

Received October 21, 2020, accepted December 4, 2020, date of publication January 5, 2021, date of current version January 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3049310

On Performance and Scalability of Cost-Effective SNMP Managers for Large-Scale Polling

PAULA ROQUERO¹ AND JAVIER ARACIL²

¹Naudit HPCN, Parque Científico de Madrid, 28049 Madrid, Spain

²High Performance Computing and Networking Group, Tecnología Electrónica y de las Comunicaciones E.P.S, Universidad Autónoma de Madrid, 28049 Madrid, Spain

Corresponding author: Paula Roquero (paula.roquero@naudit.es)

This work was supported by Ayudas para la formación de doctores en empresas, Doctorados Industriales, under Grant DI-16-08979.


ABSTRACT As networks grow larger in size and complexity, their monitoring is becoming an increasing challenge because of the required polling performance and also due to heterogeneity of devices. As it turns out, SNMP (Simple Network Management Protocol) is by far the most popular monitoring protocol. However, due to the increase in the number of network devices, it becomes necessary to employ multiple SNMP managers, which is not cost-effective due to the hardware requirements. Additionally, the different proprietary SNMP implementations require custom configuration very often, as new devices are being incorporated into the network. Therefore, current SNMP managers not only require capabilities for large-scale monitoring but also a high degree of flexibility and programmability. In response, we propose an SNMP manager architecture with a flexible multi-threaded architecture, which effectively reduces the hardware resources necessary to poll the increasing number of SNMP agents. In addition, it features a scripting component to deal with the different data representations caused by proprietary implementations. Our experience has shown that SNMP agents can have high variability in their response times. Actually, our findings show a strong correlation between high response times and CPU load. As a solution, we propose and analyze novel adaptive polling algorithms that decrease the load on agents' CPUs while keeping the desired polling rate for fast agents. Finally, we present several real-world use cases where we show the benefits of the polling algorithms and the scripting component, by means of extensive measurement campaigns.

INDEX TERMS Adaptive polling, networking, parallelism, SNMP.

I. INTRODUCTION

The vast majority of legacy networks today use SNMP as a management protocol. The first SNMP versions date back from the '90s (RFC 1157 [1]) and its adoption has been growing ever since. The main advantage of SNMP is simplicity and stateless nature. Management information is stored in *Management Information Bases* (MIBs) [2] whose elements are well identified by standardized *Object Identifiers* (OIDs). The SNMP manager polls agents at the monitored devices by means of the OIDs and gets the desired values (for instance, bits transmitted through a given switch interface). SNMP uses UDP as the underlying protocol to send the queries and get the corresponding replies, which is also stateless, thus providing simplicity at the transport layer as well.

Nevertheless, as datacenters and networks grow in size and complexity, not comparable to that of the '90s, it becomes

The associate editor coordinating the review of this manuscript and approving it for publication was Guangjie Han .

necessary to revisit the performance of SNMP in such scenarios. Since the number of devices to be polled by the SNMP managers is in the thousands, scalability becomes a major issue. A common workaround for scalability is to split the set of devices into separate groups and deploy a network manager for each of them, possibly in several virtual machines [3], [4]. Such horizontal scalability strategy takes care of the issue but comes at a cost. On the one hand, it consumes valuable computing and software resources; indeed, from our deployment experience with operators, hundreds of virtual machines are employed, which not only consume hardware resources but also software licenses. On the other hand, horizontal scalability entails data disaggregation in the different network managers, which complicates matters for data correlation afterward.

In this paper, we investigate the scalability and performance issues of SNMP managers for large-scale polling in IT infrastructures, namely networks and datacenters, driven by the need for cost-effective solutions in such

complex scenarios. In this regard, we wish to evaluate possible architectures for SNMP managers that can be successfully implemented with open-source programming languages and operating systems using off-the-shelf hardware. Clearly, polling a large population of SNMP agents requires the use of multithreading. However, the issue is to find the optimal number of devices per thread, taking into account that devices' response time is variable, during which the thread is blocked waiting for the response. Needless to say, this research does not prevent horizontal scalability of the SNMP manager instances. It actually contributes to a more cost-effective partitioning of the set of managed devices in terms of occupied hardware and software devices.

Furthermore, our experimental measurements in large datacenters show that the response time of SNMP devices can be significant. As new multimedia or real-time cooperative services are being increasingly offered on the Internet, the timescale for monitoring decreases below the typical five-minute-long intervals. Thus, the response time to SNMP queries eventually distorts the resulting time-series, as it adds to the polling interval. Consequently, we propose a novel dynamic polling algorithm for SNMP that either upgrades or downgrades agents, in terms of polling frequency, as the response time either decreases or increases, respectively. We find a strong correlation between response time and CPU load. Thus, in case the response time grows, the polling frequency should be decreased not to add more fuel to the fire. We propose novel dynamic polling techniques that take into account this issue.

Finally, we also note that there are many proprietary MIBs and that the standard MIBs specifications leave room for slight differences in implementation [5] that, in turn, make it necessary to tune the SNMP manager frequently or use a translation layer [6]. To do so, we present a software architecture based on Python scripting that allows to swiftly adapt to any kind of MIB.

In conclusion, the contributions of this paper are the following. First, we investigate design alternatives for accommodating as many devices as possible in a single SNMP manager instance. In doing so, we consider the interplay between managed device response time and multithreading architecture, with an open-source approach. Such software design alternatives also take into account ad-hoc parsing of MIBs, which can be very heterogeneous. Secondly, we propose a novel dynamic polling algorithm that takes into account the response time, and thus the CPU load, of the polled device. Our methodology encompasses prototype implementation and extensive performance evaluation in large real-world datacenters and networks. Thus, the paper results are supported not only by extensive lab experiments but also by data collection in real-world datacenters.

The paper is structured as follows. In section II we present the design alternatives for the implementation of SNMP manager instances. Section IV-A is devoted to extensive performance evaluation in a simulated lab environment. Section III presents novel adaptive polling algorithms that take into

account variable response time of the polled devices, which is primarily related to CPU load, as we will show. In IV an extensive performance evaluation is carried out, by means of laboratory experiments and real-world datacenter measurements. Section IV-B shows real-world use cases that exemplify the design principles presented in this paper. Finally, we provide the conclusions and lessons learned in section V.

II. COST-EFFICIENT ARCHITECTURES FOR SNMP MANAGERS

In this section, we discuss the different architectures for the cost-effective design of SNMP managers for large-scale polling. We begin by discussing single thread architectures and then resort to multithread alternatives, that foster better performance.

A. SINGLE-THREAD ARCHITECTURES

Single-thread architectures are the easiest to develop but lack the necessary performance. A well-known example is the *Net-SNMP* [7] single-thread SNMP manager, which can be installed as a package in most Linux distributions. It actually comes as a command-line utility that can be invoked through standard shell scripts.

In order to poll agents, *Net-SNMP* provides the *snmpget* or *snmpwalk* command-line utilities, the first of which allows us to get individual values, while the second one can retrieve tables. Once the SNMP data is readily available, UNIX command-line utilities such as *sed* and *awk* can be used to parse the results. When more complex analysis are required, *Python* scripts can also be used for this purpose.

However, this approach can only be adopted when dealing with few SNMP agents. Although the process can be parallelized by running several scripts at the same time, it quickly becomes unwieldy. In addition, the overhead of launching a new SNMP process for each query penalizes performance.

In our experiments with *Net-SNMP*, retrieval of the network interfaces table from *localhost* with *snmpwalk* took 40 ms on average, being the SNMP manager and agent in the same host. This means that in the best case there's an upper limit of 7500 hosts that can be queried every 5 minutes with one process. In a real-world scenario, with more tables included in the query, each walk can take hundreds or thousands of milliseconds to complete depending on the load and processing power of the agent. Thus, we believe that this approach is only useful for initially exploring which MIBs and OIDs are present on a device, as well as its format, but not for large-scale polling. Such initial exploration of SNMP MIBs is a most important issue since this information is often poorly documented, especially with non-standard MIBs.

The above limitations imply that a multithreaded alternative is in order, as we will discuss next.

B. MULTI-THREAD ARCHITECTURES

In order to evaluate multi-threading architectures for SNMP managers, we considered different alternatives for programming languages and libraries:

- First we considered Python but decided to not use it because threads cannot be launched in parallel due to the *Global Interpreter Lock* [8] that prevents threads from running in parallel.
- Next, we considered C++, as it features several SNMP libraries. As it turns out, *Net-SNMP* can be used as a library, but is not thread-safe, so it can only be instantiated as a single thread. Another candidate library is *SNMP++* [9], which provides thread-safety. This solution provided the best performance. However, we decided to discard C++ because it is not memory safe and can lead to buffer overflows¹ and memory leaks. These problems cause security issues [10] and increase development time due to the increased debugging time necessary to find and fix them. In contrast, other languages take care of most of these issues, including garbage collection, and allow the software designed to focus on optimal multi-threading schemes.
- Finally, we chose Java because of its good performance and support for threads, and the open-source *SNMP4J* library [11].

In Java, we represent each remote SNMP agent as an *SnmAgent* object. This object contains the address of the agent and the OIDs we wish to query. To perform the queries an array of *SnmAgent* objects is created. Then, we use a thread pool with a fixed number of threads, where each thread takes agents from the array and performs queries for all the OIDs we wish to query. Once all agents have been queried, the results are printed and the program waits until the next polling time to make requests again. We chose this approach since we will deploy the manager in environments with thousands of agents, where having a *single thread per agent is unfeasible*.

The design approach was not to have more than a single thread polling a single agent at the same time, not to overload the agent with several parallel requests, which could eventually increase the response time and impact the agent operation. Furthermore, even if a thread is busy with a slow agent, the rest of the thread pool can keep querying other agents, reducing the time to complete all requests. Finally, in case a thread is kept waiting for too long, due to a slow SNMP agent, a timeout can be configured to avoid the entire thread pool.

C. JYTHON

To allow for flexibility, Jython [12] (a version of Python that runs on the JVM (*Java Virtual Machine*)) is used as a scripting language. The code written in Jython is in charge of controlling the Java core by providing information about which SNMP agents to query, which MIBs to be queried, and how to handle the responses from the agent. Actually, Jython allows for faster development when dealing with heterogeneous MIBs, some of them proprietary, or complex use cases

whereby a query needs the results of a previous query to be performed.

For example, we found an issue when retrieving VLAN information from several routers. Usually, SNMP agents are queried using a password called *community* that gives direct access to all OIDs. However, the MIBs from such routers did not allow direct access to VLAN tables. Instead, we had to query a given table with a list of VLAN IDs using the regular *community*, and then, get detailed information about each VLAN by querying a different table with the *community* format “*community@VLAN ID*”. Figure 1 shows the process. Should Jython had not been adopted, this procedure would have to be hard-coded into the Java code and would need a new deployment and recompilation of the software.

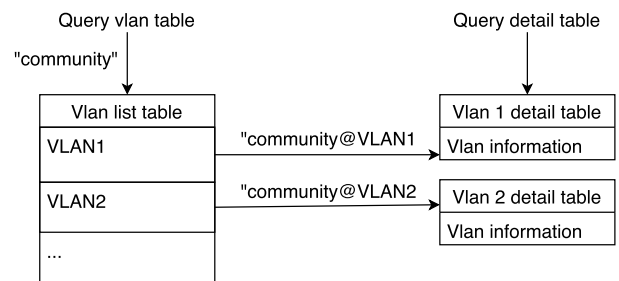


FIGURE 1. Querying vlans in two steps.

Another scenario where Jython has proven useful is in combining information from different tables. For example, tables containing different pieces of network information usually use the interface ID as a field. Thus, such tables can be joined by using these ids, allowing us to correlate different kinds of information. Figure 2 shows an example joining the interfaces and physical address tables using the interface index.

The final benefit of Jython is that analysts with python programming experience can write their own Jython code to fulfill their needs, instead of having to wait for a new feature to be added to the core Java code.

III. ADAPTIVE POLLING ALGORITHM

When dealing with real SNMP agents, we found that some of them showed a larger response time to queries. In Figure 3 we show a response time histogram for several walks performed in different agents. We observe that the histogram follows a bi-modal distribution such that response time is usually low, but some of them are rather large. As we will show in Section IV-B1, this happens when the agent CPU is loaded.

This becomes an issue since we may wish to query agents more often than the slowest one allows. For example, we may want to query agents every minute, but if one of the agents takes five minutes to reply, the entire thread pool will be blocked until the last agent finishes responding, resulting in a polling interval of five minutes in that round, for all agents. Thus, we designed and implemented an adaptive polling scheduler that tackles this issue, by upgrading or downgrading agents according to their response time.

¹<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow>

ipAddrTable		
ipAdEntAddr	192.168.1.10	192.168.1.20
ipAdEntIfIndex	1	2
ipAdEntNetMask	255.255.255.0	255.255.255.0
ipAdEntBcastAddr	192.168.1.255	192.168.1.255
ipAdEntReasmMaxSize	65535	65535

atTable		
atIfIndex	1	2
atPhysAddress	00:0c:29:bd:e8:07	00:0c:29:bd:e8:08
atNetAddress	192.168.1.10	192.168.1.20

Combined table		
ipAdEntAddr	192.168.1.10	192.168.1.20
ipAdEntIfIndex	1	2
ipAdEntNetMask	255.255.255.0	255.255.255.0
ipAdEntBcastAddr	192.168.1.255	192.168.1.255
ipAdEntReasmMaxSize	65535	65535
atIfIndex	1	2
atPhysAddress	00:0c:29:bd:e8:07	00:0c:29:bd:e8:08
atNetAddress	192.168.1.10	192.168.1.20

FIGURE 2. Joining two tables.

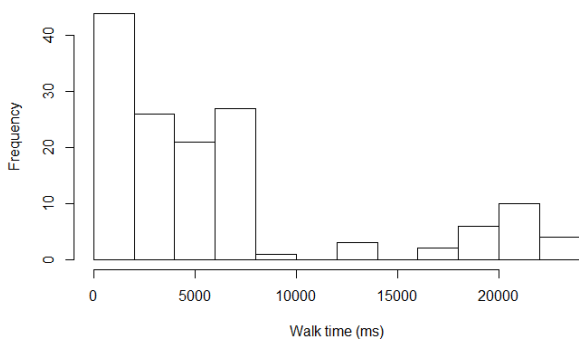


FIGURE 3. Response time for walks.

To use this functionality, the user configures a number of *polling interval buckets*, each of which represents a different polling interval and has its own thread pool. For example, a user may specify buckets for 10, 20, 60, and 100 seconds.

At boot time, all agents are assigned to the smallest polling interval bucket (in this example 10 seconds). Then, during the queries, the response time is recorded for each agent. If it is larger than its corresponding bucket time, it is moved to the next bucket (larger polling interval) to decrease the CPU load and prevent blocking in the faster bucket thread pool. Conversely, if it is lower, we have a choice of mechanisms to upgrade the agent back to a faster bucket, which will be detailed next.

A. DOWNGRADING AGENTS TO SLOWER POLLING BUCKETS

When an agent is slower in response time compared to the polling interval of its current bucket, we take an aggressive approach and immediately move it to a slower bucket as shown in Figure 4. We decided to do this because our goals are to decrease the load on the agent’s CPU as soon as possible and to make sure that fast agents are queried on time, allowing us to maintain the polling rate we wish to achieve for as many agents as possible.

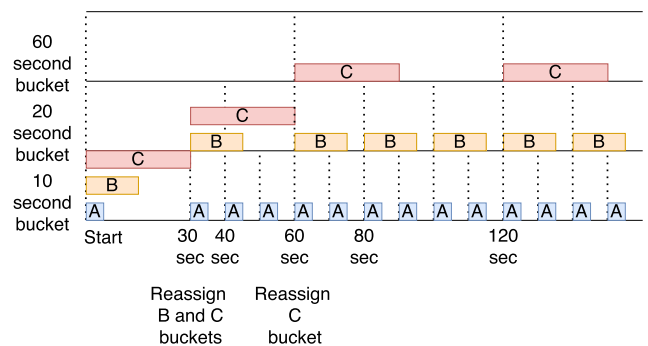


FIGURE 4. Downgrading agents to slower buckets.

For example, we may have three agents that are assigned to the 10-second polling interval bucket at the beginning. Then, it turns out that agent A took 5 seconds to respond, agent B took 15 seconds to respond, and Agent C took 30 seconds. Consequently, agents B and C will be downgraded to the 20-second bucket for the next query, while agent A stays in the 10-second bucket.

Assuming all agents always take the same time to reply in the next round, agent C (that took 30 seconds to reply) gets downgraded to the 60 seconds bucket, while agents A and B remain in the 10 and 20-second buckets respectively.

With this approach, the queries for each bucket can be performed in different threads and, as soon as they finish, a new query can be performed after the time set up for the bucket.

In our example, for each query to agent C, 3 and 6 queries can be performed in parallel to agent B and agent A, respectively.

B. UPGRADING AGENTS TO FASTER POLLING BUCKETS

When upgrading agents to faster polling buckets, we chose a more conservative approach. Instead of upgrading

it immediately, we implemented different strategies to make sure that the agent actually became faster in responding to queries.

We aim to let agents with overloaded CPUs return to an idle state and also to avoid having agents with highly variable response times constantly switching between buckets.

To find the best approach, we implemented and evaluated several algorithms to decide when to upgrade an agent to a faster polling (smaller polling interval) bucket, which will be explained next.

• **Upgrade to a faster polling bucket after *count* faster response times to queries.**

This is by far the simplest strategy. As we can see in Figure 5, each agent has a counter that counts the number of times it took the agent less time to reply than the threshold indicated by its current bucket. When the counter reaches a pre-determined *count*, it is upgraded to a faster polling bucket and the counter is reset to start the process again and, eventually, get an upgrade to a faster bucket, one at a time.

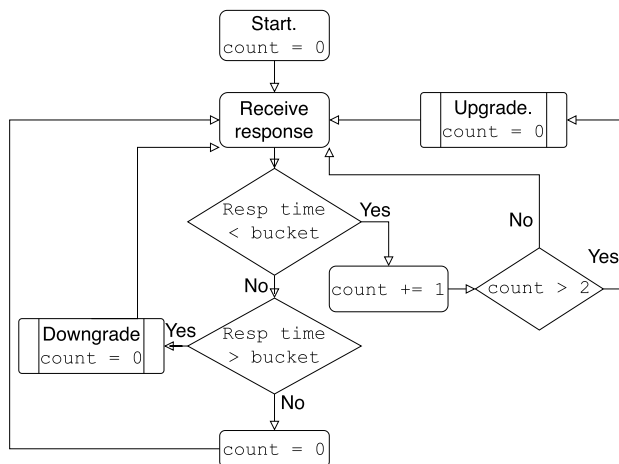


FIGURE 5. Upgrading agent to faster bucket based on count.

If the agent response time falls within the bucket it is already assigned to, the counter is reset.

Conversely, the agent is immediately moved to a slower polling rate bucket when its response time is larger than that of the bucket it is currently assigned to.

As an example, shown in Figure 6, if we have two buckets for 30 and 60 seconds, *count* is 2 and the agent is in the 60-second bucket, it has to respond in less than 30 seconds two times in a row to be upgraded to the 30-second bucket.

• **Dynamic counter**

This strategy is similar to the previous one, but instead of having a fixed counter that can only decrease, this counter can also increase when the agent is slower, as we show in Figure 7.

Thus, if an agent has been consistently slow, it should take longer for it to be upgraded to a faster bucket than if an agent has been slow in replying a small number of times.

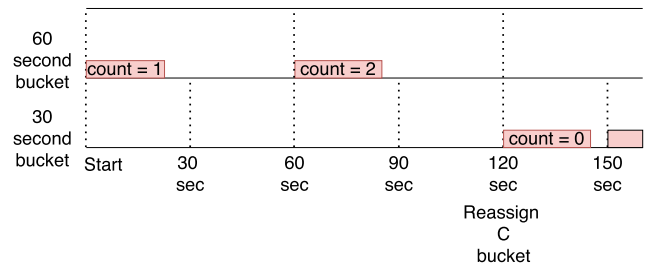


FIGURE 6. Example upgrading agent to faster bucket based on count.

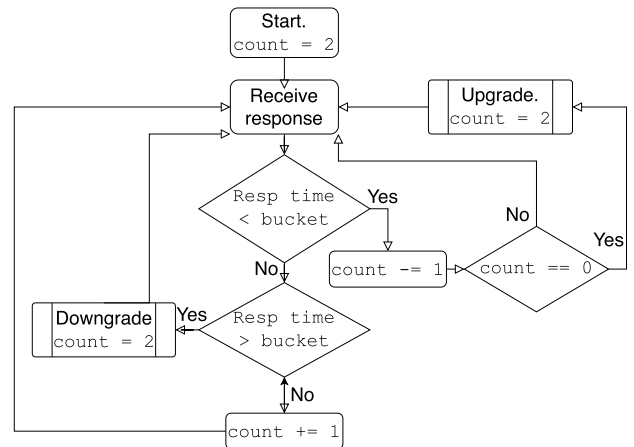


FIGURE 7. Upgrading agent to faster polling bucket based on dynamic count.

When an agent is downgraded to a slower polling bucket, the counter is set to a *minimum value* the same way as in the static counter. However, if the response time makes it stay in that bucket the counter keeps being increased up to a *maximum value*. This limit is put in place so it is possible for an agent to move to a faster bucket if, for example, the agent is slower because of higher load during peak hours.

To upgrade the agent to a faster bucket, the counter has to decrease *count* times in a row, namely, *count* smaller response times to queries or it will start increasing again. As an example, shown in Figure 8, with two buckets for 30 and 60 seconds, a *minimum counter value* of 2 and a *maximum counter value* of 5, if an agent was in the 30-second bucket but takes more than 30 seconds to respond, it will be immediately downgraded to the 60-second bucket and the counter will be initialized to 2. If the agent takes less than 30 seconds to reply two times in a row, it will be upgraded back to the 30-second bucket. However, if the agent keeps being slow in responding, the counter will be increased up to a maximum of 5. To be upgraded again, it will have to be faster than 30 seconds 5 times in a row.

• **Upgrading to a faster polling bucket after *count* seconds being faster**

Instead of counters, this algorithm uses the time an agent has been faster than its current bucket to decide when

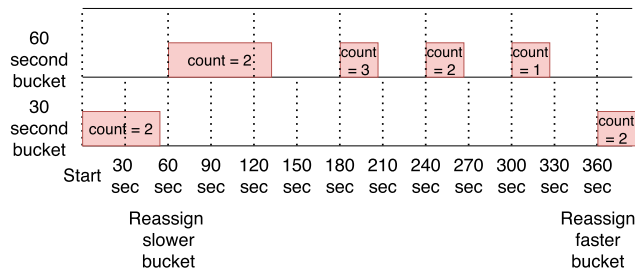


FIGURE 8. Example of upgrading agent to faster polling bucket based on dynamic count.

to upgrade it to a faster polling bucket as we show in Figure 9.

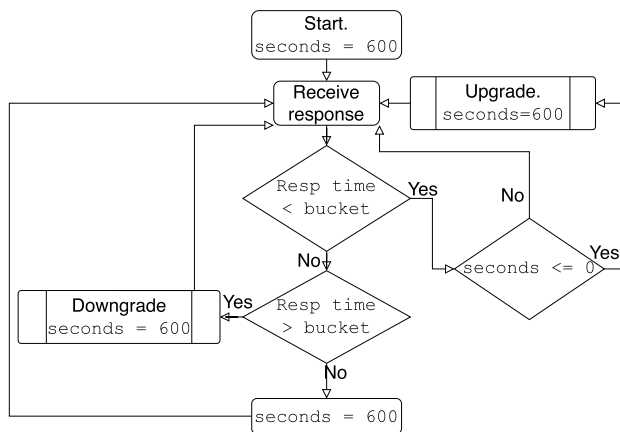


FIGURE 9. Upgrading agent to faster polling bucket after timeout.

However, an agent will be downgraded to a larger polling interval bucket if it takes longer to reply than its current bucket threshold. Whenever the agent moves up or stays in the same bucket, the current timestamp is recorded. Then, if the agent replies in less than its current bucket threshold, it must do so for a configured length of time before it can be upgraded to a smaller polling interval bucket.

For example, as shown in Figure 10, with two buckets for 30 and 60 seconds and a 10 minutes timeout, an agent

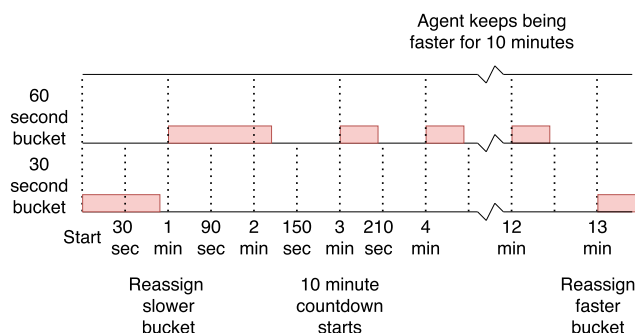


FIGURE 10. Example upgrading agent to faster polling bucket after timeout.

that is taking more than 30 seconds to reply will stay in the 60-second bucket and its timestamp will be updated every time. Eventually, if the agent response time drops to less than 30 seconds, and stays that way for 10 minutes (approximately 10 times), it will be upgraded to the 30-second bucket.

IV. PERFORMANCE EVALUATION

Performance evaluation encompasses laboratory and real-world measurement experiments, the latter carried out in a real datacenter. The purpose of the lab tests is assessing the performance limits of the software, whereas, real-world measurement campaigns not only serve to assess performance but also adequacy to industrial use cases-

A. LABORATORY MEASUREMENT EXPERIMENTS

To conduct measurement experiments in a laboratory setting, we wrote a small SNMP agent emulator capable of responding fast to queries to a table. Then, we arranged two computers connected through gigabit Ethernet as we see in Figure 11. The network manager was running in a 12-core AMD Ryzen 1600X with 32 GB of RAM and the agents were running in an 8-core Intel Core i7 7700HQ with 32 GB of RAM.

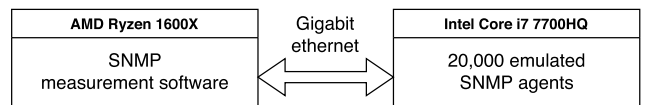


FIGURE 11. Experimental setup for 20,000 SNMP agents.

The agent receives requests and responds with hard-coded data, which allows spending very little time in processing. During the experiments, we confirmed that, even when the network manager was using all CPU resources in the first computer, the agent was using only a fraction of the resources in the second computer. This way, the agent was not preventing the network manager to increase the polling frequency whatsoever.

1) SPEEDUP WITH NUMBER OF THREADS

We performed experiments with 20,000 SNMP emulated agents, changing the number of threads in the network manager thread pool. For each experiment we measured the response time for all requests, response times for individual requests, and the average CPU load during the test. Experiments for each number of threads were repeated 5 times.

The results of these experiments are shown in Table 1, in terms of time to complete, speedup with respect to the single-thread case and cpu use. Figure 12 shows these measurements against the number of threads used for each experiment.

We note that, as we increase the number of threads, a 6X speedup was achieved for 12 threads, as we see in Figure 12, decreasing the time to query all agents from 30 to 4.6 seconds.

TABLE 1. Speedup and time to complete queries with varying number of threads.

Threads	Time to complete (s)	Speedup	CPU use (%)
1	30.04	1.00	8
2	15.01	2.00	18
4	7.94	3.78	30
6	6.05	4.97	41
8	5.17	5.81	48
10	4.79	6.27	55
12	4.61	6.52	58
16	4.58	6.56	60
20	4.69	6.41	60
24	4.69	6.41	59
30	4.71	6.38	59
35	4.74	6.34	59
40	4.74	6.34	59
50	4.73	6.35	58
60	4.70	6.39	58
80	4.73	6.35	58

number of timeouts were preventing the network manager from achieving higher speedups. In this light, we concluded that the network was the limiting factor for speedup, and not the network manager itself.

To circumvent this issue, we made the UDP buffer size larger. First, we modified buffer sizes in the network manager side, without any noticeable change in performance. Then we tried making the buffers larger on the agent side, but instead of getting better performance, the packet loss rate increased and made the problem worse.

B. REAL-WORLD MEASUREMENT CAMPAIGNS

Currently, an SNMP network manager following the above guidelines is deployed at two sites working with real-world data. In this section, we explain how it is used, challenges found, and how they were tackled.

1) LARGE-SCALE NETWORK EQUIPMENT MONITORING IN A DATACENTER

The SNMP network manager was used to monitor the status of 150 network devices, including core routers, switches, firewalls, and balancers.

We measured performance metrics, such as CPU and memory usage; network metrics, such as network traffic passing through each interface and static network information such as the configured VLANs. These collected statistics were written to text files to be processed by other tools. Our polling interval was five minutes, which is a typical polling interval in this setting.

Interestingly, this measurement campaign revealed agents taking different times to respond. Specifically, at times of the day with peak traffic, the core routers took more than ten minutes to respond, which severely distorted the measurements. We note that a five minutes time interval is rather large as the timescale of interest for traffic monitoring is decreasing. Indeed, port saturation in the order of seconds may severely affect the performance of multimedia services such as teleconferences. Such small timescale saturation epochs cannot be detected with a five minutes time interval, and even less with a ten minutes time interval.

Therefore, a root-cause analysis for these increased response times is in order, which follows next.

Finding the Root Cause of Large Response Times: As we saw in section III, the response time to SNMP walks follows a bi-modal distribution, whereby most requests are taken care of fast, but some of them can take much longer. We recorded the response times for a whole month to seek any possible pattern that can explain such slowdowns. The resulting average response time during this period is shown in Figure 13.

We observed that the plot shows several peaks where the average response time increased compared to the baseline. These peaks correspond to the weekends and after checking with the datacenter managers, we found that scheduled maintenance tasks were performed at those times.

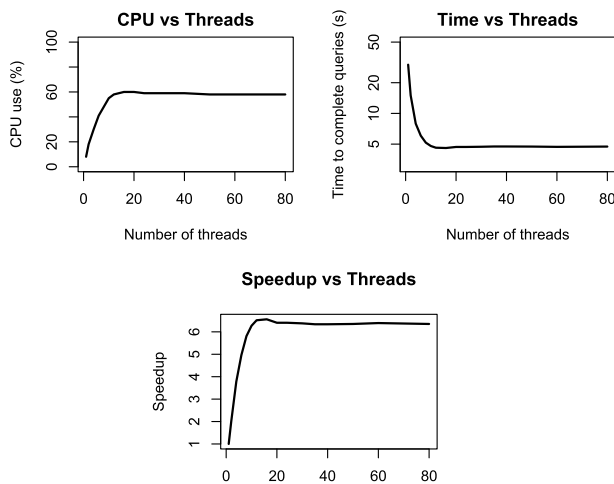


FIGURE 12. CPU, time and speedup vs threads.

However, further increasing the number of threads did not improve the results.

Initially, we hypothesize that this limitation was caused by the CPU use of the network manager. However, we found that the CPU use only went up to 60% as we see in figure 12. In the SNMP agent emulator side, we assessed that the CPU load was below 25% for all the experiments.

Next, we analyzed if the network may be causing the bottleneck, as the UDP protocol does not support congestion control and may saturate the network. Since UDP does not support retransmissions, the SNMP library is in charge of retrying requests after a configured timeout. Actually, as we increased the number of threads, we found that the SNMP library was timing out more often and retrying requests. This shows that the network was losing packets and the increasing

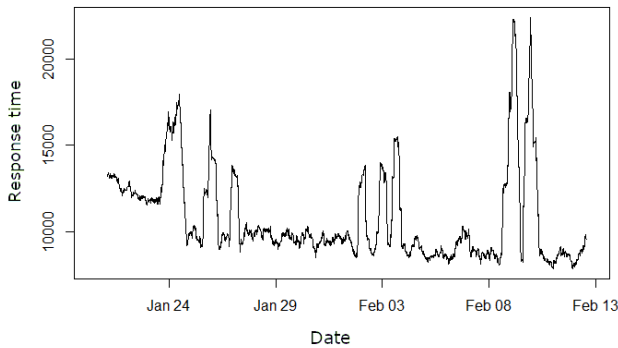


FIGURE 13. Average response time during a month.

We also hypothesized if slowdowns could be due to the network load during the maintenance windows, but we discarded that cause after checking that ping times to the affected agents did not change whenever response time increased. In addition, there are slow response times during the entire measured period as we can see in Figure 14, which shows a response time time-series, and the ping to the agents was stable during that time.

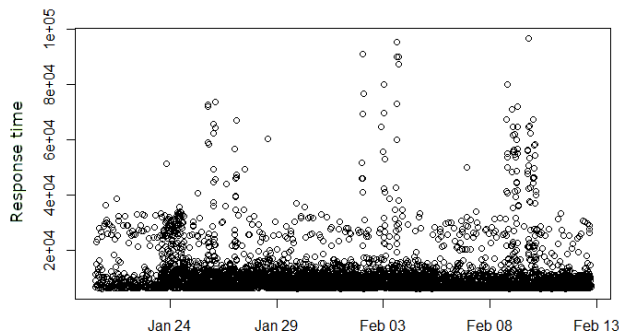


FIGURE 14. Response time during a month.

Next, we wondered if the cause may be increased CPU usage when the agents are more loaded. Luckily, the agents showing slow response times offered CPU load statistics through SNMP. To either verify or discard the former hypothesis, we carried out a measurement experiment whereby we queried a large SNMP table and, right after, queried the OID that reports CPU use.

As such, we queried *vtpVlanTable*, which contains information about VLANs and *cpmCPUTotalTable*, which contains information about CPU use. First, we checked that the response times for both tables followed the same bi-modal distribution we saw in our previous tests (as shown in Figures 15 and 16), namely large response times that happen sporadically.

Then, we derived the Pearson correlation coefficient between the response times and the CPU usages and confirmed that they were highly correlated as shown in Figures 17 and 18.

In the light of these findings, we came out with the adaptive polling algorithms presented in section III. We also note that

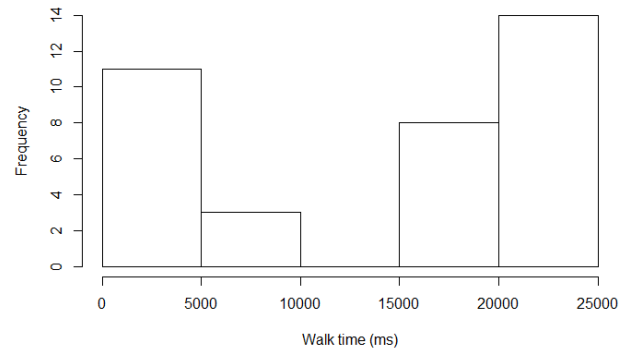


FIGURE 15. *vtpVlanTable* response time histogram.

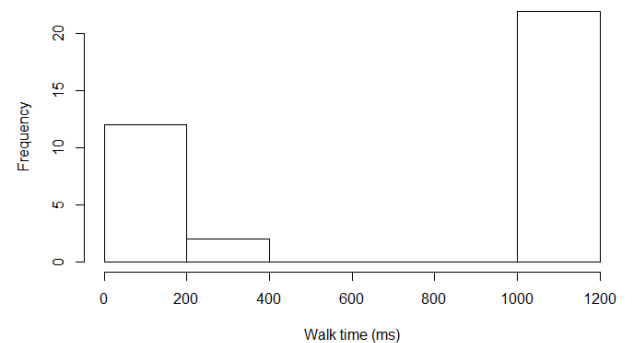


FIGURE 16. *cpmCPUTotalTable* response time histogram.

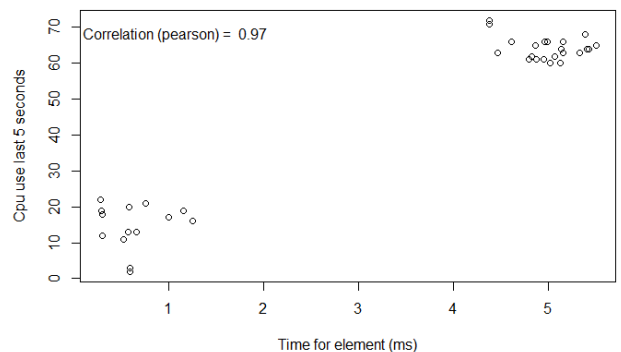


FIGURE 17. *vtpVlanTable* CPU - Response time.

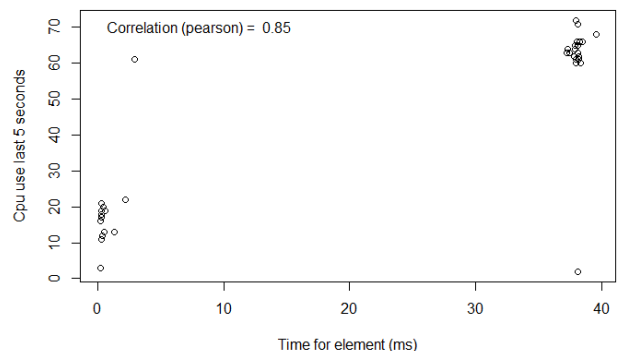


FIGURE 18. *cpmCPUTotalTable* CPU - Response time.

polling slow agents less frequently is beneficial, not to add more fuel to the fire by increasing CPU load on an already saturated device with frequent polling.

2) NETWORK TOPOLOGY IN A DATACENTER

In this case, we obtain information about interfaces and VLANs in a datacenter to infer the topology of the network. This use case is different from the previous one. There is no regular polling since the information obtained from the agents does not change often. Instead, the program is executed manually when there are changes in the datacenter hardware.

This use case motivated the incorporation of the Jython software layer to the software. As mentioned before, in order to get detailed VLAN information from some equipment, we first have to query a table with all the VLANs, then query a different table using communities with the format “community@VLAN ID”.

The first solution was implemented by using command-line scripts that queried the different tables. However, this solution did not scale well when adding more hosts, as the scripts were slow and increased in complexity. Adding the Jython layer, this solution became simpler and more efficient.

The python function used to obtain the “community@VLAN” community for each agent is shown next. This function is called for every agent after performing the first query to get the list of VLANs available on each agent. It gets as an input an agent IP and the OID of the table containing the VLAN names, then iterates over the VLAN names. For each VLAN, it creates a new *SnmPV2Agent* object where the community will have the format “community@VLAN ID” as explained earlier. These *SnmPV2Agent* objects will be used to perform further queries to obtain the desired detailed information about the VLAN.

We observe the code snippet is short and can be easily taken care of by the network analyst, instead of having to request new core functionalities to the network manager Java core developers. This is a most important feature given the complexity and heterogeneity of devices in current datacenters.

```
# Receives an agent and OID for the
# table with the VLAN IDs.
# Returns a list of SnmpAgent objects
# used to make further queries.
def get_vlan_agents_cisco(agent,
    vlan_table_oid):

    agents = []

    # Iterate over the VLAN IDs and
    # create SnmpAgent objects
    for row in agent.
        tableResults[vlan_table_oid].
            rows.keys():

        # Get VLAN id
        vlan_id = row.split('.')[1]

        # Create a new object with the
        # community in the form
        # "community@VLAN ID"
```

```
agents.append(
    SnmpV2Agent(agent.address,
        agent.name + '@' +
        vlan_id,
        agent.community + '@' +
        vlan_id))
return agents
```

C. MEASURING ADAPTIVE POLLING PERFORMANCE

In this section, we present a performance evaluation of the adaptive polling algorithm presented in section III. Our experimental setup runs in the datacenter presented in section IV-B1. More specifically, we polled 150 real network devices from a server with a 20 core Intel Xeon E5-2640 v4 @ 2.40GHz and 256 GB of RAM. We performed a throughput measurement campaign that took a whole week’s worth of measurements for each of the three schedulers, in addition to a week of measurements with no scheduler active.

For these tests we used two buckets for 30 and 60 seconds for the schedulers, as we showed in Section III.

We also used the same parameters for the schedulers as in Section III. For the Simple counter scheduler, we used a *count* value of 2. For the Dynamic counter, we used a *minimum counter value* of 2 and a *maximum counter value* of 5. Finally, we used a timeout of 10 minutes for the Timeout scheduler.

In these experiments, we focused on measuring the effectiveness of the three schedulers described in Section III. We measured response times for each monitored agent and how the scheduler moves them between time buckets.

Our main goal was to find the scheduler that minimizes the impact polling has on the agents’ CPU, since we wish to prevent overloading the agents and causing possible operational slowdowns. By downgrading the agent to a larger polling interval bucket, our schedulers ensure it will be polled less frequently, reducing the CPU load. This goal will also prevent slow agents from affecting the rest by blocking the faster bucket thread pool, allowing us to better sustain our desired polling rate of 5 minutes. We note that if an agent remains in a given bucket and the response time is actually higher than its corresponding polling interval then all agents in the bucket will suffer from polling starvation, as the new polling round will not start before the slowest agent replies. Thus, downgrading the agent to a larger polling interval bucket as soon as possible is of utmost importance.

Conversely, our secondary goal is to minimize the time an agent spends in a larger polling interval bucket, as we wish to obtain the best measurement resolution possible. We note that the former and latter goals are opposite. Additionally, some of the proposed schedulers are more aggressive than others in downgrading agents, namely are better suited to the first goal than to the second one. In this light, a performance assessment is in order, as we will present next.

Table 2 shows the percentage of time in the correct bucket for each scheduler, meaning by “correct” the ideal bucket in which agents would be placed with complete knowledge of

TABLE 2. Percentage of time the schedulers were correct or wrong.

	Correct bucket	Should have been in a slower bucket	Should have been in a faster bucket	Polling rate achieved
No scheduler	98.64%	1.36%	0.00%	98.64%
Simple counter	99.32%	0.23%	0.45%	99.77%
Dynamic counter	99.01%	0.18%	0.81%	99.82%
Timeout	99.10%	0.24%	0.66%	99.76%

their future response time beforehand. It also provides the percentage of time the agent should have been in a larger polling interval bucket, which is a figure of merit of how conservative the scheduler was in downgrading the agent. We also show the results of running the program without any scheduler active as a baseline. This means agents stay permanently in the faster bucket, with no option to go to a slower one if they take too long to respond.

As shown, all schedulers assigned agents to faster buckets than where they should have been less than 0.24% of the times, namely, they were very reactive in preventing CPU load increase. In contrast, when not using a scheduler 1.36% of all requests took longer than the faster polling interval, increasing CPU load on the agents and causing us to miss our polling interval target more often. Furthermore, the dynamic counter scheduler was more conservative in upgrading agents back to faster buckets.

The choice of a scheduler is a tradeoff between more conservative behavior in safeguarding CPU load and faster polling rates. In this regard, the simple counter provides a good balance between both.

V. CONCLUSION AND LESSONS LEARNED

In this paper, we have provided a throughout performance and scalability analysis of SNMP managers for large-scale polling. We have discussed software architectures targeted towards tackling the most important issue in large-scale polling: vertical scalability through multi-threading architectures and swift adaptation to heterogeneous data (MIBs).

We have also performed measurement campaigns in real-world datacenters, aside from laboratory experiments, that reveal slowness in response time from agents due to high CPU load epochs. Consequently, we have come up with an adaptive polling algorithm that either upgrades or downgrades agents, in terms of the polling interval, as their response time is being tracked.

Our findings serve researchers, software manufacturers, and practitioners in their quest to design cost-effective, large-scale SNMP network managers.

REFERENCES

- [1] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A simple network management protocol (SNMP)," Netw. Working Group, IETF, Tech. Rep. RFC 1157, 1990. [Online]. Available: <https://tools.ietf.org/html/rfc1157>
- [2] K. Amirthalingam and R. J. Moorhead, "SNMP—An overview of its merits and demerits," in *Proc. 27th Southeastern Symp. Syst. Theory*, Mar. 1995, pp. 180–183.
- [3] T. Song, Y. Kawahara, and T. Asami, "Cache management algorithm of load balancer for large-scale SNMP monitoring system," in *Proc. IEEE Globecom Workshops (GC Wkshps)*, Dec. 2013, pp. 901–905.
- [4] E. Magaña, L. Lefevre, M. Hasan, and J. Serrat, "SNMP-based monitoring agents and heuristic scheduling for large-scale grids," in *Proc. OTM Confederated Int. Conf. Move Meaningful Internet Syst.* Springer, 2007, pp. 1367–1384. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-76843-2_17
- [5] J. Schonwalder, "Characterization of SNMP MIB modules," in *Proc. 9th IFIP/IEEE Int. Symp. Integr. Netw. Manage. IM*, May 2005, pp. 615–628.
- [6] S. S. Chavan and R. Madanagopal, "Generic SNMP proxy agent framework for management of heterogeneous network elements," in *Proc. 1st Int. Commun. Syst. Netw. Workshops*, Jan. 2009, pp. 1–6.
- [7] Net-SNMP. (2020). *Net-SNMP*. [Online]. Available: <http://www.net-snmp.org/>
- [8] *Global Interpreter Lock*, ThomasWouters, Amsterdam, The Netherlands, 2017.
- [9] AgentPP. (2020). *SNMP++*. [Online]. Available: https://agentpp.com/api/cpp/snmp_pp.html
- [10] M. Verdi, A. Sami, J. Akhondali, F. Khomh, G. Uddin, and A. K. Motlagh, "An empirical study of C++ vulnerabilities in crowd-sourced code examples," 2019, *arXiv:1910.01321*. [Online]. Available: <http://arxiv.org/abs/1910.01321>
- [11] AgentPP. (2020). *SNMP4J*. [Online]. Available: <https://agentpp.com/api/java/snmp4j.html>
- [12] Jython. (2020). *Jython*. [Online]. Available: <https://www.jython.org/>



PAULA ROQUERO received the M.Sc. degree in computer science from the Universidad Autónoma de Madrid, Spain, in 2016. She then joined the High Performance Computing and Networking Research Group, Universidad Autónoma de Madrid. Her current research interests include as a Ph.D. student in distributed systems and network traffic analysis.



JAVIER ARACIL received the M.Sc. and Ph.D. degrees (Hons.) in telecommunications engineering from the Technical University of Madrid, in 1993 and 1995, respectively. In 1995, he was awarded with a Fulbright Scholarship. He was also appointed as a Postdoctoral Researcher with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. In 1998, he was a Research Scholar with the Center for Advanced Telecommunications, Systems and Services, The University of Texas at Dallas. He has been an Associate Professor with the University of Cantabria and the Public University of Navarra. He is currently a Full Professor with the Universidad Autónoma de Madrid, Madrid, Spain. He has authored more than 100 papers in international conferences and journals. His research interests include optical networks and performance evaluation of communication networks.

...