnmp/JavaORBImpl/Demo/MibBrowser.%html.
[10] K. McCloghrie and M. Rose, "Management information base for network management of TCP/IP-based internets: MIB-II," *Internet RFC 1213*, Mar. 1991.
[11] M. Rose, K. McCloghrie, and J. Devin, "Bulk table retrieval with SNMP," *RFC 1187*, Oct. 1990.
[12] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 1998.
[13] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, 2nd ed. Reading, MA: Addison-Wesley, 1998, vol. 3.
[14] W. R. Stevens, *UNIX Network Programming*, ser. Software Series. Englewood Cliffs, NJ: Prentice-Hall, 1990.
[15] M. Daniele, B. Wijnen, M. Ellison, and D. Francisco, "RFC 2741: Agent Extensibility (AgentX) Protocol Version 1," IETF, Jan. 2000.

[16] D. Levi and J. Schoenwaelder, "RFC 2592: Definitions of Managed Objects for the Delegation of Management Script," IETF, May 1999.

[17] D. Steedman, "Abstract syntax notation one (ASN.1), the tutorial and reference," Technology Appraisals, Ltd., Isleworth, Middlesex, U.K., ISBN 1-871 802-06-7, 1990.

David Breitgand (S'97–A'01) received the B.Sc. and M.Sc. degrees in computer science (*cum laude*) from The Hebrew University of Jerusalem, Israel, in 1994 and 1997, respectively. Currently, he is pursuing a Ph.D. degree in computer science from the Hebrew University of Jerusalem. His advisers are Prof. Danny Dolev, and Dr. Danny Raz.

From 1994 to 1999, he worked first as a Network Administrator, and later as a Software Engineer in a Distributed and Networking Secure Systems Lab at the Hebrew University. His primary research interest is application of the theory of distributed computing to network management problems.

Danny Raz (M'99) received the doctoral degree from the Weizmann Institute of Science, Israel, in 1995.

From 1995 to 1997 he was a Postdoctoral Fellow at the International Computer Science Institute, (ICSI) Berkeley, CA, and a visiting lecturer at the University of California, Berkeley. From 1997 to 2001 he was a Member of the Technical Staff at the Networking Research Laboratory at Bell Labs, Lucent Technologies, Holmdel, NJ. In October 2000, he joined the Faculty of the Computer Science Department at the Technion, Israel. His primary research interest is the theory and application of management related problems in IP networks. He served as the general chair of OpenArch 2000. He is an Editor for the *Journal of Communications and Networks* (JCN)

Dr. Raz is a TPC member for INFOCOM 2002, OpenArch 2000-2001 IM 2001, NOMS 2002, and GI 2002.

Yuval Shavitt (S'88–M'97–SM'00) received the B.Sc. degree in computer engineering (*cum laude*), the M.Sc. degree in electrical engineering, and the D.Sc. degree from the Technion, Israel Institute of Technology, Haifa, in 1986, 1992, and 1996, respectively.

From 1986 to 1991, he served in the Israel Defense Forces first as a System Engineer and the last two years as a Software Engineering Team Leader. After graduation, he spent a year as a Postdoctoral Fellow at the Department of Computer Science at Johns Hopkins University, Baltimore, MD. From 1997 to 2001, he was a Member of the Technical Staff at the Networking Research Laboratory at Bell Labs, Lucent Technologies, Holmdel, NJ. Since October 2000, he has been a Faculty Member in the Department of Electrical Engineering, Tel-Aviv University, Tel-Aviv, Israel. His recent research focuses on active networks and their use in network management, QoS routing, and Internet mapping, and characterization.

Dr. Shavitt served as TPC member for INFOCOM 2000–2002, IWQoS 2001 and 2002, ICNP 2001, and MMNS 2001, and on the executive committee of INFOCOM 2000, 2002, and 2003.