Developing an SNMP agent protocol entity with object oriented Perl

Arnoud Zwemmer, Eric van Hengstum and Marten van Sinderen
Centre for Telematics and Information Technology
University of Twente
P.O. Box 217, Enschede, the Netherlands
{zwemmer, hengstum, sinderen}@cs.utwente.nl
http://www.cs.utwente.nl/~{zwemmer, hengstum, sinderen}

Abstract

The Damocles project at the University of Twente is developing a prototyping vehicle for new SNMP versions. The motivation for such a vehicle stems from the fact that the IETF will develop new versions of SNMP in response to evolving user needs (version 2 is currently underway). Whereas the transition to newer versions is attractive from a functional and/or performance point of view, the effort and cost involved in such a transition should not be underestimated. The Damocles project investigates how modular design and implementation, and the application of object oriented techniques, facilitate modification and reduce the effort and cost of version transition. In this paper, we discuss a part of the Damocles prototyping vehicle: an SNMP agent protocol entity written in object oriented Perl. We discuss the design and implementation of the protocol entity, and how this development effort contributes to the objectives of the Damocles project.

Keywords: SNMP, SNMPv2, agent technology, protocol engineering, modularity, object oriented technology, Perl

1 Introduction

The Simple Network Management Protocol (SNMP; [Case90]) is currently the protocol of choice to provide network management on TCP/IP networks, and it is rapidly spreading into the field of PC networks. SNMP's popularity is to a large extent based on its simplicity; it supports a limited set of functions that cover the basics of network management while little overhead is imposed on the existing network. A second version of SNMP, SNMPv2, that overcomes some

of the shortcomings of the original version, has recently been promoted to the status of draft standard.

The Damocles project at the University of Twente addresses one of the main problems related to network management: the cost and effort involved in changing from one network management solution to another, in general more sophisticated, one. According to [Stallings93], the average firm in the US spends about 15% of its information systems budget on network management; this expenditure signifies the importance of network management, and thus of (the support for) network management *evolution*.

In the Damocles project, a platform for SNMP agent implementation is being developed that consists of building blocks for the construction of SNMP agents and functionality that assists the users of the framework to create and configure SNMP agents. New versions of SNMP agents may thus be prototyped within short time scales, by modification of a minimal set of building blocks. The building block approach was applied to both the protocol aspects and the Management Information Base (MIB) aspects of agents. The Damocles project is also used as a testbed for acquiring experience with protocol and MIB design approaches and with popular programming languages. In particular, building blocks are derived with functional as well as object oriented design approaches and implemented in C but also in, for example, Tcl, Perl and C^{++}.

This paper reports on our experience with the development of an SNMP agent protocol entity using object oriented (i.e., version 5) Perl. The paper discusses the high level design of the SNMP agent which was shared by different implementation teams (concerned with different parts of the agent and/or using different languages), the software architecture of the agent protocol entity, and the final implementation. It also discusses the results of this development effort in relation to other results and to the objectives of the Damocles project.

Object orientation has become a very popular paradigm for systems engineering. In fact, it is promoted as *the* solution to tackle the complexity and fast evolution of contemporary and future software based systems. At the same time, it has been observed that the use of object orientation for the design and implementation of *distributed* systems, such as network management systems, presents a number of problems ([Horn90]). Among these problems are the handling and decomposition of distributed objects and the identification of objects in a distributed environment. Our development effort did not tackle these principal problems. This was not necessary since the starting point of the effort were the SNMP specifications, of which we only considered the agent part. Where applicable, we followed the Booch method ([Booch94)]; in particular, we used CRC (Class/Responsibilities/Collaborators) cards for the object oriented analysis phase. We did not derive the classes and objects in our design from a

Finite State Machine (FSM) description of SNMP. Although this can be done in a systematic way (see [Ananthaswamy95], for example), there was little merit of this approach in our case. First, because the FSM description of the SNMP agent protocol entity is very simple (the fact that the FSM description can be thoroughly validated was thus not an argument for adopting this approach) and, second, because the pursued modularisation of the protocol entity was expected to be based on a separation of (protocol control) information processing functions and not on states.

The remaining of this paper is organized as follows: section 2 discusses the overall design of the Damocles SNMP agent; section 3 presents the object oriented software design of the protocol entity; section 4 mentions particularities of the Perl implementation; section 5 contains a discussion and evaluation of the implementation results; section 6 concludes the paper with some final remarks and an outlook of future work.

2 Overall design

To support the primary goal of the Damocles project, the Damocles agent must be intrinsically very modular. Based on the observation that the *storing* of management information and the *transport* of management information, as defined in the SNMP framework, constitute different design concerns, the Damocles agent is divided into two major modules corresponding to these concerns. The MIB module primarily supports the MIB prototyping purpose and the SNMP protocol entity (SPE) module is suited for rapid implementation of new SNMP (protocol) versions. Of course, the entire agent must be able to operate as a normal SNMP agent.

A further structuring is based on the handling of input and output events by the agent. Three different input sources / output sinks for events are identified. The first source/sink is the network to which the agent is connected. Via the network, i.e. the transport service provided by the user-datagram protocol, manager requests are delivered to the agent. Also, responses are returned by the agent via the network to the originating manager application, and (trap) notifications generated by the agent are sent via the network to designated management stations. The second source/sink is the human user of the agent. For instance, a user can be a MIB developer, who needs to interact with the agent for carrying out MIB prototyping. The third source/sink of events is the system which is actually managed by the agent. In a managed system some forms of calamity may happen. The agent must be made aware of such panic situations through appropriate events and immediately act upon such events. Figure 1 depicts the overall structuring of the Damocles agent.

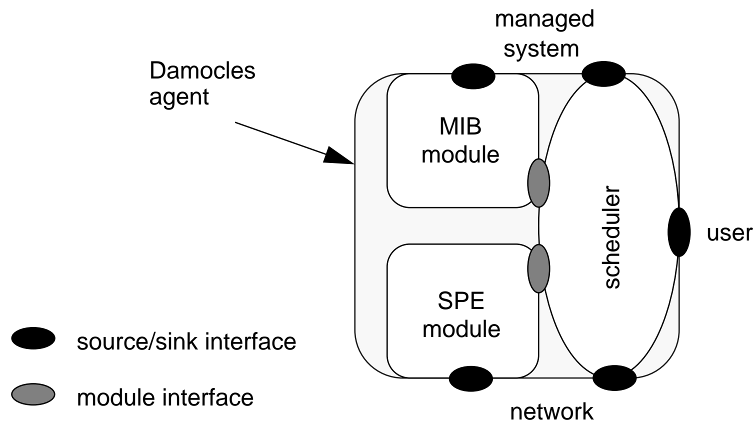

Figure 1: High-level structure of the Damocles agent

To monitor the input sources, a scheduler was introduced which polls all three sources/ sinks in a non-blocking, round-robin, equal-priority fashion. The scheduler has to communicate with the MIB module and the SPE module, so that the latter two can perform the actual agent functionality. Two module interfaces are introduced for this purpose, achieving maximum independence of the modules. The MIB module and the SNMP protocol module cannot directly access each other's functionality; all communication is via the scheduler.

Table 1 summarizes the interface functions of the modules inside the Damocles agent.

Module	Interface function	Description
MIB	MIB_read	processes a GET request
	MIB_write	processes a SET request
	MIB_trap	processes a pending TRAP situation
SPE	SNMP_receive	receives a GET/SET PDU from a manager
	SNMP_send	send a RESPONSE PDU to a manager
	SNMP_trap	sends a TRAP PDU to a manager.

Table 1: Interface functions of the Damocles agent modules

Two other important decisions shaped the development of the Damocles agent. Both decisions were based on the guiding principle that, during a first development cycle, attention should be focused on (the functional aspects of) modularization while keeping the Damocles agent in all other respects as simple as possible. The first decision was that the communication between modules will be supported by ordinary procedure calls, rather than by interprocess communication, since simplicity would be easier achieved by adopting the former. The second decision was that the Damocles agent will only support one manager request at a time. Performance, which would be favoured by concurrent processing of requests, was thus

considered as an issue of lesser importance than simplicity. The current Damocles agent relies on the fact that the network can temporarily store pending requests, and pending requests that time-out can be retransmitted by the originating manager application.

3 Object oriented software design

3.1 The development process

Several object oriented (OO) development methodologies exist, which prescribe a sequence of actions that can be taken in OO analysis and design. A discussion and comparison of some of these methods can be found in [ObjectAgency95]. During the development process, we used the Booch method described in [Booch94] as a guideline, motivated by the fact that it is well-known, widely used and was used successfully in previous projects. The method was however not followed very rigidly. Rather, we aimed at an intuitive clear design structure, using formally defined methodologies only as a reference.

The first step in the development process forms the requirements analysis. As the operation of the SNMP protocol is standardized, the problem domain is clear. The procedure rules specified in the standard must be adhered to. Due to this specification, the strategic design decisions of the system are already taken and the important abstractions and the necessary behaviour of the system can be derived relatively easy. Therefore we could almost directly proceed with the second step in the development process, i.e. the identification of classes, their responsibilities and the relationships between them.

3.2 Identification of classes and their semantics

Protocol behaviour is often modelled as a Finite State Machine (FSM). The protocol is always in a certain state and will change state whenever an event occurs. [Ananthaswamy95] describes an OO design approach for a data link protocol represented as FSM. The state information is encapsulated in an object; one such object is the current state object, which provides methods for each possible event that can occur in that particular state. When an event occurs, a method of the current state object will be invoked and after that, another object will become the current state object.

The question is whether SNMP is suitable for this approach and whether the SPE module can be modelled as FSM. A short investigation showed that the latter is problematic. Only three events related to the SPE can be identified: the interface functions *SNMP_receive*(), *SNMP_send()* and *SNMP_trap()* (see Table 1). When these occur, a sequential procedure is

started in the SPE, which is not influenced by any additional events. Thus, the SPE only comprises three states, **receive**, **send** and **trap**, which leads to three corresponding objects in the software design. A finer modularity is possible if we introduce substates for each action within a state, assuming that each state specifies an (atomic, in the original FSM) sequence of actions. This, however, introduces the disadvantage of changing the entire FSM when the sequence of actions changes. In addition, grouping of the corresponding objects in a hierarchical fashion is difficult, because each (sub)state is an equal entity. In a non-FSM approach, it is easier to add hierarchy to the design and improve modularity.

For a more intuitive approach, we used CRC cards (see [Booch94], for example) that has been proven as a useful development tool in many application areas. This led to the identification of the classes and responsibilities listed in Table 2.

The **SPE** class was identified as the main class responsible for receiving and sending all PDUs an SNMP agent must support. The INFORM-Request PDU is only needed in management stations and was not taken into account. The remaining PDUs can be characterized as either being part of the request/response interaction between a management station and the agent (GET, GETNEXT, GETBULK, SET, RESPONSE, REPORT), or as an agent-originated notification that is sent to one or more management stations (TRAP). This separation was already made in the overall design, where the *SNMP_trap()* interface function was defined in addition to the *SNMP_receive()* and *SNMP_send()* interface functions to support agent-originated notifications.

Class	Responsibilities	Methods
SPE	Responsible for entire state and behaviour of the SPE. Methods provide the functionality required by overall design. The SPE class knows how to send and receive all SNMP PDUs including traps.	receive() send() trap()
Request	Handling the request/response operations for the agent, i.e. receive PDUs and send replies back. Responsible for GET, GETNEXT, GETBULK, SET, RESPONSE and REPORT PDUs.	receive() send()
Notification	Handling of agent-originated notifications (Trap-PDUs).	trap()
Message	Encapsulates all information contained in the SNMP Message. The responsibility for this information is partly delegated to the PDU class and the AdminInfo class.	parse(), build() get_info() set_info() authenticate()
PDU	Encapsulates information contained in the PDU part of the SNMP Message.	parse(), build() get_info() set_info()
AdminInfo	Encapsulates information contained in the administrative portion of the SNMP Message.	authorize() get_info() set_info()

Table 2: Classes and responsibilities

Class	Responsibilities	Methods
Context	Reponsible for managing the context information needed in the **Request** class.	context_get() context_set()
TF	Abstract class, providing virtual methods for all transport functionality. Specific building blocks (UDP, TCP) can inherit from this class and provide an implementation of these methods.	poll() recv() send()
CF	Abstract class, providing virtual methods for all ASN.1 encoding/decoding. As with TF, specific implementations (BER, PER) can inherit from this class.	encode() decode()
PRIV	Abstract class, providing virtual methods for the encryption needed. Specific implementations are for example DES, IDEA, RC4.	encrypt() decrypt()
AUTH	Abstract class, providing virtual methods for authentication. Specific implementations are for example MD5, SHA.	auth_parse() auth_build()
ADM	Abstract class, providing virtual methods for the administrative model that is used, for example, SNMPv1, SNMPv2C, USEC, SNMPv2*.	check()

Table 2: Classes and responsibilities

The responsibilities of the **SPE** class were therefore separated into two classes: **Request**, responsible for all request/response operations, and **Notification**, responsible for notifications (traps) originating from the agent. The **Request** class must maintain some context information in order to relate a received request that is processed and sent to the MIB module to a response coming back from the MIB module. This response is sent only to a single management station. This is not the case in the **Notification** class, where context information is not needed and PDUs may be sent to a variety of management stations. This justifies the decision of separating these responsibilities.[1]

We defined a class **Message**, responsible for the SNMP Message and its semantics as a whole, and two additional classes, **PDU** and **AdminInfo**, each responsible for the semantics of the two components of the Message[2]. We encapsulated these distinct parts of information in two different classes, because of the variety of administrative models that are currently defined, each defining their own administrative information, but leaving the format of the SNMP PDU intact. Moreover, the authentication and authorization functionality of SNMP is only related to this administrative information, not to the management information contained in the SNMP PDU.

1. Although the current usage and semantics of the REPORT-PDU are defined as being an aspect of error-reporting between SNMPv2 entities and can be considered a notification in this respect, this is still related to an earlier received PDU that caused the error and is sent to the management station that the PDU came from. One can think of the Report as a negative response. Hence it follows that the REPORT-PDU is handled in the **Request** class and not in the **Notification** class.

2. In SNMP terminology, a PDU arriving from and sent to the transport service is called an SNMP Message. This Message consists of administrative information and the actual PDU containing the management information. In the rest of this text we will refer to this actual PDU inside the SNMP Message as the SNMP PDU to avoid conflicts with the meaning of PDU in OSI-terminology.

The **Message** class can delegate some of its responsibilities to the **PDU** and the **AdminInfo** classes.

The last five classes in Table 2 are all abstract classes. They form the five generic building blocks of the SPE module. The actual implementation is done by subclasses, which inherit the virtual methods from the abstract classes and provide code for them. Some possible subclasses are mentioned in the description of the responsibities of the abstract classes (BER, PER, DES, IDEA, etc.; see Table 2).

With this approach, evolutionary changes will primarily apply to the above mentioned subclasses; the classes listed in Table 2 are not likely affected by such changes (only AdminInfo may need modification, namely when additional administrative information and corresponding procedures are defined). For instance, suppose you have a class **UDP**, which inherits from the **TF** class, and you want to make a modification to the design such that a TCP provided transport service can be used. Then you just need to create a class **TCP**, look at the virtual methods defined in the **TF** class and provide TCP functionality for these methods. All other classes won't need any changes. The **TF** class serves as a framework for all modules that are related to the transport service. A similar argumentation applies to the other abstract classes.

Since the **TF** class is used for communication with the transport service, part of its functionality is also needed in the scheduler. The scheduler is responsible for monitoring the input sources and needs an interface to the transport service for that. We decided not to let the scheduler use the **TF** class directly, but force it to go via the SPE, because the transport functionality is an essential part of the SNMP specification and thus of the SPE. It would disturb modularity if the scheduler was allowed to directly use an internal module of the SPE module. Therefore, we decided to let the interface function *SNMP_receive()* perform a polling function as well, delivering its return values if data was waiting in the transport service. *SNMP_receive()* is part of the SPE module and can directly access the **TF** class without disturbing modularity. This decision affects the overall design in the sense that the two network interfaces are replaced by a single interface between the SPE module and the network (see Figure 1). An alternative was to create an additional class, which provides the network polling functions and assign this class to the scheduler. For now, this would add too much complexity for the little functionality it provides. A more sophisticated polling mechanism however could justify such an additional class.

3.3 Relationships between classes

The relationships between the classes are expressed in the class diagram of Figure 2. The notation is taken from [Booch94]. Since the methods are already mentioned in Table 2, only the class names are shown in Figure 2. In this diagram it is assumed that the SNMP agent is using UDP as the transport service, DES for encryption, MD5 for authentication and USEC as the administrative model.

The responsibility of the **SPE** class is separated into the **Request** and **Notification** classes. Both classes are physically contained in the **SPE** class and therefore the relationship between these classes is an aggregation relationship. Since the **SPE** class can be initialized before the operation (receive, send, trap) is actually known, the aggregation is by reference, allowing the lifetime of the objects to be less intimately connected.

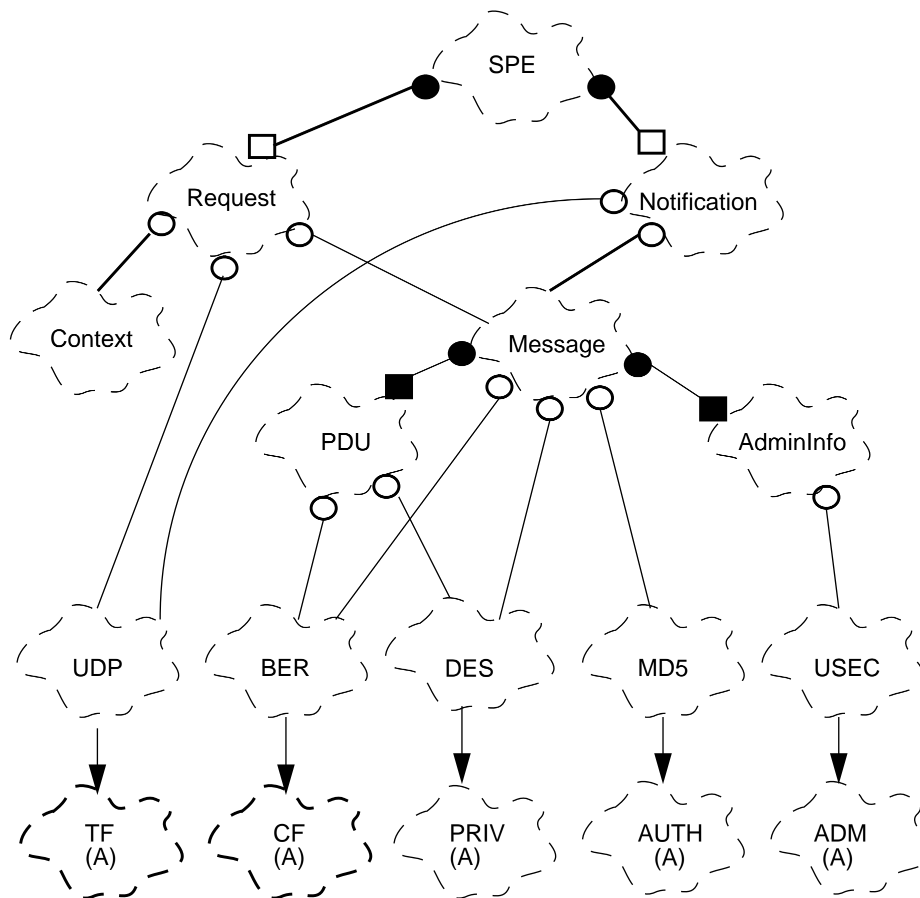

Figure 2: Class diagram of the SPE module

Both the **Request** and the **Notification** classes use the **UDP** class for the transport service and the **Message** class for the operations that need to be performed on the SNMP Message information. An SNMP Message consists of an SNMP PDU and an administrative part, leading to a whole/part (aggregation) relationship. This aggregation relationship is by value, because an

SNMP Message does not exist without both the aggregated classes. Both the **Message** class and the **PDU** class then have a 'using' relationship with the **BER** and **DES** classes. The **AdminInfo** class has a 'using' relationship with the **USEC** class, and the **Message** class has one additional 'using' relationship, namely with the **MD5** class, because authentication is done on the SNMP Message as a whole.

In Figure 2, **UDP**, **BER**, **DES**, **MD5** and **USEC** form a level of subclasses where functionality can be added or replaced. The constructor of the **SPE** class must have parameters to indicate what functionality is requested and according to those parameters, an object will be instantiated from one of the available classes. As mentioned in Section 3.2, these classes have inheritance relationships with the five abstract classes at the bottom of the diagram in Figure 2.

4 Perl implementation

4.1 Implementation decisions

An often mentioned advantage of OO design is that all major design decisions are taken in the analysis and design phases, making the implementation phase easier and more straightforward than with a functional approach, where decisions are often postponed to the implementation phase. Some decisions however still need to be made during the implementation phase, mainly with regard to the chosen implementation language and the machine the software must run on. Both may introduce restrictions that must be taken into account. Whereas the analysis and design phases focus upon the outside view of abstractions, the implementation phase focuses upon their inside view and how to accomplish the expected behaviour.

When using an interpreted, higher-level language as Perl, one should be careful to overestimate the importance of using that language for everything. Some things can always be done easier in a lower-level language as C. Also, when performance is critical, a lower-level language is to be preferred, because of its faster execution time. Since one of the goals of the project was to increase implementation experience with relatively new languages, and performance was not considered a major requirement, it was decided to use Perl as much as possible. However, implementations of certain functions do already exist in C, such as BER, MD5 and DES. In Perl, it is possible to dynamically load C routines, allowing them to be called as procedures in Perl, totally transparent to the user of the module. A special extension language, called XS, must be used to create the 'glue' between the C and Perl code.

We decided to use standard C implementations of MD5 and DES. An XS interface was available for these implementations too. For the BER module, the choice was between a MIB-

compiler (SNACC), an existing C module implementing BER, and an existing Perl module implementing BER. We decided to use the Perl module, again because performance wasn't a major issue, and because of the overhead a MIB compiler would give. In Perl, the code can be changed easily and the hassle of compiling new MIB definitions *and* compiling the generated C code for every change is avoided. Furthermore, the code needed a few adaptations to fit our needs and we saw that this was much easier with the Perl module, because of its very clear, well-organized structure. As the code is meant to be part of a development platform, easy understanding of the code is certainly an issue. Moreover, when a good module is available in your language of choice, it is most logical to use that module.

For the UDP module, Perl supports the socket mechanisms that the operating system supports. In this case, the operating system used was Solaris 2.5.

4.2 Modularity and object orientation in Perl

As of version 5, Perl has possibilities to cleanly divide code into modules and create reusable building blocks. This modularity and reusability is achieved by using packages, which declare pieces of code as being in a given namespace. This is a mechanism to protect modules from 'stomping' on each others variables. A module is just a package that is defined in a library file of the same name, and exports some of its subroutines to the outside world, so they can be used by other packages (see [Perlmod96]).

As Perl is intended to be practical and easy to use, requiring a minimum of development time, its OO features are not very complicated. There is no special class syntax in Perl. A package may function as a class if it provides subroutines that function as methods. A method is just a subroutine that expects its first argument to be the object or package it is being invoked on. A static method expects a class name as the first argument, a virtual method expects an object reference as its first argument.

An object is simply a reference to something 'blessed' into a class. A class typically provides a subroutine that acts as a constructor. The constructor defines a data structure containing the data members of the class, creates a reference to the data structure and 'blesses' this reference into the class. Now the reference knows which class it belongs to. A package that creates such an object can invoke methods on the object that are defined in the class (package) it belongs to. Since the object knows which class it belongs to, methods are not exported, in contrast to the functional approach.

Inheritance in Perl is implemented by a special array in each package, called **@ISA**. This array tells the package where to look for a method if it can't be found in the current package.

Both single and multiple inheritance are supported this way. In fact, the *bless()* function and the **@ISA** array are the only syntactic constructs added for OO programming. More detailed information on this can be found in [Perlobj96] and [Perlbot96].

5 Evaluation

Before embarking the OO development traject, we followed approximately the same traject using a functional (non-OO) approach. A comparison with regard to flexibility and modularity showed that it is possible to achieve about the same degree of modularity with both. The focus of modularity is however different. Whereas, in a functional approach, functionality is decomposed and related parts are grouped in modules, the OO approach focuses on identifying abstractions, yielding both data and functionality decomposition. Another difference is that with a functional approach, the relationships between the modules are formed on an equal basis. The OO approach allows a further refinement in this by providing inheritance and aggregation relationships, which provide an easier understanding and grouping of responsibilities and dependencies in the modular structure.

We think the goal of delivering a set of reusable building blocks has been achieved. By separating the responsibilities, changes will normally apply to only a small subset of the classes, notably the classes that inherit from the five defined abstract classes. The separation of PDU and administrative information makes it possible for both to evolve over time without affecting the other. These building block properties make it possible to replace pieces of the functionality with improved versions or different implementations that provide the same service, possibly in different implementation languages. Perl provides easy communication with C, while communication with other languages (Tcl/Python) is increasingly supported. As long as the modules provide the same interface, the interoperability should not be problematic.

For testing purposes, the implementation was used with a MIB module implementation, C and Tcl submodules. It cost about half an hour to make it fully functional, thanks to the interfaces we defined in the overall design. Together with the MIB module, a fully functional agent was formed which was tested with a manager application from Carnegie Mellon University (CMU). This also gave no problems, indicating that the agent is compliant with the SNMP specification.

A comparison with other languages showed that Perl is well suited for prototype development. Perl is a language intended to be practical, easy to use and effective, and although it was at first primarily used for scanning text, Perl version 5 includes many features which make it a language that can be used for virtually anything. It relieves the programmer of many

storage allocation and typing problems. However, for applications that need high performance, Perl is not the right language. Due to its interpreted nature, it's usually much slower than C.

Compared to Tcl, Perl is a more general-purpose language. Whereas Tcl is most effective as extension language and is very string-oriented, Perl has possibilities to create complex datatypes, can dynamically load C code, and provides mechanisms to divide the functionality in modules and is therefore better suited for application/tool development.

Compared to C^{++}, Perl supports an easy-to-use application of OO programming techniques. C^{++} requires understanding some of the deep and subtle points about the working of the language before it can be used effectively. Furthermore, considerable time is spent with C^{++}, as with most conventional languages, on debugging due to the low-level properties of the language.

6 Conclusions and future work

The research presented in this paper investigated the use of object oriented technology in developing a protocol entity for an SNMP agent platform. The abstractions and encapsulation coming forth from this technology proved to yield a very modular design and implementation, which is expected to be easily adaptable to changes in SNMP standardization. This aspect of the research contributed to the primary goal of the Damocles project. The secondary goal, gaining implementation experience with relatively new programming languages, is also met. Perl proved to be a very interesting language with many possibilities.

Future work may concentrate on the following topics: The current implementation only supports SNMPv1; although the design is certainly meant to support newer versions, it would be interesting to see if implementations of Community-based SNMPv2, USEC and SNMPv2* are indeed as simple to implement as this paper suggests. Further, this protocol entity is restricted to an agent's protocol stack, but it can be extended with a manager's protocol stack, thus forming a general SNMP 'service', used by both manager and agent. An alternative is to stay with agent technology and investigate the possibilities for adding more intelligence to the agent, shifting from centralized management to a more distributed form of management (hierarchical management, management by delegation). These are all interesting research topics in which a generic development platform will provide the developer with a solid basis.

The implementation may also be interfaced with other programming languages (Tcl, C^{++}, Java, Python), either for performance aspects (C/C^{++}) or to accomodate the integration of building blocks written in various languages. Interesting to note is also that a Perl compiler is

currently being developed, generating C or byte code. It certainly is desirable to keep an eye on this development.

Acknowledgements

We would like to thank our partners in the Damocles project. They provided us with an inspiring environment for fruitful discussions on the topics discussed in this article.

References

[Ananthaswamy95] A. Ananthaswamy, *Data communications using object-oriented design and C^{++}*, McGraw-Hill, 1995.

[Booch95] G. Booch, *Object-oriented analysis and design, with applications*, Benjamin/ Cummings Publshing Company, 1994.

[Case90] J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin, *Simple Network Management Protocol (SNMP)*, RFC 1157, 1990.

[Horn90] C. Horn, *Is object orientation a good thing for distributed systems*, in: Progress in distributed operating systems and distributed systems management, LNCS 433, Springer Verlag, 1990.

[ObjectAgency95] The Object Agency, Inc., *A comparison of object-oriented development methodologies*, http://www.toa.com/pub/html/mcr.lhtml.

[Perlbot96] *Perlbot - Bag'o Object Tricks (the BOT)*, Perl 5.002 documentation.

[Perlmod96] *Perlmod - Perl modules (packages)*, Perl 5.002 documentation.

[Perlobj96] *Perlobj - Perl objects*, Perl 5.002 documentation.

[Stallings93] W. Stallings, *SNMP, SNMPv2 and CMIP: the practical guide to network management standards*, Addison Wesley, 1993.