JACOBS
UNIVERSITY

Implementation and Evaluation of the Simple Network Management Protocol over IEEE 802.15.4 Radios under the Contiki Operating System

by

Siarhei Kuryla

A thesis for the conferral of a Master of Science in Smart Systems

Superviser: Prof. Dr. Jürgen Schönwälder
Second Reader: Dr. Bendick Mahleko

Date of Submission: July 22nd, 2010

School of Engineering and Science
Jacobs University Bremen gGmbH
Campus Ring 1
28759 Bremen
Germany

Declaration

This thesis is the result of my own independent work and has not previously been accepted in substance for any degree and is not concurrently submitted in candidature for any degree.

This thesis is being submitted in fulfillment of the requirements for the degree of Master of Science in Smart Systems.

.....

Siarhei Kuryla

Acknowledgments

I would like to first of all thank my supervisor, Prof. Dr. Jürgen Schönwälder, for his continuous guidance throughout this thesis and for many helpful comments.

I also extend a warm thank you to Tatsiana Zhytsinets, my family and friends, who put up with my busy schedule over several months.

Thank you!

Abstract

Low-power wireless personal area networks (LoWPANs), composed of a large number of tiny devices with wireless communication capabilities, are becoming increasingly important due to their reduced cost and a range of real world applications. Devices in such networks are expected to be deployed in exceedingly large numbers. Furthermore, they are expected to have limited display and input capabilities. This makes network management functionality critical for LoWPANs. The Simple Network Management Protocol (SNMP) is a widely deployed application protocol for network management and data retrieval. In the course of this thesis, an SNMP agent for devices supporting IEEE 802.15.4 Radios and the Contiki Embedded Operating System has been developed. The evaluation of the implementation has been performed on the AVR Raven board hardware platform.

Contents

Chapter 1

Introduction

A low-power wireless personal area network (LoWPAN) is a simple low cost communication network that allows wireless connectivity in applications with limited power and relaxed throughput requirements. LoWPANs conform to the IEEE 802.15.4 standard [40] and typically include devices limited in their computational power, memory, and energy availability. Such networks can benefit from IP and, in particular, IPv6 networking. IP-based devices can be easily connected to other IP-based networks by reusing existing infrastructure. However, due to the limited nature of IEEE 802.15.4 devices the IPv6 protocol [13] needs to be adapted for its use in LoWPANs. Devices within LoWPANs are expected to be deployed in exceedingly large numbers. Furthermore, they are expected to have limited display and input capabilities. This makes network management functionality critical for LoWPANs. The Simple Network Management Protocol (SNMP) is a widely deployed application layer protocol for network management and data retrieval. SNMP is datagram-oriented and the implementations of SNMP can be very lightweight. It may fit LoWPAN applications very well. The purpose of this thesis is to investigate whether the SNMP functionality may be reused "as is" in LoWPANs.

The rest of the thesis is organized as follows. Chapter 2 provides an introduction to the IEEE 802.15.4 standard. Chapter 3 describes the adaptation layer that allows the transmission of IPv6 packets over 802.15.4 links. Chapter 4 gives an overview of the Simple Network Management Protocol. A short overview of the current approaches on network management in 6LoWPANs is given in Chapter 5. Chapter 6 provides an introduction to the hardware platform and the Contiki Operating System. Chapter 7 describes the implementation details, including its main features and the overall architecture. Chapter 8 presents performance results and resource consumption statistics for the implemented SNMP agent. Finally, the thesis is concluded in Chapter 9.

Chapter 2

IEEE 802.15.4

The IEEE 802.15.4 standard developed by the 802.15.4 Task Group within the IEEE specifies the physical layer and media access control for low-rate wireless personal area networks providing node-to-node frame delivery between devices within reachable distance from each other. The standard is targeted at low cost, low speed and low power consumption devices. Several versions of the standard have been published, such as 802.15.4-2003 [40], 802.15.4-2006 [41], 802.15.4a-2007 [42], 802.15.4c-2009 [43] and 802.15.4d-2009 [44] . This chapter provides an overview of the IEEE 802.15.4 standard.

2.1 Device Types

The IEEE 802.15.4 standard distinguishes between two types of nodes, reduced-function devices (RFDs) and full-function devices (FFDs). FFDs typically have more resources, implement the complete protocol set and may be mains powered. An FFD can talk to RFDs or other FFDs, while an RFD can talk only to an FFD. FFDs aid RFDs by providing functions such as network coordination and packet forwarding. An RFD is intended for applications that are extremely simple, which do not have the need to send large amounts of data and may only associate with a single FFD at a time. Therefore, RFDs implement a minimal subset of the IEEE 802.15.4 protocol and can be implemented using minimal resources and memory capacity.

Two or more devices within a personal operating space communicating on the same physical channel constitute a wireless personal area network (WPAN). However, a network has to include at least one FFD.

2.2 Network Topologies

An 802.15.4 network may operate in either the star or the peer-to-peer topology. In the star topology devices communicate with a single central personal area network (PAN) coordinator. Only an FDD device can be a PAN coordinator. The PAN coordinator is usually mains powered, while the other devices are most likely battery operated. In the peer-to-peer topology a device can communicate with any other device as long as they are in range of each other. Based on the peer-to-peer topology more complex network formations may be constructed, such as mesh networking topology. However, the standard does not define a network layer, therefore, routing is not directly supported, but such an additional layer can add support for multihop communication. All devices operating on a network of either topology have unique 64 bit extended addresses. This address can be used for direct communication within the PAN. Each independent PAN selects a unique identifier. This PAN identifier allows communication between devices within a network using short 16 bit addresses and enables transmissions between devices across independent networks.

2.3 Frame Structure

The basic unit of data transport is a frame. The standard defines four frame structures: a beacon frame, used by a coordinator to transmit beacons; a data frame, used for all transfers of data; an acknowledgement frame, used for confirming successful frame reception; a MAC command frame, used for handling all MAC control transfers. Additionally, a superframe structure may be defined by the coordinator. In such case two beacons act as superframe limits and provide synchronization to other devices. A superframe consists of sixteen equal-length slots, which can be further divided into an active part and an inactive part, during which the coordinator may enter power saving mode. Any device wishing to communicate during the contention access period (CAP) between two beacons shall compete with other devices using a slotted CSMA-CA mechanism. For applications requiring specific data bandwidth, the PAN coordinator may dedicate portions of the active superframe, which form the contention-free period (CFP).

An important aspect of 802.15.4 is its limitation on the frame size, which is specified by the frame length 7 bit field (0-127 bytes). Taking into account the frame header, which is up to 25 octets (disabled security), this leaves 102 bytes for the payload of the higher layers.

2.4 Security

The standard defines several security services such as maintaining an access control list (ACL) and using symmetric-key cryptography to protect transmitted frames. Access control is a security service that provides the ability for a device to select the other devices with which it is willing to communicate. If the access control service is enabled, a device maintains a list of devices in its ACL from which it expects to receive frames. In order to protect data from being read by parties without permission, data may be encrypted using a key shared by a group of devices or using a key shared between two peers. However, key management is not specified in the standard and must be provided by the higher layers. Security services also specify frame integrity that uses a message integrity code to protect data from being modified by parties without the cryptographic key. This provides assurance that data came from a party with the cryptographic key.

Chapter 3

IPv6 over IEEE 802.15.4

The 6lowpan (Internet Protocol Version 6 over Low power Wireless Personal Area Networks) working group of the IETF is concerned with the specification of mechanisms to allow IPv6 packet transmission over IEEE 802.15.4 based networks. The working group has already completed two documents: "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals" [26] that documents and discusses the problem statement and "Transmission of IPv6 Packets over IEEE 802.15.4 Networks" [28] that defines the format for the adaptation between IPv6 and 802.15.4. The latter document describes the frame format for transmission of IPv6 packets and the method of forming IPv6 link-local addresses and statelessly autoconfigured addresses on IEEE 802.15.4 networks.

This chapter describes the Adaptation Layer and Header Compression Formats that have been introduced to allow the transmission of IPv6 packets over 802.15.4 links.

3.1 Addressing

IEEE 802.15.4 defines addresses of two types: IEEE 64-bit extended addresses or 16-bit short addresses unique within the PAN. Both types are supported by 6LoWPAN. 6LoWPAN supports stateless address autoconfiguration, which allows to obtain the interface identifier [20] for an IEEE 802.15.4 interface based on the EUI-64 identifier [22] [12] assigned to the IEEE 802.15.4 device. Even though all 802.15.4 devices have an EUI-64 address, 16-bit short addresses also can be used for address autoconfiguration. In this case, a pseudo 48-bit address is formed by concatenating 16 zero bits to the 16-bit PAN ID, the resulting 32 bits are concatenated with the 16-bit short address. The interface identifier is formed from this 48-bit address as defined in the "IPv6 over Ethernet" specification [12].

The IPv6 link-local address for an IEEE 802.15.4 interface is formed by appending the interface identifier to the prefix `FE80::/64`.

3.2 Adaptation Layer

The IPv6 minimum Maximum Transmission Unit (MTU) size is defined as 1280 octets. As described in Chapter 2, the maximum IEEE 802.15.4 frame size is limited to 127 octets. Taking into account the maximum frame overhead of 25 octets (disabled security), 102 octets are left at the media access control layer. Link-layer security imposes further overhead, which in the maximum case (21 octets for AES-CCM-128) leaves only 81 octets available, which is far below the minimum IPv6 packet size. Therefore, a fragmentation and reassembly adaptation layer is provided at the layer below IP.

Even though 802.15.4 networks are expected to commonly use mesh routing, the IEEE 802.15.4-2003 [40] specification does not define such capability. In order to support mesh routing, in addition to the hop-by-hop source and destination link-layer addresses, the originator and final destination addresses need to be known. For this purpose a Mesh Addressing header is introduced.

IPv6 datagrams transported over IEEE 802.15.4 are prefixed by an encapsulation header stack. Each header in the header stack contains a header type followed by zero or more header fields. There are four encapsulation headers defined: Mesh Addressing Header, Broadcast Header, Fragmentation Header and Dispatch Header. The first three of them are optional, whereas the last one is required.

3.2.1 Dispatch Header

The dispatch header is shown in Figure 3.1. The dispatch type is defined by a zero bit as the first bit followed by a one bit.

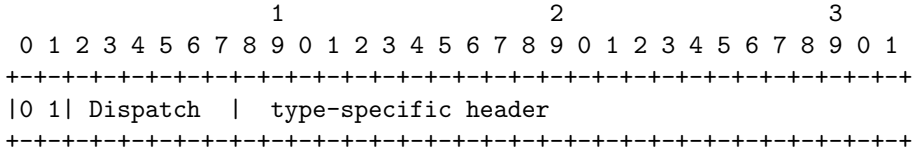

Figure 3.1: Dispatch Type and Header

The dispatch value identifies the type of payload immediately following the Dispatch Header. This can be, for example, a compressed or uncompressed IPv6 header. More details can be found in [28].

3.2.2 Mesh Addressing Header

The mesh header type is defined by a one bit as the first bit followed by a zero bit:

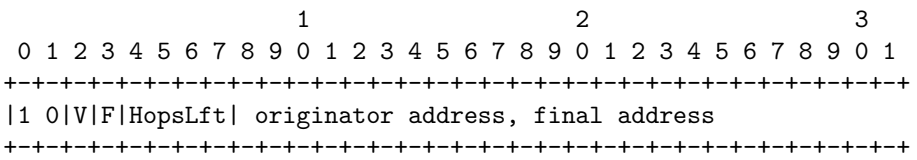

Figure 3.2: Mesh Addressing Type and Header

The `V` and `F` bits indicate whether an extended 64-bit (EUI-64) or short 16-bit address is used for the originator and the final destination respectively. The `HopsLft` field is a hop limit which is decremented by every forwarding node. The rest of the header specifies the originator and final destination link-layer addresses.

3.2.3 Fragmentation Header

When an entire IPv6 datagram may not fit within a single 802.15.4 frame, it is broken into several fragments. The first link fragment contains the fragment header shown in Figure 3.3.

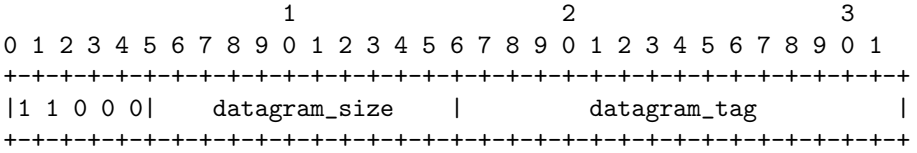

Figure 3.3: First Fragment

The second and subsequent link fragments contains a fragmentation header that conforms to the format shown in Figure 3.4.

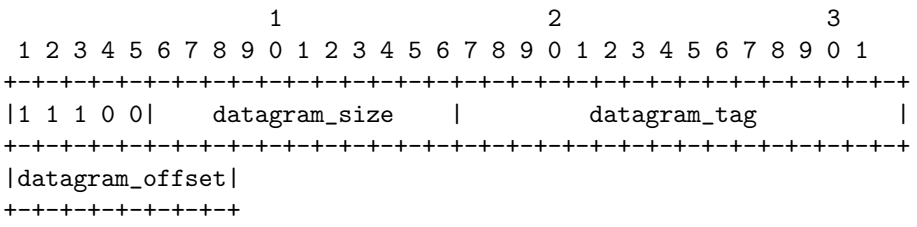

Figure 3.4: Subsequent Fragments

The first and subsequent fragment header types are identified by the binary sequences `11000` and `11100` respectively. The `datagram_size` 11-bit field encodes

the size of the entire IP packet before link-layer fragmentation. The `datagram_tag` field is the same for all link fragments of a datagram and incremented by the sender for successive fragmented datagrams. The `datagram_offset` field is present only in the second and subsequent link fragments and specifies the offset of the fragment from the beginning of the payload datagram.

3.2.4 Broadcast Header

Broadcast is an additional mesh routing functionality. For that a broadcast header is defined, the format of which is shown in Figure 3.5.

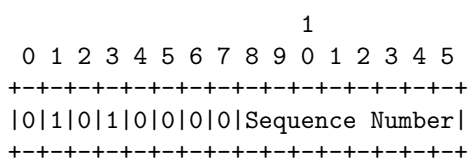

Figure 3.5: Broadcast Header

The broadcast header consists of the dispatch type `01010000` followed by a sequence number field used to detect duplicate packets. This field is incremented by the originator whenever it sends a new mesh broadcast or multicast packet.

3.3 Header Compression

As described in Section 3.2, 81 octets are left in an IEEE 802.15.4 frame for IPv6. The IPv6 header is 40 octets long which leaves only 41 octets for upper-layer protocols, like UDP. The latter uses 8 octets in the header, which leaves 33 octets for application data. The fragmentation and reassembly layer, described in Section 3.2, will also use at least one additional octet. This makes header compression almost unavoidable.

RFC 4944 [28] defines a stateless header compression mechanism for IPv6 datagrams (LOWPAN_HC1 and LOWPAN_HC2) to reduce the relatively large IPv6 and UDP headers. However, LOWPAN_HC1 and LOWPAN_HC2 are insufficient for most practical uses of 6LoWPAN networks, since they are effective only for link-local unicast communication, where IPv6 addresses can be derived directly from IEEE 802.15.4 addresses. When communicating with devices external to the LoWPAN or in a route-over configuration where IP forwarding occurs within the LoWPAN, LOWPAN_HC1 carries both IPv6 source and destination addresses in-line. The internet draft [21] defines an encoding format, LOWPAN_IPHC, for effective compression of unique local, global, and multicast IPv6 addresses based

on shared contexts. In addition, it defines an encoding format, LOWPAN_NHC, for arbitrary next header compression.

3.3.1 HC1 Header Encoding

Devices in the same 6LoWPAN network share some state. This makes it possible to compress headers without explicitly building any compression context state. The following IPv6 header values are expected to be common on 6LoWPAN networks: Version is IPv6; both IPv6 source and destination addresses are link local; the IPv6 interface identifiers for the source or destination addresses can be inferred from the layer two source and destination addresses; the packet length can be inferred either from layer two or from the `datagram_size` field in the fragment header; both the Traffic Class and the Flow Label are zero; and the Next Header is UDP, ICMP or TCP. The Hop Limit (8 bits) field is the only one that always needs to be carried. Fields, which do not match the described common case, have to be carried in-line. The HC1 encoding allows to compress a 40 byte IPv6 header to 2 bytes.

In addition to HC1, the HC2 encoding may be used, which allows to compress a UDP header to 4 octets instead of the original 8 octets. More details can be found in [28].

3.3.2 IPHC Header Encoding

The IPHC encoding format enables effective compression of the IPv6 header relying on information pertaining to the entire 6LoWPAN network. The encoding assumes the following common case parameters for 6LoWPAN communication: Version is 6; Traffic Class and Flow Label are both zero; Payload Length can be inferred from lower layers from either the 6LoWPAN Fragmentation header or the IEEE 802.15.4 header; Hop Limit will be set to a well-known value by the source; addresses assigned to 6LoWPAN interfaces will be formed using the link-local prefix or a single routable prefix assigned to the entire 6LoWPAN network; addresses assigned to 6LoWPAN interfaces are formed with an interface identifier derived directly from either the 64-bit extended or 16-bit short IEEE 802.15.4 addresses. The compression mechanism is adapted for these values of the header fields and in such scenario the LOWPAN IPHC can compress the IPv6 header down to two octets with link-local communication. When routing over multiple IP hops, IPHC can compress the IPv6 header down to 7 octets.

In contrast to HC2, the NHC encoding used with IPHC provides a flexible and extensible mechanism for arbitrary header compression, not only for UDP, TCP, and ICMPv6. With NHC, chains of next headers can be encoded efficiently, which is not possible with HC1 and HC2.

Chapter 4

Simple Network Management Protocol

Large networks have too many components to be managed by humans alone. Network devices have to maintain a large amount of management data such as configuration information, operational state and statistics. Management information can be used to understand how a network performs, how devices in a network are configured and to change their configuration. The Simple Network Management Protocol (SNMP) [9] is an application layer protocol that facilitates the exchange of management information between network devices. It exposes management data in the form of variables on the managed systems, which describe the system configuration. Since its first publication in 1988, the SNMP protocol has become a widely-used network management tool for IP-based networks.

Currently, there are three versions of SNMP defined. The first version (SNMPv1) is nowadays a historical IETF standard, although it is still widely supported by many vendors. One of the most important weaknesses of SNMPv1 is the lack of adequate mechanisms for securing the management function. Its security is based on a community string, which is a type of password transmitted in clear text. SNMPv2 extended the functionality of SNMPv1 and includes a number of improvements, such as additional protocol operations. A new security system was proposed, however, it was too complex and was not widely accepted. SNMPv3 addresses the security problems of the previous versions. The SNMPv3 architecture introduces a well defined extensible architecture and the User-based Security Model (USM) for message security.

The SNMP protocol is datagram-oriented and its implementation can be very lightweight [29]. It can fit resource-constrained devices very well. This makes it a perfect candidate as a management protocol for 6LoWPAN applications. In this chapter an overview of the SNMP protocol is given.

4.1 Management Information Base

SNMP itself does not specify which information is offered by a managed system. Management information, which can be accessed via SNMP, is viewed as a collection of managed objects in a virtual information store, called the Management Information Base (MIB). MIB modules describe the structure of the device's management information using a hierarchical namespace containing object identifiers (OID). Each OID identifies a variable that can be read or set via SNMP. Collections of related objects are defined in MIB modules written in the Structure of Management Information (SMI) data definition language, which is an adapted subset of the Abstract Syntax Notation One (ASN.1) language. A number of specific MIB modules has been defined as part of the SNMP suite specifying managed objects for the most common network management subjects.

4.2 Operational Model

In a simple case, two different types of hardware devices are defined: managed nodes and network management stations. Managed nodes are regular nodes on a network equipped with software to allow them to be managed using SNMP. A network management station is a designated network device that executes applications that monitor and control managed devices. Each device that participates in SNMP network management runs a piece of software called an SNMP entity. The SNMP entity is responsible for implementing all of the various functions of the SNMP protocol. The SNMP entity on a managed node consists of the SNMP agent, which implements the SNMP protocol, and the SNMP Management Information Base, which defines the types of information that the node provides via the SNMP protocol.

4.3 Protocol Architecture

The SNMP protocol has a modular architecture, which allows the evolution of the protocol and enables protocol extensions. The protocol architecture is composed of an SNMP engine and applications. An SNMP entity is an implementation of this architecture. The SNMP engine includes several subsystems communicating via defined abstract service interfaces. Abstract service interfaces describe the conceptual interfaces between various subsystems. While the subsystems can be changed and extended over time, the interfaces are fixed.

According to the SNMP architecture defined in [18], the SNMP engine is composed of a dispatcher, a message processing subsystem, a security subsystem, an access

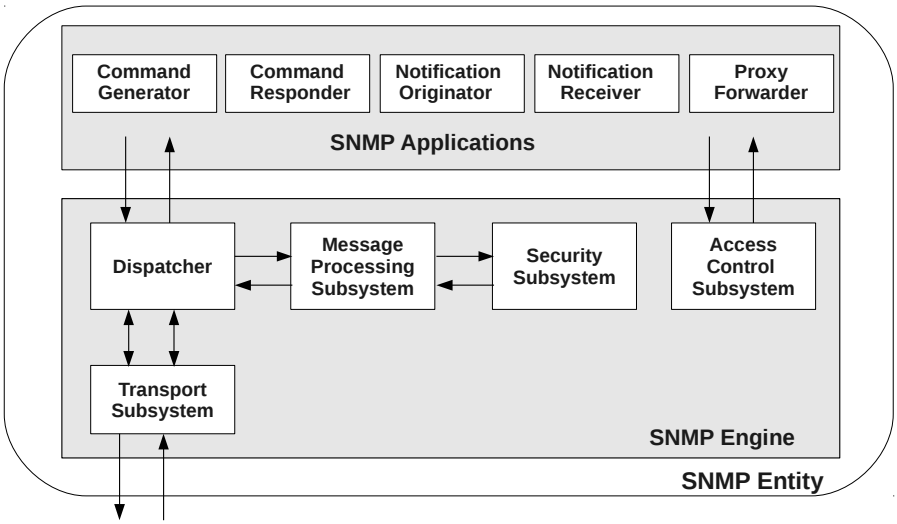

Figure 4.1: Structure of an SNMP entity

control subsystem, and a transport subsystem (Figure 4.1). Each subsystem may include multiple concrete models providing different implementations of the same service. The dispatcher is the key part of an SNMP engine. It is responsible for controlling the data flow within an SNMP entity. When an SNMP message needs to be prepared or when data needs to be extracted from an SNMP message, the dispatcher delegates these tasks to a version-specific message processing model within the message processing subsystem. It also dispatches SNMP protocol data units (PDUs) to SNMP applications. Each message processing model defines the format of a particular version of an SNMP message. Typically, the message processing subsystem supports three models for SNMPv1, SNMPv2c, and SNMPv3. The security subsystem provides security services such as authentication and privacy of messages. Authorization services are provided by the access control subsystem. The transport subsystem [19] allows for multiple transport protocols to be used.

There are several types of defined applications (Figure 4.1), such as a command generator which monitor and manipulate management data, a command responder providing access to management data, a notification originator initiating asynchronous messages, a notification receiver processing these messages, and a proxy forwarder which forwards messages between entities. Applications may use the services provided by the SNMP engine. An SNMP entity which includes one or more command generator and notification receiver applications is called an SNMP manager. An SNMP entity containing one or more command responder and notification originator applications is called an SNMP agent.

4.4 Message Format

The SNMPv3 message (Figure 4.2) contains the SNMP version, fields for global data (such as the message identifier, the maximum message size, the security model and the level of security), fields for the security model information and naming scope (context identifier and name), and finally the protocol data unit (PDU).

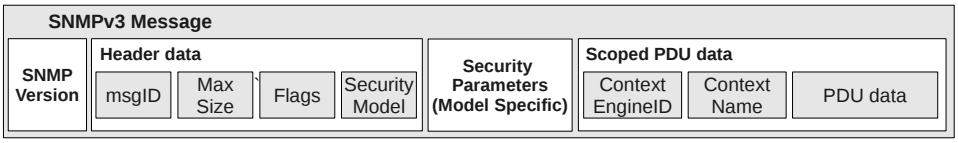

Figure 4.2: SNMPv3 Message Format

The SNMP Version field identifies the version of the message format. The message identifier (msgID) is used between two SNMP entities to coordinate request messages and responses and to coordinate the message processing by different subsystem models. The maximum message size (Max Size) is the maximum message size supported by the sender of the message, i.e., the maximum message size that the sender can accept when another SNMP engine sends an SNMP message. The Flag field contains several bit fields which control processing of the message. The security model field enables the concurrent support of multiple security models identifying which security model was used by the sender to generate the message. The security parameters field is used for communication between the security model modules in the sending and receiving SNMP engines. The contents and format of the data in this field are defined by the security model.

The naming scope contains a PDU and the context in which it has to be processed. The context engineID identifies the engine which realizes the managed objects referenced within the PDU, and the context name defines the particular context associated with the management information contained in the PDU of the message. These fields are followed by the PDU data.

The common format of protocol data units is shown in Figure 4.3. This format is used for all operations described in Section 4.5, except for the GetBulk request. The PDU is tagged with a number that identifies the type of the PDU. The request identifier is used to match requests with replies. It is generated by the device that sends a request and copied into this field in a response PDU by the responding SNMP entity. The error status tells the requesting SNMP entity the result of its request. A value of zero indicates that no error occurred; the other values indicate what sort of error happened. When error status is non-zero, the error index field points to the object generating the error, and has the value of zero in a request. The variable bindings is a set of name-value pairs identifying the MIB objects in

the PDU, and in the case of messages other than requests, containing their values.

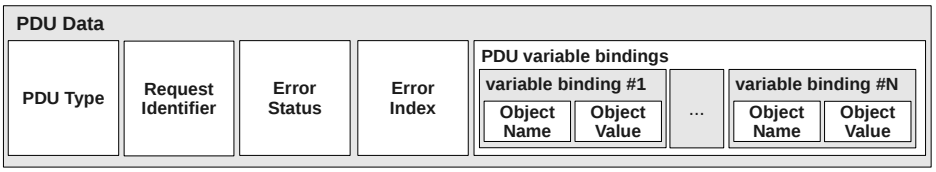

Figure 4.3: SNMP Common PDU Format

4.5 Protocol Operations

As described in Section 4.4, an SNMP message encapsulates a PDU which specifies the operations performed by the receiving SNMP engine. The SNMP protocol defines PDU formats for the following operations: `Get`, `GetNext`, `GetBulk`, `Response`, `Set`, `Trap`, `Inform` and `Report` (Figure 4.5).

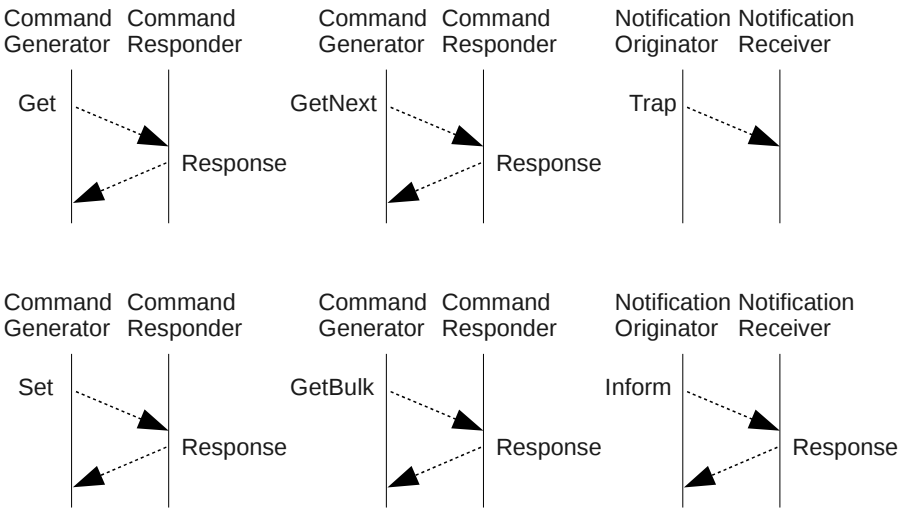

Figure 4.4: SNMP Operations

The `Get` request initiated by a manager allows to access and retrieve the value of one or more instances of management information. The agent processes each variable binding in the variable-binding list of the received `Get` PDU by resolving their values and produces a `Response` PDU. The `Response` PDU is sent back to the manager. If the processing of any variable binding fails, the details of the error are reflected in the the error status and error index fields in the `Response` PDU. The `GetNext` operation retrieves the value of the next existing MIB instances

in lexicographic order. Successive `GetNext` operations can be used to walk the MIB instances without prior knowledge about the MIB structure. The `GetBulk` operation is a generalization of the `GetNext` operation where the agent performs a series of `GetNext` operations internally. This makes it easier to acquire large amounts of related information without initiating repeated `GetNext` operations. The manager can set values for the managed objects by using the `Set` operation.

The `Trap` is sent by an agent to a manager and indicates that an event has occurred or a certain condition is present. There is no confirmation associated with this notification delivery mechanism. The confirmed notification delivery mechanism is provided by the `Inform` operation. Finally, the `Report` operation can be used internally to allow SNMP engines to communicate with each other for error notifications and clock synchronization.

4.6 Security Models

SNMPv1 and SNMPv2c provide only trivial security based on a clear-text community string and lack of adequate mechanisms for securing the management function. The security issues are addressed in SNMPv3 where the User-based Security Model (USM) [5] is defined that provides security services for authentication, timeliness and privacy. Another security model for the SNMP protocol is the Transport Security Model defined in [17] that makes use of existing commonly deployed security infrastructures providing lower-layer secure transports.

4.6.1 User-based Security Model

The USM security model is split into three modules: the authentication module provides data integrity and data origin authentication; the timeliness module provides delay and replay protection; the privacy module provides protection against disclosure of the message payload. The timeliness module is fixed, while multiple authentication and privacy modules are possible, which implement specific authentication and privacy protocols.

The USM provides three levels of security:

1) `noAuthNoPriv` mode provides no authentication and no encryption services.

2) `authNoPriv` mode provides message authentication, message integrity, and timeliness checking services but no encryption.

3) `authPriv` mode message provides authentication, message integrity, timeliness checking services and encryption of the payload.

4.7 Transport Mappings

Once an SNMP message has been generated, it is sent using the protocols at the levels below the application layer where SNMP resides. The document [35] defines the transport of SNMP messages over various protocols. Even though several mappings are defined, the mapping onto UDP over IPv4 is preferred for systems supporting IPv4.

When the UDP over IPv4 transport is in use, each SNMP message is serialized (i.e., encoded according to the Basic Encoding Rules (BER) [4]) onto a single UDP datagram. Since UDP does not guarantee data delivery, a request or reply can be lost in transit. Only the device that initially sends a request can know if there is a problem with the transport. This puts the responsibility for retransmission on the part that sends the request message. Two well-known UDP port numbers are reserved for SNMP. All devices that are set up to listen for SNMP requests, both agents and managers, listen on port 161. The second UDP port number is 162, which is reserved for SNMP notifications.

Chapter 5

Related Work

This chapter gives a short overview of the current approaches on network management in 6LoWPANs. Section 5.1 provides the details of the LoWPAN Network Management Protocol. The Simple Network Management Protocol for 6LoWPAN is discussed in Section 5.2.

5.1 LNMP

LoWPAN Network Management Protocol (LNMP) presented in [30] is a management architecture for 6LoWPAN based networks. The architecture emphasizes on reduction of communication cost in order to increase the network lifetime. One of the goals of LNMP is interoperability with SNMP. However, SNMP was considered to be quite bulky both in terms of complexity and communication for resource-constrained devices.

In the proposed architecture, SNMP is supported on the IPv6 network side only. The 6LoWPAN gateway acts as a proxy between SNMP and the local management framework. Whenever an SNMP request arrives at the gateway, it is translated from SNMP to a simplified query format. The gateway sends a UDP based query that contains identifiers of the objects to be retrieved to the destination device's agent. Similarly, when a reply arrives at the gateway, it is translated back to the SNMP format, and an SNMP response packet is sent back to the source of the requested object value. The gateway is also responsible for responding to requests for objects whose values are constant for the whole network. The proposed management architecture was implemented for the Atmel ATmega 128L microcontroller.

The advantage of LNMP is efficiency and reduced overhead on the network. However, it does not provide the native support of SNMP, and at the gateway similar

protocol operations have to be re-implemented. The gateway is a single point of failure in this architecture and has a large computational overhead caused by the conversion between protocols. Security implications are not considered in the paper [30].

5.2 6LoWPAN-SNMP

Simple Network Management Protocol for 6LoWPAN (6LoWPAN-SNMP) [10] is an extended modification of the Simple Network Management Protocol optimized for the resource-constrained nature of the low-power and low data-rate wireless networks. 6LoWPAN-SNMP utilizes several techniques to reduce the amount of SNMPv1 and SNMPv2c traffic on the 6LoWPAN network. First, the length of the SNMP message header fields is optimized. Second, the object identifier delta compression mechanism described in [37] is applied to compress the object identifier field in the PDU variable bindings. Extended protocol operations such as PeriodicGetRequest and broadcast/multicast SNMP messages are proposed, which effectively reduce the number of radio packets transmitted over the network. Instead of sending request messages every time, multiple periodic responses can be initiated by sending a singe PeriodicGetRequest message. A StopPeriodicGet message can be transmitted to stop the periodic responses. The current SNMP standard is designed only for point-to-point communication and network management systems (NMS) collect management information mainly by polling each host in the network. The proposed broadcast Get Request message allows to reduce the overhead of polling individual nodes in the network. A NMS injects a broadcast Get Request message into the network and every host immediately responds with a Response message to the NMS. However, broadcasting SNMP messages can not be applied in situations where each mote uses different OIDs for the same information (e.g., when table rows are indexed differently on different motes).

Compatibility with the standard SNMP is achieved by a proxy located on the gateway. The proposed mechanism is implemented on actual hardware platforms using the open-source Net-SNMP library and the Berkeley 6LoWPAN on the TinyOS 2.1. 6LOWPAN-SNMP does not support SNMPv3 and its authentication and encryption schemes.

Chapter 6

Targeted Platform

The hardware platform chosen for the project, the AVR Raven board, is described in Section 6.1. Section 6.2 gives the details of the network setup used in the project. In Section 6.3, the Contiki operating system is introduced providing the details of its networking capabilities.

6.1 AVR Raven Board

The AVR Raven board [1], produced by the Atmel Corporation, includes two microcontrollers (MCUs), one 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 ATmega1284PV and ATmega3290PV are modified Harvard architecture 8-bit RISC single chip MCUs with program and data stored in separate physical memory systems and different address spaces. Flash, EEPROM, and SRAM are all integrated onto a single chip. Program instructions are stored in non-volatile flash memory. Each instruction takes one or two 16-bit words. The MCUs of this family ensure minimal power consumption.

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 universal synchronous and synchronous serial receiver and transmitter (USART) is used as an inter processor communication bus which enables communication between two MCUs.

The AT86RF230 is a 2.4 GHz radio transceiver targeted for IEEE 802.15.4 and

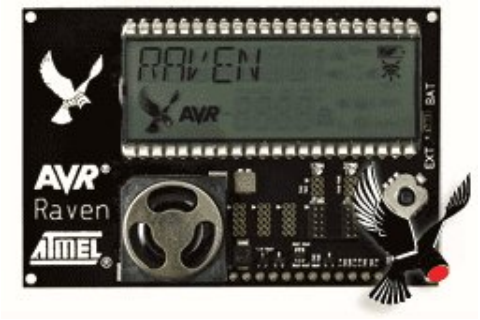

Figure 6.1: AVR Raven board

6LoWPAN applications. It supports automatic frame acknowledgement and retransmission, automatic CSMA-CA and data transfer speeds of up to 250 kbps. However, no security operation is available.

The AVR Raven board can be powered either from batteries or an external 5 to 12 Volts DC source. The board has been designed to run from two 1.5V LR44 battery cells.

A wide variety of third-party programming and debugging tools are available for the AVR. In this project the Atmel JTAGICE mkII programming platform has been used.

6.2 Network Setup

The network setup used in the project includes a PC acting as an IPv6 router with an 802.15.4 interface and an AVR Raven board acting as an IPv6 host. The AVR RZUSBstick, a USB stick with a 2.4 GHz transceiver, is used to provide the PC with an 802.15.4 interface. It enables the communication between the PC and AVR Raven board devices.

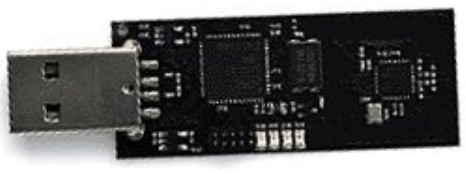

Figure 6.2: AVR RZUSBstick

6.3 Contiki Operating System

Contiki [15] is an open source, highly portable, multi-tasking operating system for memory-constrained networked embedded systems and wireless sensor networks. It consists of an event-driven kernel on top of which application programs are dynamically loaded and unloaded at runtime. The Contiki code footprint is on the order of kilobytes and memory usage can be configured to be as low as tens of bytes. It is written in the C programming language and is freely available under a BSD-style license.

Contiki processes use protothreads that provide a linear, thread-like programming style on top of the event-driven kernel. The main advantage of protothreads over ordinary threads in memory constrained systems is that they are very lightweight and do not require their own stacks. A protothread requires only two bytes of memory. All protothreads share the same stack and context switching is done by stack rewinding.

Contiki contains two communication stacks: Rime and uIP. Rime is a lightweight communication stack designed for low-power radios, which provides a wide range of communication primitives. uIP is a small RFC-compliant TCP/IP stack. While the uIP stack provides IPv4 connectivity, the IPv6 implementation is a part of uIPv6 [16] (shown in Figure 6.3). uIPv6 does not depend on any particular MAC or link layer and can run over 802.15.4/6LowPAN, 802.11 or Ethernet. SICSLoWPAN is the 6LoWPAN layer implementation for Contiki used to format packets between the 802.15.4 and the IPv6 layers. It is called by the MAC process when a 6lowpan packet is received, and by the `tcpip` process when an IPv6 packet needs to be sent. The implementation supports header compression mechanisms defined in RFC4944 [28] and the IPHC compression mechanism defined in [21].

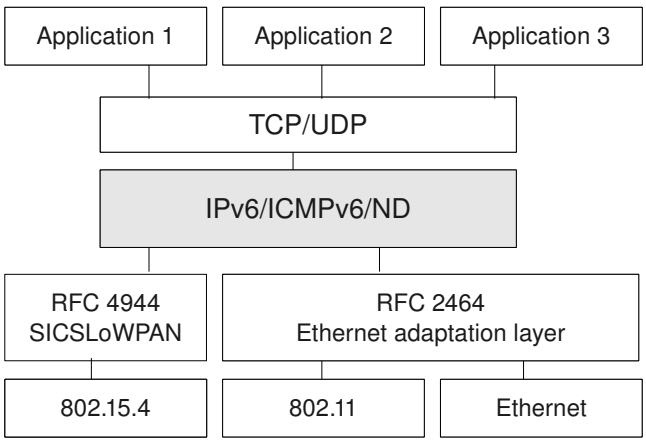

Figure 6.3: The uIPv6 stack

A complete IPv6/6lowpan/802.15.4 Contiki system requires around 40 kB of ROM

and 10 kB of RAM.

Chapter 7

Implementation

In the course of this project, the SNMP protocol has been implemented for the Contiki Operating System. The implementation includes SNMPv1 and SNMPv3 message processing models and supports `Get`, `GetNext` and `Set` operations. The USM security model has been implemented and supports the HMAC-MD5-96 authentication and CFB128-AES-128 symmetric encryption protocols. Management data are accessed via the MIB module, which provides an interface to define and configure accessible managed objects. 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 the implementation. The implementation has been accomplished in the C programming language and uses network primitives provided by the Contiki uIPv6 stack.

Section 7.1 describes the implementation design principles. In Section 7.2 an overview of modules and interfaces is presented. Section 7.3 provides the details of the processing of an incoming SNMP message. Section 7.4 describes the data structures used in the implementation. The implementation details of the User-based Security Model are discussed in Section 7.5. Section 7.6 describes the interfaces and data structures of the MIB module. Section 7.7 provides the configuration details of the agent.

7.1 Design Principles

The main challenge of the project is to run on the AVR Raven board hardware platform and fit its resource limitations. The Contiki operating system uses up to 10 kB of RAM and 40 kB of ROM of the targeted platform, therefore, the SNMP agent should not require more than 88 kB of ROM and 6 kB of RAM. The preference is given to easily readable and maintainable source code over memory efficient hard to understand programming constructs and hacks. Due to the small

amount of available RAM memory, necessary data structures have to be optimized and designed very accurately. Since the amount of ROM memory provided by the hardware platform is significantly larger than the amount of RAM, ROM memory should be preferred for storing read-only data.

The implementation does not follow the SNMP modular architecture exactly how it is defined in [18]. For such resource-constrained devices, this architecture is too bulky and, therefore, it would result in an inefficient implementation. In this project, the architecture and the abstract service interfaces it describes have been simplified to minimize the consumption of resources.

7.2 Modules and Interfaces

The implementation is divided into several logical subsystems interacting with each other via provided interfaces. Each subsystem handles a separate function, which allows to easily extend the implementation.

According to the SNMP architecture described in Section 4.3, the key subsystem is a dispatcher. The dispatcher interface is defined in the file `dispatcher.h` and declares the `dispatch()` function:

```
s8t dispatch(u8t* const input, const u16t input_len,
             u8t* output, u16t* output_len, const u16t max_output_len);
```

The first two arguments of this function are a pointer to an incoming SNMP message in the BER encoding format and the length of the message. The incoming message is processed and the resulting BER-encoded response message is stored to the `output` buffer. The `output_len` and `max_output_len` are the length of the response and the `output` buffer respectively. If the processing has finished successfully, the function returns a zero value and nonzero error code otherwise.

The message processing subsystem provides models for SNMPv1 and SNMPv3. The interfaces of these models are defined in files `msg-proc-v1.h` and `msg-proc-v3.h` respectively and each of them declares two functions:

```
s8t prepareDataElements_vX(u8t* const input, const u16t input_len,
                           u16t* pos, message_vX_t* request);

s8t prepareResponseMessage_vX(message_vX_t* message,
                              u8t* output, u16t* output_len,
                              const u8t* const input, u16t input_len,
                              const u16t max_output_len),
```

where `X` is the SNMP version. The `prepareDataElements_vX()` function extracts the abstract data elements from an incoming SNMP message and populates the fields of the version-specific `message_t` structure. The data structures are discussed in more details in Section 7.4. The `prepareResponseMessage_vX()` func-

tion prepares an outgoing SNMP response message. It takes a version-specific `message_t` structure as input and serializes it to a BER-encoded SNMP message.

The security subsystem provides an implementation of the User-based Security Model, discussed in Section 7.5. The model interface is defined in the file `usm.h` and declares the following functions:

```
s8t processIncomingMsg_USM(u8t* const input, const u16t input_len,
                           u16t* pos, message_v3_t* request);

s8t prepareOutgoingMsg_USM(message_v3_t* message, u8t* output,
                           u16t output_len, s16t* pos);

s8t authenticate(message_v3_t* message, u8t* output, u16t output_len);
```

The `processIncomingMsg_USM()` function extracts model specific security parameters from an incoming BER-encoded SNMP message and stores them to the fields of the `message_v3_t` structure. It is also responsible for authentication of the message and decryption of the scoped PDU. The `prepareOutgoingMsg_USM()` function adds model specific security parameters to an outgoing SNMP message and encrypts the scoped PDU data if necessary. The `authenticate()` function computes the message authentication code for a message which is ready to be sent.

The command responder application supports `Get`, `GetNext` and `Set` operations. The interface of the command responder is defined in the file `cmd-responder.h` and declares one function:

```
s8t handle(message_t* message);
```

This function processes an SNMP request stored in a generic `message_t` structure. In order to access management data, the MIB module is used. For this purpose the following functions are declared in the file `mib.h`:

```
mib_object_t* mib_get(varbind_t* req);

mib_object_t* mib_get_next(varbind_t* req);

s8t mib_set(mib_object_t* object, varbind_t* req);
```

The `mib_get()` function retrieves the value of the object named in the variable binding. If there is no such object, a zero value is returned and a pointer to the object otherwise. The `mib_get_next()` function retrieves the value of the lexicographical successor to the object named in the variable binding. If there is no such object, a zero value is returned. The `mib_set()` function assigns the value specified in the variable binding to the object.

The BER module defines a number of primitives for BER encoding and decoding. The interfaces of these functions are declared in the file `ber.h`. The module deals with integers, strings, object identifiers and sequences. It also defines encoding and decoding rules for the SNMP PDU.

7.3 Message Processing

The SNMP agent is run as a separate process defined in the file `snmpd.c`. At startup, the process accomplishes initialization of the Management Information Base and starts listening a specific UDP port for incoming SNMP request.

Upon receipt of a message from the network, it is passed to the dispatcher by calling the `dispatch()` function (shown in Figure 7.1). The dispatcher determines the version of the SNMP message and calls the version-specific `prepareDataElements_vX()` function of the message processing model. The message processing model decodes the BER-encoded message and populates the fields of the version-specific `message_t` structure. In case the USM security model is used, the message processing model calls the `processIncomingMsg_USM()` function, which performs the timeliness check, authenticates and decrypts the message if necessary.

Once the control has been returned to the dispatcher, the `handle()` function of the command responder is called. The command responder processes the request by calling the API functions of the MIB module, described in Section 7.6.

Once the request has been processed, the dispatcher calls the version-specific `prepareResponseMessage_vX()` function, which performs BER-encoding of the response message. In case of using the USM security model, the `prepareOutgoingMsg_USM()` and `authenticate()` functions are called to encode the security parameters and encrypt the scoped PDU.

The dispatcher returns the generated response message, which is sent back to the originator.

7.4 Data structures

The data structures used in the processing of an SNMP request are defined in the file `snmp.h`. An incoming BER-encoded SNMP message is decoded and stored into a version-specific `message_t` structure. While processing a message, the same data structures are used to generate a response.

The generic `message_t` structure contains the `version` and `pdu` members common for both SNMPv1 and SNMPv3:

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

The `message_v3_t` structure extends `message_t` with SNMPv3 and USM specific members such as `msgId`, `msgFlags`, `contextEngineID`,

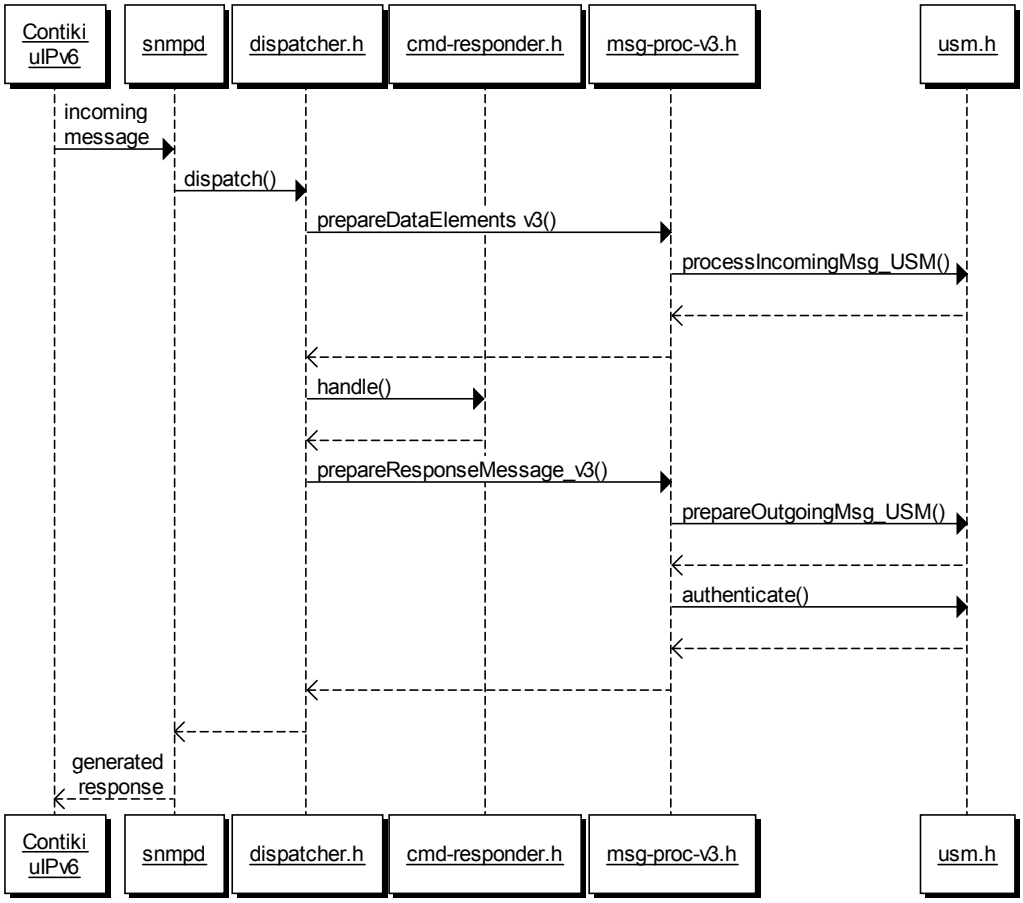

Figure 7.1: Processing of an incoming SNMPv3 message

`contextName`, `msgAuthoritativeEngineID`, `msgAuthoritativeEngineBoots`, `msgAuthoritativeEngineTime`, `msgUserName`, `msgAuthenticationParameters` and `msgPrivacyParameters`. The values of string type message fields are not copied from the message buffer. Instead, they are of the `ptr_t` type, which is a structure containing a pointer to the first byte of the value and its length. This allows to reuse the message buffer and save memory.

The members of the `pdu_t` structure correspond to the fields of the SNMP PDU:

```
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;
```

The `response_type` member specifies the response PDU type and can be either `Response-PDU` or `Report-PDU`. The `varbind_index` member specifies the byte in the request message buffer where the variable binding list starts. It is used to generate a response when an error occurs and the `error_status` field has a none-zero value. In this case, the variable-bindings field of the response PDU is re-formatted with the same values as in the received request and can be copied from the request message buffer. The `varbind_first_ptr` member of the `varbind_list_item_t` type points to the head of the linked list of variable bindings.

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

A variable binding is stored in an instance of the `varbind_t` structure:

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

In the implementation OIDs are not decoded and stored in the BER encoding format, which allows to save a significant amount of memory space. The `ptr_t` type is used to represent an OID in the `varbind_t` structure and specifies a sequence of bytes in the received message buffer where the OID is stored. The `varbind_value_t` type represents the value of the variable binding:

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

The `i_value` and `u_value` members are used for signed and unsigned integer-based values respectively, while `p_value` corresponds to string-based values and OIDs.

7.5 User-based Security Model

The User-based Security Model with the HMAC-MD5-96 authentication and the CFB128-AES-128 symmetric encryption protocols is supported by the agent. The implementation is based on the cryptographic primitives provided by the OpenSSL library [3]. The MD5 cryptographic hash function has been ported to the AVR Raven platform without significant changes. The source code is available in files `md5.h` and `md5.c`. The HMAC-MD5-96 calculation is implemented in the file `usm.c`. The implementation of the AES cipher algorithm provided by the OpenSSL library uses an array of constants of 4096 bytes. This would constitute 25% of

the overall RAM available on the targeted hardware platform. While adapting the AES implementation, the constants have been moved to the read-only flash program memory.

The HMAC-MD5-96 calculation and CFB128-AES-128 encryption uses localized authentication and encryption keys respectively. These keys are hardcoded in the file `keytools.c`. In order to construct a localized key based on the engine identifier and password, a `keygen` command-line tool has been developed.

7.6 Management Information Base

The MIB module provides access to management data. The interface of this module is defined in the file `mib.h` and allows to add and configure managed objects accessible via SNMP. Initially, the MIB has no objects in it. The definition of managed objects is accomplished in the implementation of the `mib_init()` function declared in the file `mib-init.h`. All objects have to be added to the MIB in ascending order of their object identifiers.

Two types of managed objects are supported: scalar and tabular. Scalar objects define a single object instance, while tabular objects define multiple related object instances that are grouped in MIB tables. The number of rows in a table is not fixed at compile time and can vary over time.

7.6.1 Interfaces

A managed object can be added to the MIB by calling either the `add_scalar()` or `add_table()` function, for scalar and tabular objects respectively.

```
add_scalar(ptr_t* oid, u8t attrs, u8t value_type, void* value,
           get_value_t gfp, set_value_t svfp);

add_table(ptr_t* oid_prefix,
          get_value_t gfp, get_next_oid_t gnofp, set_value_t svfp);
```

The first parameter of the `add_scalar()` function is the OID of the managed object. In case of the tabular object, this is the longest OID prefix common for all instances. The `attrs` parameter specifies the attributes of the managed object. The `FLAG_ACCESS_READONLY` value of this attribute can be used for defining read-only objects. The type of the object and its default value are specified by passing `value_type` and `value`.

User-defined functions for getting and setting the value of an object can be specified via the `gfp` and `svfp` arguments. These function types are defined in the following way:

```
typedef s8t(*get_value_t)(mib_object_t* object, u8t* oid, u8t len);

typedef s8t(*set_value_t)(mib_object_t* object, u8t* oid, u8t len,
                          varbind_value_t value);
```

The `object` argument is a pointer to the managed object. The `oid` and `len` arguments are passed only for tabular objects and specify the BER-encoded OID of the object instance and its length. The last argument of the setter function is the value to set.

The `add_table()` function requires one more argument, `gnofp`, of the `get_next_oid_t` function type:

```
typedef ptr_t* (*get_next_oid_t)(mib_object_t* object, u8t* oid, u8t len);
```

A function of this type has to resolve the `GetNext` request for the given OID and tabular object. If the OID is a lexicographical successor of the object then `0` value is returned.

7.6.2 Data Structures

Each managed object is represented in the MIB by an instance of the `mib_object_t` structure.

```
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
};
```

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 is an optional feature and 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 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

critical for resource constrained platforms. Such data can be moved out 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`.

The 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.

7.6.3 MIB Modules

The hierarchical structure of the `SNMPv2-MIB`, `IF-MIB` and `ENTITY-SENSOR-MIB` modules has been implemented for the AVR Raven board platform. The `SNMPv2-MIB` defines managed objects which describe the behavior of an SNMP entity. The `IF-MIB` provides access to information related to managing 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 for the AVR platform has been adapted. The managed objects of the `ENTITY-SENSOR-MIB` provides access to physical sensors. The readings of the temperature sensor can be obtained via the objects of this module.

7.6.4 BER Encoder

When adding a managed object to the MIB, the OID of the object has to be specified. As mentioned above, the implementation deals with OIDs encoded using the BER encoding format, which allows to save memory and avoid OID decoding. Therefore, OIDs of managed objects have to be defined encoded in BER.

For this reason, a BER encoder tool has been developed. The encoder takes an OID in the dot notation as an argument and outputs an array in the C programming language containing the OID encoded in BER. The generated array can be used in the definition of the managed object.

7.7 Configuration

Configuration of the agent can be accomplished by modifying the file `snmpd-conf.h`. The maximum supported length of a message is defined by the `MAX_BUF_SIZE` macro definition. In order to enable and disable message processing

models, the values of the `ENABLE_SNMPv1` and `ENABLE_SNMPv3` macro definitions can be switched between `1` and `0`. Disabling a message processing model allows to reduce memory usage of the agent. The authentication and privacy protocols can be disabled by changing the values of the `ENABLE_AUTH` and `ENABLE_PRIVACY` macro definitions respectively. The `TIME_WINDOW` macro definition allows to configure the length of the USM time window. The user name, engine identifier and community string can be also changed in this file.

Chapter 8

Evaluation

The implementation has been tested for interoperability with the `snmpget`, `snmpgetnext`, `snmpset` and `snmpwalk` applications from the Net-SNMP suite [2]. The network setup used for the experiments is described in Section 6.2. All measurements reported in this chapter were made with Contiki 2.4 using the IPHC header compression mechanism. Due to extreme resource limitations of the targeted platform, it is essential to measure the memory usage of the agent. The memory usage estimations are presented in Section 8.1. The response latency has been measured for the agent against different SNMP versions, security levels and operations. Section 8.2 presents the results of the measurements.

8.1 Memory and Code Footprint

Due to the limited memory resources of the targeted platform, 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 I present memory usage estimations obtained by using three different approaches.

8.1.1 Flash ROM and Static Memory Usage

In the first approach, which allows to determine the flash ROM and static memory usage, the avr-size utility was employed. 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, I measured the code and memory footprint of each module of the agent. Table 8.1 shows the detailed breakdown of the measurements.

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

Table 8.1: Flash ROM and static memory usage for the agent (bytes).

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 a huge code size. Using functions instead of them could reduce the code size, but would also effect the performance. It is worth to mention that the cryptographic primitives were ported from the OpenSSL library and are not optimized for embedded platforms.

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

8.1.2 Stack

An experimental approach has been taken to estimate the stack size used by the agent to process a request. Upon receipt of an incoming SNMP message, 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 8.2 presents the maximum stack size observed during experiments for different versions of SNMP and security modes used. Most of the stack is occupied by the response message buffer of 484 bytes. The SNMPv1 and SNMPv3 with `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.

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

Table 8.2: The experimental results for the maximum stack size. The measurements are given in bytes.

8.1.3 Heap

Memory for the data structures used to store the fields of an SNMP message and discussed in Section 7.4 is allocated from the heap using the `malloc` function.

Table 8.3 provides a memory estimation for the `message_t` structure.

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
Total	13

Table 8.3: Memory estimation for the `message_t` structure.

Each variable binding is stored in a linked list as an instance of the `varbind_list_item_t` structure. In addition to the memory calculation shown in Table 8.4, a variable binding uses 4 more bytes for an instance of the `ptr_t` structure which points to the OID stored in the message buffer.

The overall memory utilized to store an SNMPv1 message with `N` variable bindings can be found by the following formula: `13+N(9+4) = 13(N+1)`. The SNMP

Member	Size (bytes)
next_ptr	2
varbind.oid_ptr	2
varbind.value_type	1
varbind.value	4
Total	9

Table 8.4: Memory estimation for a variable binding.

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 unrealistic case an SNMPv1 message may carry `484/7 = 69` variable bindings, which would require `13(69+1)=910` bytes of the heap size to store such a message.

8.1.4 Managed Objects

As discussed in Section 7.6.2, each managed object is stored in memory as an instance of the `mib_object_t` structure. Table 8.5 presents a memory estimation for the members of this structure.

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
Total	16

Table 8.5: Memory estimation for an instance of the `mib_object_t` 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 can be found by the following 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.

8.2 Response Latency

A simple `upd-echo` application has been developed for Contiki, which allows to get a round-trip time estimation for a UDP datagram of a certain size. Table 8.6 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 the payload of 1 byte and 90 bytes, both of which fit into one 802.15.3 frame, is around 8 ms. The delay is caused by the low speed of the radio transceiver. While sending a 91 byte payload the 6LoWPAN fragmentation is used and, as expected, we observe an abrupt increase in the round-trip time comparing to the 90 byte payload.

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

Table 8.6: Round-trip time (RTT) measured with `upd-echo`.

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. 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 8.7 and Figure 8.1 present latency measurements for SNMPv1 and SNMPv3 in three different security modes. All experiments were done 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` mode. It 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.

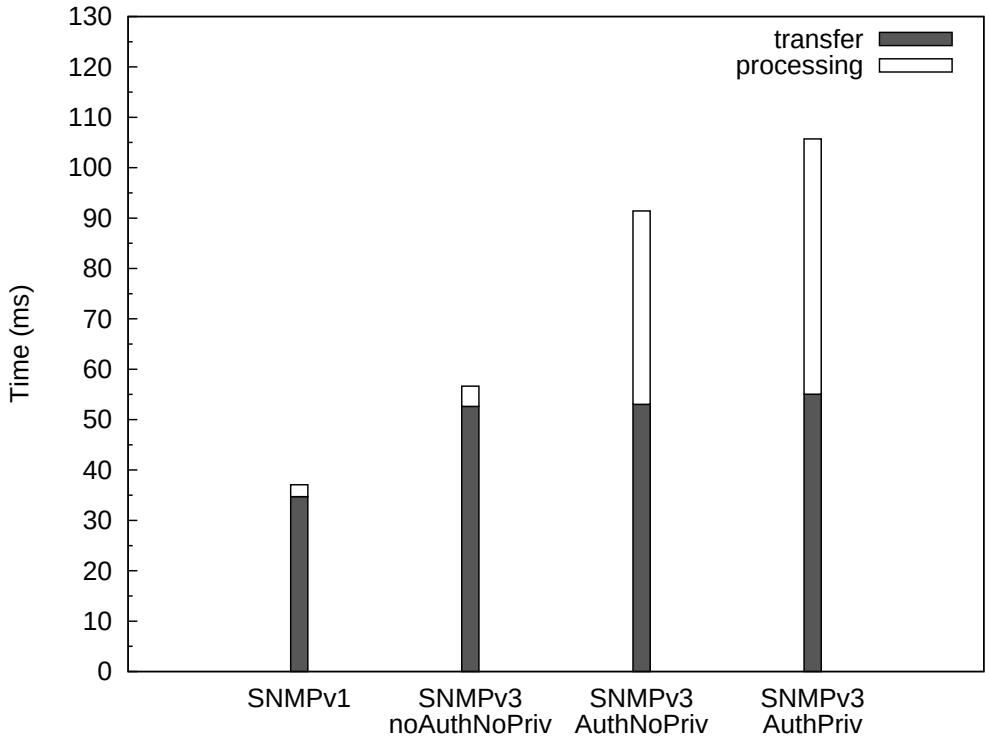

Figure 8.1: Time taken for transferring and processing an SNMP request.

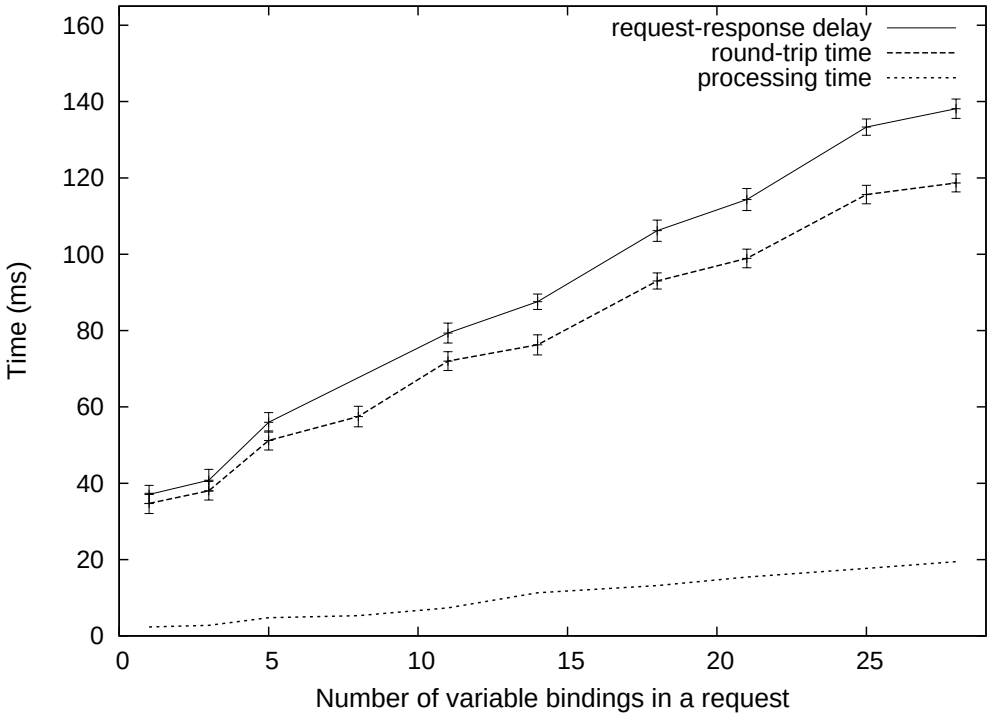

Figure 8.2: Time spent in transferring and processing SNMPv1 requests and responses as a function of the number of variable bindings in a request.

Version	Operation	Security mode	Latency	Variance	RTT	Δ
v1	Get	–	37.05	5.69	34.70	2.35
v1	Next	–	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	Next	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	Next	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	Next	authPriv	106.46	2.59	55.03	51.43
v3	Set	authPriv	106.73	3.59	55.03	51.70

Table 8.7: 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 `upd-echo`. The last column is the processing time taken by the agent. The measurements are given in milliseconds.

Figure 8.2 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 comparing 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.

Chapter 9

Conclusions

An implementation of the Simple Network Management Protocol for resource constrained devices under the Contiki Embedded Operating System has been developed. 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 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 for the agent.

The evaluation of the implementation has been carried out on the AVR Raven Board hardware platform. The experimental results reveal that the request processing time for SNMPv1 and SNMPv3 in the `noAuthNoPriv` mode is relatively small comparing 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 add more features to the implementation. For example, the `GetBulk` operation and the message processing model for SNMPv2c could be implemented. It is also possible to put more effort on defining the Management Information Base for 6LoWPANs.

Appendix A

Appendix

A.1 Source Code

The source code and other files necessary to build the project are publicly available for download and experimentation. Access to the source code repository is available using the Apache Subversion control system at `http://contiki-snmp.googlecode.com/svn/trunk/`. The project has the following directory structure:

- `app` – the application which starts the agent's process;
- `src` – the source code of the agent;
- `ber-encoder` – the source code of the BER encoder utility;
- `keygen` – the source code of the keygen utility;
- `test` – scripts used for testing of the agent;
- `contiki` – Contiki files changed for the agent;

A.2 Running SNMP Agent

Use the following steps to install the agent on the AVR Raven board under a Linux computer:

- Download Contiki 2.4 from `http://sourceforge.net/projects/contiki/files/Contiki/Contiki%202.4/contiki-2.4.zip/download`. Unpack the archive to the `$SETUP_ROOT_DIR` directory:

```
cd $SETUP_ROOT_DIR
unzip contiki-2.4.zip
```

- Check out the latest source code of the agent from the Subversion repository:

```
cd $SETUP_ROOT_DIR
svn checkout http://contiki-snmp.googlecode.com/svn/trunk/ \
             jacobs-snmp-read-only
```

- Modify the file `$SETUP_ROOT_DIR/jacobs-snmp-read-only/app/Makefile` setting the value of the `CONTIKI` variable so that it points to the directory `$SETUP_ROOT_DIR/contiki-2.4`:

```
6:  CONTIKI=$SETUP_ROOT_DIR/contiki-2.4
```

- Create a symbolic link with the name `snmpd` in the directory `$SETUP_ROOT_DIR/contiki-2.4/apps` pointing to `$SETUP_ROOT_DIR/jacobs-snmp-read-only/src`:

```
cd $SETUP_ROOT_DIR/contiki-2.4/apps
ln -s ../../jacobs-snmp-read-only/src/ snmpd
```

- Update the Contiki files to which changes were applied by running the command:

```
cp $SETUP_ROOT_DIR/jacobs-snmp-read-only/contiki/* \
   $SETUP_ROOT_DIR/contiki-2.4/cpu/avr/radio/rf230
```

- Compile the agent:

```
cd $SETUP_ROOT_DIR/jacobs-snmp-read-only/app
make raven-mib
```

The command `make upload` can be used to program the binary on the AVR Raven board.

Bibliography

[1] *AVR Raven*. `http://www.atmel.com/dyn/products/tools_card_v2.asp?tool_id=4395`. Accessed 5th June, 2010.

[2] *Net-SNMP*. `http://net-snmp.sourceforge.net/`. Accessed 9th June, 2010.

[3] *OpenSSL Project*. `http://www.openssl.org/`. Accessed 21th June, 2010.

[4] Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). *International Organization for Standardization. International Standard 8825*, December 1987.

[5] U. Blumenthal and B. Wijnen. User-based Security Model (USM) for version 3 of the Simple Network Management Protocol (SNMPv3). *IETF, Internet RFC 3414*, December 2002.

[6] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). *IETF, Internet RFC 1157*, May 1990.

[7] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework. *IETF, Internet RFC 1908*, January 1996.

[8] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Introduction to Community-based SNMPv2. *IETF, Internet RFC 1901*, January 1996.

[9] J. Case, R. Mundy, D. Partain, and B. Stewart. Introduction and Applicability Statements for Internet Standard Management Framework. *IETF, Internet RFC 3410*, December 2002.

[10] H. Choi and H. Cha N. Kim. 6LoWPAN-SNMP: Simple Network Management Protocol for 6LoWPAN. *2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 305–313, 2009.

[11] A. Conta, S. Deering, and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. *IETF, Internet RFC 4443*, March 2006.

[12] M. Crawford. Transmission of IPv6 Packets over Ethernet Networks. *IETF, Internet RFC 2464*, December 1998.

[13] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. *IETF, Internet RFC 2460*, December 1998.

[14] R. Draves and D. Thaler. Default Router Preferences and More-Specific Routes. *IETF, Internet RFC 4191*, November 2005.

[15] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[16] Mathilde Durvy, Julien Abeillé, Patrick Wetterwald, Colin O'Flynn, Blake Leverett, Eric Gnoske, Michael Vidales, Geoff Mulligan, Nicolas Tsiftes, Niclas Finne, and Adam Dunkels. Making sensor networks ipv6 ready. In *Proceedings of the Sixth ACM Conference on Networked Embedded Sensor Systems (ACM SenSys 2008), poster session*, Raleigh, North Carolina, USA, November 2008. Best poster award.

[17] D. Harrington and W. Hardaker. Transport security model for the simple network management protocol (snmp). *IETF, Internet RFC 5591*, June 2009.

[18] D. Harrington, R. Presuhn, and B. Wijnen. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. *IETF, Internet RFC 3411*, December 2002.

[19] D. Harrington and J. Schoenwaelder. Transport subsystem for the simple network management protocol (snmp). *IETF, Internet RFC 5590*, June 2009.

[20] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. *IETF, Internet RFC 4291*, February 2006.

[21] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams in 6LoWPAN Networks. *IETF, Internet-Draft draft-ietf-6lowpan-hc-07*, April 2010.

[22] IEEE. Guidelines for 64-bit global identifier (eui-64) registration authority.

[23] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. *IETF, Internet RFC 3775*, June 2004.

[24] E. Kim, D. Kaspar, N. Chevrollier, and JP. Vasseur. Design and Application Spaces for 6LoWPANs. *IETF, Internet-Draft draft-ietf-6lowpan-usecases-05*, November 2009.

[25] E. Kim, D. Kaspar, C. Gomez, and C. Bormann. Problem Statement and Requirements for 6LoWPAN Routing. *IETF, Internet-Draft draft-ietf-6lowpan-routing-requirements-06*, March 2010.

[26] N. Kushalnagar, G. Montenegro, and C. Schumacher. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. *IETF, Internet RFC 4919*, August 2007.

[27] P. Levis, A. Tavakoli, and S. Dawson-Haggerty. Overview of Existing Routing Protocols for Low Power and Lossy Networks. *IETF, Internet-Draft draft-ietf-roll-protocols-survey-07*, April 2009.

[28] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. *IETF, Internet RFC 4944*, September 2007.

[29] H. Mukhtar, S. Joo, J. Schoenwaelder, and K. Kim. SNMP optimizations for 6LoWPAN. *IETF, draft-hamid-6lowpan-snmp-optimizations-02*, October 2009.

[30] H. Mukhtar, Kim Kang-Myo, S.A. Chaudhry, A.H. Akbar, Kim Ki-Hyung, Seung-Wha Yoo, and Suwon Ajou Univ. LNMP - Management architecture for IPv6 based low-power wireless Personal Area Networks (6LoWPAN). *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 417–424, April 2008.

[31] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). *IETF, Internet RFC 4861*, September 2007.

[32] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. *IETF, Internet RFC 2474*, December 1998.

[33] J. Postel. User Datagram Protocol. *IETF, Internet RFC 768*, August 1980.

[34] R. Presuhn, J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Management Information Base (MIB) for the Simple Network Management Protocol (SNMP). *IETF, Internet RFC 3418*, December 2002.

[35] R. Presuhn, J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Transport Mappings for the Simple Network Management Protocol (SNMP). *IETF, Internet RFC 3417*, December 2002.

[36] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *IETF, Internet RFC 3168*, September 2001.

[37] J. Schoenwaelder. SNMP Payload Compression. *IETF, draft-irtf-nmrg-snmp-compression-01*, October 2001.

[38] J. Schönwälder, A. Pras, M. Harvan, J. Schippers, and R. Meent. SNMP Traffic Analysis: Approaches, Tools, and First Results. *IEEE Computer Society Press*, pages 324–332, May 2007.

[39] Z. Shelby, P. Thubert, J. Hui, S. Chakrabarti, C. Bormann, and E. Nordmark. 6LoWPAN Neighbor Discovery. *IETF, Internet-Draft draft-ietf-6lowpan-nd-11*, July 2012.

[40] IEEE Computer Society. IEEE Std. 802.15.4-2003. October 2003.

[41] IEEE Computer Society. IEEE Std. 802.15.4-2006. September 2006.

[42] IEEE Computer Society. IEEE Std. 802.15.4a-2007. August 2007.

[43] IEEE Computer Society. IEEE Std. 802.15.4c-2009. April 2009.

[44] IEEE Computer Society. IEEE Std. 802.15.4d-2009. April 2009.

[45] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. *IETF, Internet RFC 4862*, September 2007.

[46] P. Thubert and T. Winter. RPL: IPv6 Routing Protocol for Low power and Lossy Networks. *IETF, Internet-Draft draft-ietf-roll-rpl-04*, October 2009.

[47] JP. Vasseur, M. Kim, K. Pister, and H. Chong. Routing Metrics used for Path Calculation in Low Power and Lossy Networks. *IETF, Internet-Draft draft-ietf-roll-routing-metrics-04*, December 2009.