

# Developing an SNMP agent protocol entity with object oriented Perl

Arnoud Zwemmer, Eric van Hengstum, and Marten van Sinderen

CTIT, University of Twente, P.O. Box 217, the Netherlands

## Abstract

*The Damocles project at the University of Twente is developing an SNMP prototyping vehicle. 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. Whereas the transition to newer versions is attractive from a functional 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 methods and OO techniques facilitate modification and reduce the effort and cost of version transition. In this paper, we discuss an SNMP agent protocol entity written in OO Perl. We discuss the design and implementation of the protocol entity, and how this development effort contributes to the objectives of the Damocles project.*

## 1 Introduction

The Simple Network Management Protocol (SNMP; [3]) 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.

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.

This paper reports on our experience with the development of an SNMP agent protocol entity using object oriented (OO) Perl. The paper discusses the high level design of the SNMP agent which was shared by different implementation teams, 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.

## 2 Overall design

Based on the observation that storing and transport of management information 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, 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 is the system which is managed by the agent. The agent must be informed through appropriate events of error situations in the managed system and immediately act upon such events. Figure 1 depicts the overall structuring of the Damocles agent.

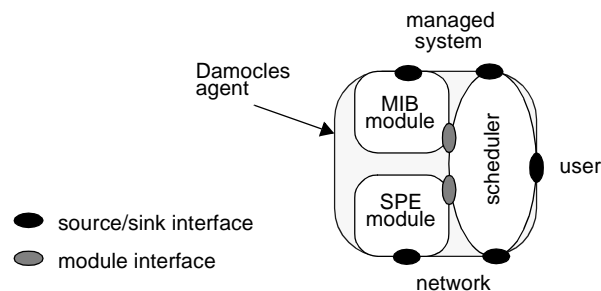


Figure 1: High-level structure of the Damocles agent

To monitor the input sources, a scheduler is 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 (see Figure 1), thus achieving maximum independence of the modules.

Two other important decisions shaped the development of the Damocles agent. Both decisions are based on the guiding principle that, during a first development cycle, attention should be focused on modularization while keeping the Damocles agent in all other respects as simple as possible. The first decision is to base the communication between modules on ordinary procedure calls, rather than on interprocess communication. The second decision is to make the Damocles agent only support one manager request at a time, rather than being able to process multiple requests concurrently. Simplicity is thus considered more important than performance. 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 and implementation**

#### **3.1 The development process**

Several 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 [4]. During the development process, we used the Booch method described in [2] as a guideline, motivated by the fact that it is well-known and widely used. The method was however not followed very rigidly. Rather, we aimed at an intuitively clear design structure, using formally defined methodologies only as a reference.

The first step in the development process forms the requirements analysis. Since we are concerned with a standard, the problem domain is clear. The procedure rules specified in the standard must be adhered to. They embody the strategic design decisions of the system; important abstractions and the necessary behaviour of the system can be established 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. In [1], an OO design approach is described for a data link protocol represented as an 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 the SPE module can be modelled as an FSM. A short investigation indicates that this is problematic. Only three events related to the SPE can be identified: the interface functions *SNMP\_receive()* for processing incoming data, *SNMP\_send()* for sending the response back, and *SNMP\_trap()* for sending traps to the manager. 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 [2], for example) that has been proven as a useful development tool in many application areas. This led to the identification of the following classes and responsibilities:

The **SPE** class is identified as the main class responsible for receiving and sending all SNMP agent PDUs. The responsibilities of the **SPE** class can be separated, leading to two contained classes: **Request**, responsible for all request/response operations, and **Notification**, responsible for notifications (traps) originating from the agent. The **Request** class must use 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. Maintenance of this context information is the responsibility of the **Context** class. Both the **Request** and the **Notification** class require the construction, interpretation and manipulation of SNMP messages. This responsibility is assigned to the **Message** class. An SNMP message consists of two components, viz. a PDU component and a component containing administrative information. This led to the identification of two contained classes, **PDU** and **AdminInfo**, to which the **Message** class can delegate some of its responsibilities. The remaining identified classes are **UDP**, **BER**, **DES**, **MD5** and **USEC**. They form the generic building blocks of the SPE module, with responsibilities as suggested by their names (which also reveal associated design choices).

### 3.3 Relationships between classes

The relationships between the classes are expressed in the class diagram of Figure 2.

The **Request** and **Notification** 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.

Both the **Request** and the **Notification** class use the **UDP** class for the transport service and the **Message** class for the operations that need to be performed on the

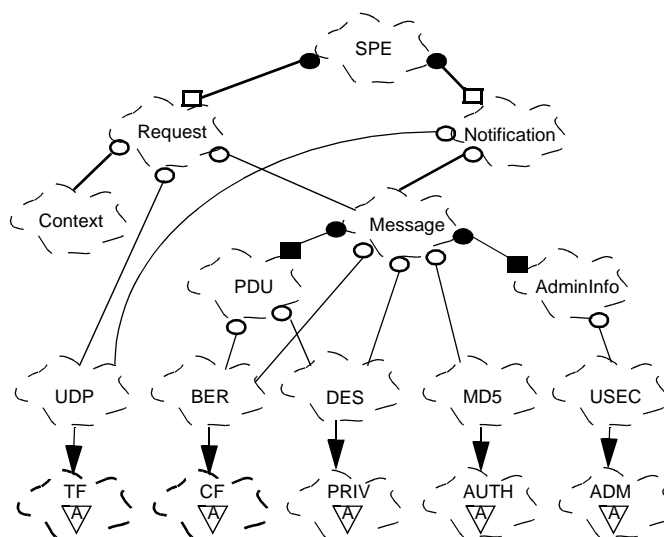


Figure 2: Class diagram of the SPE module

SNMP message information. An SNMP message consists of a 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** and the **PDU** class have a ‘using’ relationship with the **BER** and the **DES** class. The **Message** class also has a ‘using’ relationship with the **MD5** class, because authentication is done on the SNMP message as a whole. The **AdminInfo** class has a ‘using’ relationship with the **USEC** class. Figure 2 also shows the abstract classes **TF**, **CF**, **PRIV**, **AUTH** and **ADM**, with which **UDP**, **BER**, **DES**, **MD5** and **USEC**, respectively, have an inheritance relationship. The former are independent of specific design choices, and permit the latter to be easily replaced or modified.

### 3.4 Perl implementation

Perl is used as the implementation language of choice. As of version 5, Perl can be used for OO programming (two syntactic constructs, the *bless()* function and the *@ISA* array, are added for this purpose). More detailed information on this can be found in the Perl 5.002 documentation.

Perl supports the use of existing C code: C routines can be dynamically loaded and referred to by procedure calls in Perl. We could therefore use standard C implementations of MD5 and DES; for BER, a Perl module was available, which could be adapted to our needs. The other classes were all implemented from scratch in Perl. The implementation of UDP was facilitated by support in Perl of UDP socket communication.

## 4 Evaluation and future work

Before embarking the OO development trajectory, we developed an SNMP agent protocol entity 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 object abstractions, yielding both data and functionality decomposition. The OO approach also allows inheritance and aggregation relationships between objects, 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.

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.

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. Another piece of interesting future work is the extension of the current implementation with a manager protocol stack.

## References

1. A. Ananthaswamy, *Data communications using OO design and C++*, McGraw-Hill, 1995.
2. G. Booch, *OO analysis and design, with applications*, Benjamin/Cummings Publ. Comp., 1994.
3. J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin, *Simple Network Management Protocol (SNMP)*, RFC 1157, 1990.
4. The Object Agency, Inc., *A comparison of OO development methodologies*, <http://www.toa.com/pub/html/mcr.lhtml>.