Evaluation of the Resource Requirements of SNMP Agents on Constrained Devices

Siarhei Kuryla, Jürgen Schönwälder

Computer Science, Jacobs University Bremen, Germany
{s.kuryla,j.schoenwaelder}@jacobs-university.de

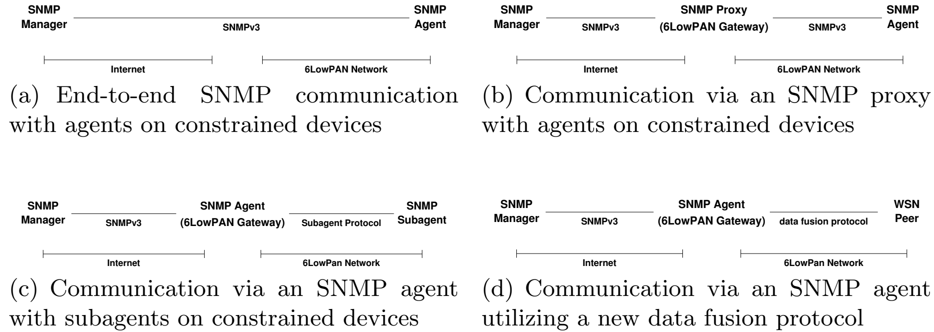
Abstract. Constrained devices equipped with a microcontroller and a low-power low-bitrate wireless interface are becoming part of the Internet. We investigate whether the monitoring and configuration of such constrained devices can be performed by adapting the Simple Network Management Protocol (SNMP) to the capabilities of these devices. To this end, we have implemented an SNMP agent under the Contiki operating system. We provide an analysis of its resource requirements and its runtime behaviour on an 8-bit AVR Raven platform.

Key words: SNMP, 6LoWPAN, Contiki, Internet of Things

1 Introduction

The Simple Network Management Protocol (SNMP) [1] is widely deployed to monitor, control, and sometimes also configure network elements. Even though the SNMP technology is well documented and well understood, it remains unclear what the exact resource requirements are of running SNMP on constrained devices, such as an 8-bit microcontroller with 16 kB of RAM connected to the Internet via an IEEE 802.15.4 transceiver. The origins of SNMP date back to the late 1980s when computers had much less resources compared to what we are used to today. In fact, one of the stated goals was that "the impact of adding network management to managed nodes must be minimal, reflecting a lowest common denominator" [2]. From this historic perspective, SNMP seems to be a reasonable fit for managing today's constrained devices. However, it must be noted that SNMP did evolve during the 1990s and in particular security mechanisms present in SNMP version 3 (SNMPv3) add significant complexity, increasing the code size and impacting runtime performance. Hence, we were approached with question such as the following:

- What are the resource requirements of a minimal SNMPv3 implementation running on constrained devices?
- Which parts of an SNMP protocol engine are most expensive?
- What is the cost of adding instrumentation (additional MIB objects)?
- What is the runtime behaviour of SNMP over an IPv6 link using an IEEE 802.15.4 radio and the 6LoWPAN adaptation layer?



(a) End-to-end SNMP communication with agents on constrained devices

(b) Communication via an SNMP proxy with agents on constrained devices

(c) Communication via an SNMP agent with subagents on constrained devices

(d) Communication via an SNMP agent utilizing a new data fusion protocol

Fig. 1. Architectural options for using SNMP in 6LoWPAN networks

The answers to these questions are crucial in order to understand whether it is feasible to run multiple protocols concurrently on constrained devices providing end-to-end interoperability with deployed systems or whether it is necessary to adopt architectures, where interoperability with deployed systems is achieved via gateways translating between standard Internet protocols and a single protocol (e.g., CoAP [3]) interfacing constrained devices.

The rest of the paper is structured as follows. We first discuss some architectural options for using SNMP in constrained networks in Section 2 before we review related work in Section 3. The design choices behind our SNMP implementation running on the Contiki [4] operating system are summarized in Section 4. In Section 5, we describe our experimental setup in. We present an analysis of the memory requirements in Section 6, which is followed by a discussion of the observed latency in Section 7. We conclude the paper in Section 8.

2 Architectural Considerations

Low-Power Wireless Personal Area Networks (LoWPANs) typically consist of a (potentially large) number of constrained devices embedded into everyday objects. Unlike conventional networks, nodes in such networks should need minimal configuration, they should preferably work “out of the box”, they should be easy to bootstrap, and they should be largely self-healing [5]. However, even the best automated mechanisms may fail and require explicit management once in a while. As such, a certain amount of explicit management can never be completely removed. Since the goal of IPv6 over LoWPANs (6LoWPAN) is to reuse existing protocols as much as possible, it makes sense to look at the question how SNMP can be used to manage 6LoWPAN networks consisting of constrained devices. Figure 1 outlines four architectural options for the common scenario where a network management system residing in the normal Internet manages constrained devices connected via a 6LoWPAN network.

Figure 1(a) assumes direct end-to-end SNMP communication. This option provides straight forward access to individual 6LoWPAN nodes. Reuse of existing

deployed SNMP-based tools is easy and end-to-end security can be provided. The downsides of this option are related to message sizes and fragmentation issues, the requirement to embed a full SNMP engine into constrained devices, and the trap-directed polling nature of SNMP if energy consumption is a concern.

By utilizing an SNMP proxy (see Figure 1(b)), it is possible to optimize the transport of SNMP messages on the 6LoWPAN network, e.g., by using an alternate encoding or by using different security mechanisms. Since SNMP proxies are well defined in the SNMP specifications, management applications supporting SNMP proxies should need no modifications. Note that this approach still requires an SNMP agent on the constrained devices and that it does not overcome the trap-directed polling nature of SNMP.

Figure 1(c) outlines the usage of SNMP subagent technology where a single SNMP agent, typically running on an edge router, provides access to management information, utilizing a special purpose subagent protocol to interact with subagents residing on constrained devices. By tailoring the subagent protocol to the constraints of the link and the devices, it is possible to improve efficiency in the 6LoWPAN network and some amount of caching of data in the SNMP agent becomes feasible. However, this approach is not transparent to the management system since management applications need to address management information by using specific SNMP contexts. The fact that SNMP security ends at the SNMP agent residing at the edge router may be considered a feature or a problem, depending on the deployment scenario. The subagent option only has a clear advantage if the implementation costs of subagents is significantly lower than the costs for implementing an SNMP agent.

Finally, Figure 1(d) develops the subagent approach further by introducing a new protocol enabling data fusion in the 6LoWPAN network, e.g., in-network data aggregation. This approach requires not only new protocols, but also new management applications that are able to properly address management information and that understand the semantics of aggregated management data. On the other hand, communication in the 6LoWPAN network can be optimized and in particular tree-like routing structures can be exploited by aggregating data as it travels towards the edge router.

3 Related Work

During the past decade, a large number of papers discussing various performance aspects of SNMP have been published [6]. The majority of these papers assume high bandwidth network links and PC-like end devices. In this section, we focus solely on recent work dealing with 6LoWPAN networks and constrained devices.

In 2008, Hamid Mukhtar et al. [7] proposed a LoWPAN Network Management Protocol (LNMP). Their approach aims at a reduction of communication cost in order to increase the network lifetime and to avoid congestion. In the proposed architecture, SNMP is supported on the native IPv6 network side only and a 6LoWPAN gateway acts as a subagent proxy, translating between the SNMP protocol and the protocol used on the 6LoWPAN network. Their work

therefore follows the subagent model shown in Figure 1(c). Whenever an SNMP request arrives at the gateway, it is translated from SNMP to a simplified query format. The gateway sends a query (via UDP) that contains identifiers of the objects to be retrieved from the agent residing on the destination device. Similarly, when a reply from the device arrives at the gateway, it is translated back into the SNMP protocol format and an SNMP response packet is sent back to the SNMP engine that requested the object. The gateway is also responsible for responding to requests for objects whose values are constant for the whole network. The proposed LNMP protocol was implemented for the Atmel ATmega 128L microcontroller. Unfortunately, the query protocol between the gateway is not further detailed and no details are given concerning the code size or the memory requirements. Furthermore, the proposal appears to be incomplete since protocol security has not been considered nor has the addressing through SNMP contexts and the OID mapping been fully worked out.

Haksoo Choi et al. describe two modifications of SNMP in order to optimize SNMP for resource-constrained low-power and low data-rate wireless networks [8]. The first change is the introduction of compressed SNMPv1 and SNMPv2c message headers and the usage of SNMP delta compression [9]. The second modification is the introduction of new protocol operations to push management information periodically from an SNMP agent towards an SNMP manager and the usage of UDP multicasts. In order to interface existing SNMP applications with their extended version of SNMP, they introduce an SNMP proxy that performs the necessary translations. From an architectural viewpoint, this work follows the proxy model shown in Figure 1(b). The work by Haksoo Choi et al. has two major limitations: (i) The work is limited to SNMPv1 and SNMPv2c and does not consider protocol security. In fact, the elimination of community strings on the 6LoWPAN network causes SNMP agents to respond to any request. (ii) The push mechanism only works for non-dynamic MIB objects since it relies on Get-operations. Periodic retrieval of data in dynamically changing tables is thus not optimized. Multicasting has the same limitations and becomes rather complex if protocol security mechanisms would be considered. The modified SNMP protocol was implemented using the TinyOS operating system [10] on the Tmote Sky platform featuring the 16-bit TI MSP430 microcontroller with 10 kB of RAM and 48 kB of flash memory.

4 Implementation

Our Contiki SNMP implementation supports the SNMPv1 and SNMPv2 message processing models and the `Get`, `GetNext` and `Set` protocol operations. The USM security model has been implemented. It supports the HMAC-MD5-96 authentication and CFB128-AES-128 symmetric encryption protocols. The standard specifies that SNMP entities must accept messages up to at least 484 bytes in size, which is the maximum message size supported by our implementation.

One of the goals of our implementation is memory efficiency. However, whenever a choice had to be made between easily readable and maintainable source

code and memory efficient but hard to understand programming constructs, we gave preference to the readable code. Due to the small amount of available RAM memory, necessary data structures were optimized and designed very carefully. Since the amount of ROM memory provided by the hardware platform is significantly larger than the amount of RAM, ROM memory has been preferred for storing read-only data.

```
typedef struct {
  u8t    version;
  pdu_t  pdu;
} message_t;

typedef struct {
  u8t                  request_type;
  u8t                  response_type;
  s32t                 request_id;
  u8t                  error_status;
  u8t                  error_index;
  varbind_list_item_t *varbind_first_ptr;
  u16t                 varbind_index;
} pdu_t;

typedef struct varbind_list_item_t {
  varbind_t                    varbind;
  struct varbind_list_item_t*  next_ptr;
} varbind_list_item_t;

typedef struct varbind_t {
  ptr_t*             oid_ptr;
  u8t                value_type;
  varbind_value_t    value;
} varbind_t;

typedef union {
  s32t        i_value;
  u32t        u_value;
  ptr_t       p_value;
} varbind_value_t;
```

Fig. 2. Key data structures representing SNMP messages

Some key data structures are shown in Figure 2. In the implementation, OID values are never decoded but instead natively stored in ASN.1/BER format. This not only enables a memory efficient representation of OID values, it also allows to refer to the OID of a varbind by pointing inside the received packet, thus reducing the need for runtime memory allocations.

MIB objects are internally represented by instances of the `mib_object_t` structure shown in Figure 3. The `attrs` member specifies the attributes of the object. The `varbind` member holds the OID of the object and its value. The `get_fnc_ptr` and `set_fnc_ptr` members are pointers to the user-defined getter and setter functions respectively. In case of enabled tabular objects, two additional bytes are required for the `get_next_oid_fnc_ptr` pointer for every managed object. Tabular objects are an optional feature and they can be disabled by changing the value of the `ENABLE_MIB_TABLE` macro definition.

The OID of a managed object is an array of bytes which does not change its value over time. By default, such arrays are handled as all other initialized variables: they occupy RAM and they occupy the same amount of flash ROM so they can be initialized to the actual value by startup code. This is a waste of RAM, which is the most critical resource of today's constrained devices. Of course, such data can be moved to flash ROM. However, to access the data stored in the flash ROM special functions have to be used, which results in additional complexity. The implementation allows to store OIDs either in RAM or flash ROM. This can be defined at compile time, by switching the value of the `ENABLE_PROGMEM` macro definition between `0` and `1`.

```
struct mib_object_t {
  u8t attrs;
  varbind_t varbind;
  get_value_t get_fnc_ptr;
#if ENABLE_MIB_TABLE
  get_next_oid_t get_next_oid_fnc_ptr;
#endif
  set_value_t set_fnc_ptr;
#ifndef MIB_SIZE
  struct mib_object_t* next_ptr;
#endif
};
```

Fig. 3. MIB data structures representing MIB objects

Our SNMP implementation supports two ways to organize the storage of managed objects by using either an array or a linked list. In the first case, the number of objects has to be predefined at compile time by using the `MIB_SIZE` macro definition, otherwise a linked list is used. In the latter case, every managed object requires two extra bytes for the **`next_ptr`** member.

Several objects of the `SNMPv2-MIB` [11], `IF-MIB` [12] and `ENTITY-SENSOR-MIB` [13] have been implemented. The `SNMPv2-MIB` defines managed objects describing the identity and the capabilities of an SNMP entity. The `IF-MIB` provides access to information related to network interfaces. It exposes counters of packets received on and transmitted out an interface. In order to obtain such statistical data, the Contiki radio driver has been instrumented. The managed objects of the `ENTITY-SENSOR-MIB` provide access to physical sensors. The readings of the temperature sensor can be obtained via the objects of this module.

5 Hardware Platform and Experimental Setup

The AVR Raven board includes two microcontrollers (MCUs), a radio transceiver chip and an LCD display. The ATmega1284PV MCU runs the communication while the LCD display is driven by the ATmega3290PV. The wireless communication is enabled by the AT86RF230 transceiver. Both the ATmega1284PV and the ATmega3290PV are modified Harvard architecture 8-bit RISC single chip MCUs. The ATmega1284PV runs at 20 MHz and has 16 kB of SRAM, 128 kB of flash program memory and 4 kB of EEPROM. It embeds two 16-bit timers, two 8-bit timers and one real time counter. The ATmega3290PV runs at 16 MHz and has 2 kB of SRAM, 32 kB of flash memory and 1 kB of EEPROM. The AT86RF230 is a 2.4 GHz radio transceiver targeted for IEEE 802.15.4 and 6LoWPAN applications. It supports automatic frame acknowledgement and retransmission, automatic CSMA-CA and data transfer speeds of up to 250 kbps.

The network setup used consists of a PC equipped with an 802.15.4 interface connected via the USB bus. The AVR RZUSBstick, a USB stick with a 2.4 GHz IEEE 802.15.4 transceiver, is used to provide the PC with an 802.15.4 interface. The software running on the MCU on the USB stick is doing the 6LoWPAN adaptation and hence no software changes are needed on the PC.

The Contiki SNMP implementation has been tested for interoperability with the `snmpget`, `snmpgetnext`, `snmpset` and `snmpwalk` applications from the Net-SNMP[1] suite. All measurements reported in this paper were made with Contiki 2.4 using the IPHC header compression mechanism.

6 Memory and Code Footprint

Due to the limited memory resources of constrained devices, the RAM and flash ROM used by the agent are important parameters that have to be evaluated. However, RAM usage is hard to measure because of the variable size of the stack and the heap used for dynamic memory allocation. In this section, we present memory usage estimations obtained by using three different approaches.

6.1 Flash ROM and Static Memory Usage

In the first step, the `avr-size` utility was employed to determine the flash ROM and static memory usage. We are interested in measuring the memory used only by the agent and not by the whole program that also includes the Contiki operating system. To achieve this, we first compile the source code of Contiki with the agent and measure the size of the output object file using the `avr-size` utility. Then, the same procedure is repeated for Contiki without the agent. The total memory used by the agent is obtained by a simple subtraction. The full implementation uses 31220 bytes of ROM, which is around 24% of the available ROM on the targeted platform, and 235 bytes of statically allocated RAM. In case SNMPv1 is only enabled, the agent uses 8860 bytes of ROM (about 7% of available ROM) and 43 bytes of statically allocated RAM. In a similar way, we measured the code and memory footprint of each module of the agent. Table 1 shows the detailed breakdown of the measurements.

As can be seen from the table, the cryptographic primitives occupy a significant amount of flash ROM. The AES and MD5 implementations constitute around 31% and 33% respectively of the agent code size. Almost half of the ROM occupied by the AES implementation is used to store constants. The MD5 implementation intensively uses macro definitions for transformations, which results in the large code size. Using functions instead of macros could reduce the code size, but would negatively impact the performance. It is worth to mention that the cryptographic primitives were ported from the OpenSSL library and that they are not optimized for the instruction set of the MCU.

The USM security model occupies almost half of the agent's statically allocated RAM. This RAM is mostly used to store localized keys and OIDs of the error indication counters.

6.2 Stack Usage

An experimental approach has been taken to estimate the stack size used by the agent while processing a request. Upon receipt of an incoming SNMP message,

[1] `http://www.net-snmp.org/`

Table 1. Flash ROM and static RAM memory usage of the agent (bytes)

Module	Flash ROM	RAM (static)
snmpd.c	172	2
dispatch.c	1076	26
msg-proc-v1.c	634	6
msg-proc-v3.c	1184	30
cmd-responder.c	302	0
mib.c	1996	6
ber.c	4264	3
usm.c	1160	122
aes_cfb.c	9752	40
md5.c	10264	0
utils.c	416	0

the memory region allocated to the program stack is filled with a specific bit pattern. When the processing has been finished, the stack is examined to see how much of it is overwritten.

Table 2. Maximum runtime stack size usage (bytes)

Version	Security level	Max. stack size
v1	–	688
v3	noAuthNoPriv	708
v3	authNoPriv	1140
v3	authPriv	1144

Table 2 presents the maximum stack size observed during experiments for different versions of SNMP and different selected security levels. Most of the stack is occupied by the response message buffer of 484 bytes. The SNMPv1 and SNMPv3 protocol versions with the `noAuthNoPriv` security level use approximately the same stack size, which constitutes around 4% of the available RAM. When authentication and privacy are enabled, the stack grows up to 1144 bytes, which is about 7% of RAM on the targeted platform.

6.3 Heap Usage

Memory for the data structures used to store the fields of an SNMP message (see Section 4) is allocated from the heap using the `malloc()` function. Table 3(a) provides a memory estimation for the `message_t` structure.

Each variable binding is stored in a linked list using an instance of the `varbind_list_item_t` structure. In addition to the memory calculation shown in Table 3(c), a variable binding uses 4 additional bytes for an instance of the `ptr_t` structure, which points to the OID stored in the message buffer.

Table 3. Memory usage of key data structures

(a) Memory estimation for the `message_t` structure (13 bytes)

Member	Size (bytes)
version	1
pdu.request_type	1
pdu.response_type	1
pdu.request_id	4
pdu.error_status	1
pdu.error_index	1
pdu.varbind_first_ptr	2
pdu.varbind_index	2

(b) Memory estimation for the `mib_object_t` structure (16 bytes)

Member	Size (bytes)
attrs	1
get_fnc_ptr	2
set_fnc_ptr	2
get_next_oid_fnc_ptr	2
next_ptr	2
varbind.oid_ptr	2
varbind.value_type	1
varbind.value	4

(c) Memory estimation for a variable binding (9 bytes)

Member	Size (bytes)
next_ptr	2
varbind.oid_ptr	2
varbind.value_type	1
varbind.value	4

The overall memory utilized to store an SNMPv1 message with `N` variable bindings is given by the formula $13 + N(9+4) = 13(N+1)$. The SNMP message size is limited in the implementation to 484 bytes. Each variable binding encoded in the BER format requires at least 7 bytes, therefore, in the worst (albeit unrealistic) case an SNMPv1 message may carry $484/7 = 69$ variable bindings, which would require $13(69+1) = 910$ bytes of the heap to store such a message.

6.4 Managed Objects

As discussed in Section 4, each managed object is stored in memory as an instance of the `mib_object_t` structure. Table 3(b) presents a memory estimation for the members of this structure. A managed object uses additional $4+L$ bytes for the OID, where L is the length in bytes of the OID encoded in the BER format. In case a managed object is of a string-based type, S extra bytes are required to store its value, where S is the length of the value. The total RAM usage for a managed object is given by the formula $16 + (4+L) + S$. Using flash ROM to store OIDs allows to save $4+L$ bytes of RAM for every managed object. In this case, the formula changes to $16 + S$, which would require 1600 bytes of RAM for 100 managed objects of an integer-based type.

7 Response Latency

A simple `udp-echo` application has been developed for Contiki, which allows to obtain a round-trip time estimation for a UDP datagram of a certain size. Table 4

Table 4. Round-trip time (RTT) measured with `udp-echo`

Payload (bytes)	802.15.4 frames	RTT (ms)	Variance
1	1	30.75	2.07
90	1	38.87	6.58
91	2	49.90	6.54
175	2	58.93	6.63

provides some experimental results obtained using `udp-echo`. The results reveal that the transmission time of an 802.15.4 frame depends significantly on the amount of data being sent. For example, the difference between the round-trip time for datagrams with a payload of 1 byte and 90 bytes, both of which fit into one 802.15.4 frame, is around 8 ms. The delay is caused by the low speed of the radio transceiver. When sending a UDP packet with a 91 byte payload, 6LoWPAN fragmentation is used and, as expected, we observe an abrupt increase in the round-trip time compared to the 90 byte payload packet.

In order to estimate the SNMP request processing time taken by the agent, the request-response latency for individual messages was measured at the gateway by noting the delay between sending of the message and receiving of the response. In addition to the actual processing time, the measured delay also includes the time to transmit messages over the air between the gateway and the agent. If a message does not fit into a single 802.15.4 frame, the 6LoWPAN fragmentation takes place, which causes additional overhead as explained above. The SNMP request processing time can be found as the difference between the the SNMP request-response latency and the round-trip time estimated with `udp-echo` for datagrams with the same payload length.

Table 5 and Figure 4(a) present latency measurements for SNMPv1 and SNMPv3 with three different security levels. All measurements were obtained for requests with one variable binding referring to the same MIB object. The first observation is that the time spent in the SNMP request processing is small relative to that spent in data transfer for SNMPv1 and SNMPv3 in the `noAuthNoPriv` security level. The SNMP processing constitutes only around 6-7% of the total latency. As expected, the usage of the authentication protocol results in a significant increase of this metric. The results also reveal that encryption does not have that much impact on the processing time as the authentication does. It is important to mention that the measurements for SNMPv3 do not include the discovery procedure, which would result in an additional message exchange.

Figure 4(b) shows changes in the processing time by varying the number of variable bindings in a request. These measurements were accomplished for the SNMPv1 protocol and the `Get` operation. The processing time varies from 2 to 19 ms, which is not significant when compared to the transfer time. Even though the object lookup time depends on its position in the MIB, for this targeted platform, it is unlikely that the MIB will contain that many objects to change the results considerably.

Table 5. Experimental results obtained by measuring the response latency (presented in the latency and variance columns) for SNMP requests. The round-trip time (RTT) is estimated using `udp-echo`. The last column is the processing time taken by the agent. All time measurements are given in milliseconds

Version	Operation	Security level	Latency	Variance	RTT	Δ
v1	`Get`	–	37.05	5.69	34.70	2.35
v1	`GetNext`	–	36.98	6.58	34.70	2.28
v1	`Set`	–	37.14	4.39	34.70	2.44
v3	`Get`	`noAuthNoPriv`	56.64	4.05	52.62	4.02
v3	`GetNext`	`noAuthNoPriv`	56.58	2.72	52.62	3.96
v3	`Set`	`noAuthNoPriv`	56.78	3.00	52.62	4.16
v3	`Get`	`authNoPriv`	91.41	3.45	53.02	38.39
v3	`GetNext`	`authNoPriv`	91.95	3.75	53.02	38.93
v3	`Set`	`authNoPriv`	92.41	3.22	53.02	39.39
v3	`Get`	`authPriv`	105.70	5.38	55.03	50.67
v3	`GetNext`	`authPriv`	106.46	2.59	55.03	51.43
v3	`Set`	`authPriv`	106.73	3.59	55.03	51.70

8 Conclusions

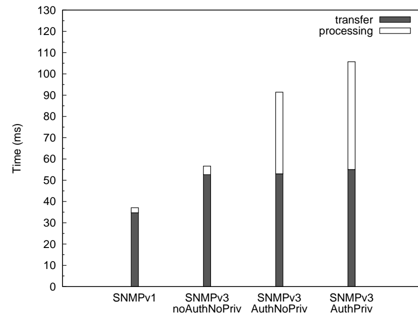
An implementation of the Simple Network Management Protocol for resource constrained devices under the Contiki embedded operating system has been presented. The implementation is modular and extensible by design. It supports the `Get`, `GetNext` and `Set` operations, the SNMPv1 and SNMPv3 message processing models and the User-based Security Model (with the HMAC-MD5-96 authentication and the CFB128-AES-128 symmetric encryption protocols). The implementation provides an interface to define and configure accessible managed objects. A couple of the existing MIB modules have been implemented as part of the agent.

The evaluation of the implementation has been carried out on the AVR Raven hardware platform. The experimental results reveal that the request processing time for SNMPv1 and SNMPv3 with the `noAuthNoPriv` security level is relatively small compared to the transfer time. Using the authentication protocol results in a significant increase of this metric, while the encryption protocol does not have that much impact on it. The RAM and flash ROM usage has been estimated by using three different approaches.

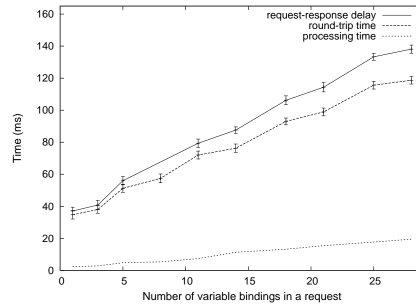
Possible further work would be to implement the `GetBulk` operation, notifications (`Trap` and `Inform`), and the SNMPv2c message processing model. It is also possible to use this platform to develop and prototype additional MIB objects for 6LoWPAN networks and related protocols.

References

1. Case, J., Mundy, R., Partain, D., Stewart, B.: Introduction and Applicability Statements for Internet Standard Management Framework. RFC 3410 (December



(a) Time taken for transferring and processing an SNMP request with different message processing models and security levels



(b) Time spent in transferring and processing SNMPv1 requests and responses as a function of the number of variable bindings in a request

2002)
2. Rose, M.T.: The Simple Book – An Introduction to Management of TCP/IP based Internets. Prentice Hall (1991)
3. Shelby, Z., Hartke, K., Bormann, C., Frank, B.: Constrained Application Protocol (CoAP). Internet-Draft <draft-ietf-core-coap-05>, (March 2011)
4. Dunkels, A., Gronvall, B., Voigt, T.: Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: Proc. 29th IEEE International Conference on Local Computer Networks (LCN'04). (2004)
5. Kushalnagar, N., Montenegro, G., Schumacher, C.: IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919 (August 2007)
6. Andrey, L., Festor, O., Lahmadi, A., Pras, A., Schönwälder, J.: Survey of SNMP Performance Analysis Studies. International Journal of Network Management **19**(6) (2009) 527–548
7. Mukhtar, H., Kang-Myo, K., Chaudhry, S.A., Akbar, A.H., Ki-Hyung, K., Yoo, S.W.: LNMP - Management architecture for IPv6 based low-power Wireless Personal Area Networks (6LoWPAN). In: Proc. 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), IEEE (April 2008) 417–424
8. Choi, H., Kim, N., Cha, H.: 6LoWPAN-SNMP: Simple Network Management Protocol for 6LoWPAN. In: Proc. 11th IEEE International Conference on High Performance Computing and Communications, IEEE (2009) 305–313
9. Schönwälder, J.: SNMP Payload Compression. Internet Draft <draft-irtf-nmrg-snmp-compression-01.txt>, (April 2001)
10. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC Language: A Holistic Approach to Networked Embedded Systems. In: Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI03), ACM (June 2003)
11. Presuhn, R.: Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP). RFC 3418 (December 2002)
12. McCloghrie, K., Kastenholz, F.: The Interfaces Group MIB. RFC 2863 (June 2000)
13. Bierman, A., Romascanu, D., Norseth, K.C.: Entity Sensor Management Information Base. RFC 3433 (December 2002)