Network Working Group
Request for Comments: 1187

M. Rose
Performance Systems International, Inc.
K. McCloghrie
Hughes LAN Systems
J. Davin
MIT Laboratory for Computer Science
October 1990

Bulk Table Retrieval with the SNMP

1. Status of this Memo

This memo reports an interesting family of algorithms for bulk table retrieval using the Simple Network Management Protocol (SNMP). This memo describes an Experimental Protocol for the Internet community, and requests discussion and suggestions for improvements. This memo does not specify a standard for the Internet community. Please refer to the current edition of the "IAB Official Protocol Standards" for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Table of Contents

2. Abstract

This memo reports an interesting family of algorithms for bulk table retrieval using the Simple Network Management Protocol (RFC 1157) [1].

The reader is expected to be familiar with both the Simple Network Management Protocol and SNMP's powerful get-next operator. Please send comments to: Marshall T. Rose <mrose@psi.com>.

3. Bulk Table Retrieval with the SNMP

Empirical evidence has shown that SNMP's powerful get-next operator is effective for table traversal, particularly when the management station is interested in well-defined subsets of a particular table. There has been some concern that bulk table retrieval can not be efficiently accomplished using the powerful get-next operator. Recent experience suggests otherwise.

In the simplest case, using the powerful get-next operator, one can traverse an entire table by retrieving one object at a time. For example, to traverse the entire ipRoutingTable, the management station starts with:

```
get-next (ipRouteDest)
```

which might return

```
ipRouteDest.0.0.0.0
```

The management station then continues invoking the powerful get-next operator, using the value provided by the previous response, e.g.,

```
get-next (ipRouteDest.0.0.0.0)
```

As this sequence continues, each column of the ipRoutingTable can be retrieved, e.g.,

```
get-next (ipRouteDest.192.33.4.0)
```

which might return

```
ipRouteIfIndex.0.0.0.0
```

Eventually, a response is returned which is outside the table, e.g.,

```
get-next (ipRouteMask.192.33.4.0)
```

which might return

```
ipNetToMediaIfIndex.192.33.4.1
```

So, using this scheme, O(rows x columns) management operations are required to retrieve the entire table.

This approach is obviously sub-optimal as the powerful get-next operator can be given several operands. Thus, the first step is to retrieve an entire row of the table with each operation, e.g.,

```
get-next (ipRouteDest, ipRouteIfIndex, ..., ipRouteMask)
```

which might return

```
ipRouteDest.0.0.0.0
ipRouteIfIndex.0.0.0.0
ipRouteMask.0.0.0.0
```

The management station can then continue invoking the powerful get-next operator, using the results of the previous operation as the operands to the next operation. Using this scheme O(rows) management operations are required to retrieve the entire table.

Some have suggested that this is a weakness of the SNMP, in that O(rows) serial operations is time-expensive. The problem with such arguments however is that implicit emphasis on the word "serial". In fact, there is nothing to prevent a clever management station from invoking the powerful get-next operation several times, each with different operands, in order to achieve parallelism and pipelining in the network. Note that this approach requires no changes in the SNMP, nor does it add any significant burden to the agent.

4. The Pipelined Algorithm

Let us now consider an algorithm for bulk table retrieval with the SNMP. In the interests of brevity, the "pipelined algorithm" will retrieve only a single column from the table; without loss of generality, the pipelined algorithm can be easily extended to retrieve all columns.

The algorithm operates by adopting a multi-threaded approach: each thread generates its own stream of get-next requests and processes the resulting stream of responses. For a given thread, a request will correspond to a different row in the table.

Overall retrieval efficiency is improved by being able to keep several transactions in transit, and by having the agent and management station process transactions simultaneously.

The algorithm will adapt itself to varying network conditions and topologies as well as varying loads on the agent. It does this both by varying the number of threads that are active (i.e., the number of transactions that are being processed and in transit) and by varying the retransmission timeout. These parameters are varied based on the

transaction round-trip-time and on the loss/timeout of transactions.

4.1. The Maximum Number of Active Threads

One part of the pipelined algorithm which must be dynamic to get best results is the determination of how many threads to have active at a time. With only one thread active, the pipelined algorithm degenerates to the serial algorithm mentioned earlier. With more threads than necessary, there is a danger of overrunning the agent, whose only recourse is to drop requests, which is wasteful. The ideal number is just enough to have the next request arrive at the agent, just as it finishes processing the previous request. This obviously depends on the round-trip time, which not only varies dynamically depending on network topology and traffic-load, but can also be different for different tables in the same agent.

With too few threads active, the round-trip time barely increases with each increase in the number of active threads; with too many, the round-trip time increases by the amount of time taken by the agent to process one request. The number is dynamically estimated by calculating the round-trip-time divided by the number of active threads; whenever this value takes on a new minimum value, the limit on the number of threads is adjusted to be the number of threads active at the time the corresponding request was sent (plus one to allow for loss of requests).

4.2. Retransmissions

When there are no gateways between the manager and agent, the likelihood of in-order arrival of requests and responses is quite high. At present, the decision to retransmit is based solely on the timeout. One possible optimization is for the manager to remember the order in which requests are sent, and correlate this to incoming responses. If one thread receives a response before another thread which sent an earlier request, then lossage could be assumed, and a retransmission made immediately.

4.3. Some Definitions

To begin, let us define a "thread" as some state information kept in the management station which corresponds to a portion of the table to be retrieved. A thread has several bits of information associated with it:

(1) the range of SNMP request-ids which this thread can use, along with the last request-id used;

(2) last SNMP message sent, the number of times it has been

(re)sent, the time it was (re)sent;

(3) the inclusive lower-bound and exclusive upper-bound of the object-instance for the portion of the table that this thread will retrieve, along with the current object-instance being used;

(4) the number of threads which were active at the time it was last sent;

When a thread is created, it automatically sends a get-next message using its inclusive lower-bound OID. Further, it is placed at the end of the "thread queue".

Let us also define an OID as a concrete representation of an object identifier which contains two parts:

(1) the number of sub-identifiers present, "nelem";

(2) the sub-identifiers themselves in an array, "elems", indexed from 1 up to (and including) "nelem".

4.4. Top-Level

The top-level consists of starting three threads, and then entering a loop. As long as there are existing threads, the top-level waits for events (described next), and then acts upon the incoming messages. For each thread which received a response, a check is made to see if the OID of the response is greater than or equal to the exclusive upper-bound of the thread. If so, the portion of the table corresponding to the thread has been completely retrieved, so the thread is destroyed.

Otherwise, the variable bindings in the response are stored. Following this, if a new thread should be created, then the portion of the table corresponding to the thread is split accordingly. Regardless, another powerful get-next operator is issued on behalf of the thread.

The initial starting positions (column, column.127, and column.192), were selected to form optimal partitions for tables which are indexed by IP addresses. The algorithm could easily be modified to choose other partitions; however, it must be stressed that the current choices work for any tabular object.

```
pipelined_algorithm (column)
OID  column;
{
```

```
        timeout ::= some initial value;

        start new thread for [column, column.127);
        start new thread for [column.127, column.192);
        start new thread for [column.192, column+1);

        while (threads exist) {
           wait for events;
           foreach (thread that has an incoming message,
                    examined in order from the thread queue) {
               OID     a;

               if (message's OID >= thread's upper-bound) {
                   destroy thread;
                   continue;
               }

               store variable-bindings from message;

               if (number of simultaneous threads does NOT
                           exceed a maximum number
                        && NOT backoff
                        && (a ::= oid_median (message's OID,
                                              thread's
                                                  upper-bound))) {
                    start new thread for [a, thread's upper-bound);
                    thread's upper-bound ::= a;
                    place thread at end of thread queue;
                    backoff ::= TRUE;
                }
                do another get-next for thread;
            }
        }
    }
```

4.5. Wait for Events

Waiting for events consists of waiting a small amount of time or until at least one message is received.

Any messages encountered are then collated with the appropriate thread. In addition, the largest round-trip time for request/responses is measured, and the maximum number of active threads is calculated.

Next, the timeout is adjusted: if no responses were received, then the timeout is doubled; otherwise, a timeout-adjustment is calculated

as 1.5 times the largest observed round-trip time. If the timeout-adjustment is greater than the current timeout, the current timeout is set to the timeout-adjustment. Otherwise, the current timeout is averaged with the timeout-adjustment.

Finally, if at least one thread did not receive a response, then the thread is identified which has waited the longest. If the elapsed time (with noise factor) since the last request (or retransmission) is greater than the current timeout value, another retransmission is attempted.

```
wait for events ()
{
    backoff ::= TRUE, maxrtt ::= 0;
    find the thread which has been waiting the longest
        for a response;
    timedelta = timeout
                    - time since request was sent for thread;
    wait up to timedelta seconds or until some messages arrive;

    if (least one message arrived) {
        discard any messages which aren't responses;
        foreach (response which corresponds to a thread) {
            if (the response is a duplicate)
                drop it and continue;

            if (this response is for a message that was
                    not retransmitted) {
               if (the round-trip time is larger than maxrtt)
                    set maxrtt to the new round-trip time;
                if (round-trip time / number of active threads
                      < minimum previous round-trip time / number
                           of active threads) {
                    set new minimum round-trip time per number of
                        active threads
                    set new maximum number of threads
               }
                backoff ::= FALSE;
            }
        }
    }
    if (backoff)
        double timeout;
    elsif (maxrtt > 0) {
       timeadjust ::= maxrtt * 3 / 2;
        if (timeadjust > timeout)
            timeout ::= timeadjust; backoff ::= TRUE;
        else
```

```
                timeout ::= (timeout + timeadjust) / 2;
        }
        if (timeout exceeds some threshold)
           set timeout to that threshold;
      elsif (timeout is smaller than some threshold)
           set timeout to that threshold;

        if (at least one thread didn't receive a response) {
            find the thread which has been waiting the longest
                for a response,
                and determine the elapsed time since a message
                was sent;
            if (the elapsed time with noise is greater than timeout) {
                if (the number of retransmissions for this thread
                        exceeds a threshold)
                    abort the algorithm;
                retransmit the request;
                backoff ::= TRUE;
            }
        }
  }
```

4.6. Finding the Median between two OIDs

The object identifier space is neither uniform nor continuous. As such, it is not always possible to choose an object identifier which is lexicographically-between two arbitrary object identifiers. In view of this, the pipelined algorithm makes a best-effort attempt.

Starting from the beginning, each sub-identifier of the two OIDs is scanned until a difference is encountered. At this point there are several possible conditions:

(1) The upper OID has run out of sub-identifiers. In this case, either the two OIDs are are identical or the lower OID is greater than the upper OID (an interface error), so no OID is returned.

(2) The lower OID has run out of sub-identifiers. In this case, the first subsequent non-zero sub-identifier from the upper OID is located. If no such sub-identifier is found, then no OID exists between the lower and upper OIDs, and no OID is returned. Otherwise, a copy of the upper OID is made, but truncated at this non-zero sub-identifier, which is subsequently halved, and the resulting OID is returned.

(3) Otherwise, a copy of the lower OID is made, but truncated

at the point of difference. This last sub-identifier is then set to the arithmetic mean of the difference. In the case where the difference is only 1 (so the last sub-identifier remains the same) then a new sub-identifier is added, taking care to be larger than a possible sub-identifier present in the lower OID. Regardless, the resulting OID is returned.

```
oid_median (lower, upper)
OID     lower,
        upper;
{
    for (i ::= 1; i < upper:nelem; i++) {
        if (i > lower:nelem) {
            while (upper:elems[i] == 0)
                if (++i > upper:nelem)
                    return NULL;
            median ::= copy of upper;
            median:nelem ::= i;
            median:elems[i] ::= upper:elems[i] / 2;

            return median;
       }

       if (lower:elems[i] == upper:elems[i])
           continue;

        median ::= copy of lower;
        median:nelem ::= i;
        median:elems[i] ::= (lower:elems[i]+upper:elems[i])/2;
        if (median:elems[i] == lower:elems[i]) {
            median:nelem ::= (i + 1);
           if (lower:nelem < i)
               median:elems[median:nelem] ::= 127;
            elsif ((x ::= lower:elems[i + 1]) >= 16383)
               median:elems[median:nelem] ::= x + 16383;
            elsif (x >= 4095)
               median:elems[median:nelem] ::= x + 4095;
            elsif (x >= 1023)
                median:elems[median:nelem] ::= x + 1023;
            elsif (x >= 255)
                median:elems[median:nelem] ::= x + 255;
            else median:elems[median:nelem] ::=
                                        (x / 2) + 128;
        }

         return median;
    }
```

```
            return NULL;
        }
```

4.7. Experience with the Pipelined Algorithm

This pipelined algorithm has been implemented and some experimentation has been performed. It would be premature to provide extensive performance figures at this time, as the pipelined algorithm is still being tuned, and is implemented only in a prototype setting. However, on tables of size O(2500), performance of 121 entries/second has been observed. In contrast, the serial algorithm has performance of roughly 56 entries/second for the same table.

4.8. Dynamic Range of Timeout Values

It should be noted that the pipelined algorithm takes a simplistic approach with the timeout value: it does not maintain a history of the value and act accordingly.

For example, if the timeout reaches the maximum timeout limit, and then latches for some period of time, this indicates a resource (either the network or the agent) is saturated. Unfortunately, a solution is difficult: an elegant approach would be to combine two threads (but it is quite possible that no two consecutive threads exist when this determination is made). Another approach might be to delay the transmission for threads which are ready to issue a new get-next operation.

Similarly, if the timeout drops to the minimum value and subsequently latches, more threads should be started.

4.9. Incorrect Agent Implementations

An interesting result is that many agents do not properly implement the powerful get-next operator. In particular, when a get-next request contains an operand with an arbitrarily-generated suffix, some agent implementations will handle this improperly, and ultimately return a result which is lexicographically less than the operand!

A typical cause of this is when the instance-identifier for a columnar object is formed by a MAC or IP address, so each octet of the address forms a sub-identifier of the instance-identifier. In such circumstances, the incorrect agent implementations compare against only the least significant octet of the sub-identifiers in the operand, instead of the full value of the sub-identifiers.

Upon encountering such an interaction, the pipelined algorithm implementation declares the thread dead (noting a possible gap in the table), and continues.

5. The Parallel Algorithm

One interesting optimization is to view the problem in two steps: in the first step, one column of the table is traversed to determine the full range of instances identifiers meaningful in the table. (Indeed, although as described above, the pipelined algorithm retrieves a single column, the prototype implementation can retrieve multiple columns). In the second step, additional columns can be retrieved using the SNMP get operation, since the instance identifiers are already known. Further, the manager can dynamically determine how many variables can be placed in a single SNMP get operation in order to minimize the number of requests. Of course, since the agent's execution of the get operation is often less expensive than execution of the powerful get-next operation, when multiple columns are request, this two-step process requires less execution time on the agent.

A second algorithm can be developed, the "parallel algorithm". At present, each thread is mapped onto a single SNMP operation. A powerful insight is to suggest mapping several threads onto a single SNMP operation: the manager must dynamically determine how many variables can be placed in a single powerful get-next operation. This has the advantage of reducing traffic, at the expense of requiring the agent to utilize more resources for each request.

Earlier it was noted that the serial retrieval of objects could be viewed as a degenerate case of the pipelined algorithm, in which the number of active threads was one. Similarly, the pipelined algorithm is a special case of the parallel algorithm, in which the number of threads per SNMP operation is one.

5.1. Experience with the Parallel Algorithm

The parallel algorithm has been implemented and some experimentation has been performed. It would be premature to provide extensive performance figures at this time, as the algorithm is still being tuned, and is implemented only in a prototype setting. However, on tables of size O(2500), performance of 320 entries/second has been observed, a performance improvement of 571% over the serial algorithm.

6. Acknowledgements

A lot of the ideas on pipelining are motivated by Van Jacobson's work

on adaptive timers in TCP. The parallelization modifications were originally suggested by Jeffrey D. Case.

Finally, the comments of the following individual is acknowledged:

Frank Kastenholz, Racal-Interlan

7. References

[1] Case, J., Fedor, M., Schoffstall, M., and J. Davin, Simple Network Management Protocol (SNMP), RFC 1157, SNMP Research, Performance Systems International, Performance Systems International, MIT Laboratory for Computer Science, May 1990.

Security Considerations

Security issues are not discussed in this memo.

Authors' Addresses

Marshall T. Rose
PSI, Inc.
PSI California Office
P.O. Box 391776
Mountain View, CA 94039

Phone: (415) 961-3380
EMail: mrose@PSI.COM

Keith McCloghrie
Hughes LAN Systems
1225 Charleston Road
Mountain View, CA 94043

Phone: (415) 966-7934
EMail: KZM@HLS.COM

James R. Davin
MIT Laboratory for Computer Science, NE43-507
545 Technology Square
Cambridge, MA 02139

Phone: (617) 253-6020
EMail: jrd@ptt.lcs.mit.edu