Bringing AgentX Subagents to the Operating System Kernel Space

Oliver Wellnitz and Frank Strauß

Technical University of Braunschweig
Institute of Operating Systems and Computer Networks
Mühlenpfordtstraße 23
38106 Braunschweig, Germany
{wellnitz,strauss}@ibr.cs.tu-bs.de

Abstract. SNMP agents on conventional operating system platforms are mostly monolithic and implement Managed Objects in a single program. The concept of subagents makes it possible to delegate the implementation of Managed Objects to several subagents located close to the managed subsystems. All subagents are managed by a master agent. While this concept is well accepted for hardware subsystems of modular devices and for host services running in the user space, it is not yet applied for components of conventional operating systems.
This paper examines to what extent the IETF standard subagent protocol AgentX is suitable for the management of kernel components. For this purpose, on the Linux platform two subagents have been implemented within the kernel subsystems they manage. They communicate with a master agent in user space. The implemented software contains a generic intermediate layer which carries out AgentX protocol operations and access to Managed Objects. Based on this layer, the network interface subsystem and the Netfilter subsystem have been enhanced with management extensions.

1 Introduction

Network management is essential for the operation and supervision of medium to large computer networks and the Simple Network Management Protocol (SNMP) is the standard network management protocol of the Internet [1]. Current implementations of SNMP agents on conventional operating system platforms are mostly monolithic and rarely extensible. They run in user space and cover a broad spectrum of management information from high-level service management to low-level hardware device management. In many cases they have to gather information from the operating system kernel through different means such as system calls, device driver input/output control functions (ioctls) or special filesystems. These kernel interfaces can be difficult to handle because, for example, on many Unix systems the agent has to parse files from the /proc filesystem which have a structure that is likely to change in subsequent kernel revisions. These interfaces can in many cases also be incomplete in respect to the MIB which is to be implemented, e.g., they may lack attributes to uniquely identify instances of Managed Objects. Furthermore, in many cases SNMP notifications cannot be created efficiently because there is no mechanism in the methods mentioned above to notify user space processes upon certain events that occur in kernel space. So whenever an SNMP agent handles notifications, it has to use a polling strategy to gather information and detect changes itself. If the polling interval is set too small, polling data from the kernel wastes CPU time. On the other hand, setting the polling interval too large, the data in question may change two or more times during that time so that changes cannot be detected accurately. Another problem is that aside from the maintainer of a component the author of an SNMP agent must also know specific details about the managed component. If we could minimize the necessary amount of knowledge that a component maintainer needs to have about network management, he could provide a network management interface for his component all by himself, so that it is much more likely to keep the management plane in sync with the component.

A kernel implementation of AgentX subagents may overcome theses problems. Inside the kernel, a subagent has access to every available data structure. Additionally, it also can be synchronously and accurately triggered upon changes because it can call notification functions directly at the point where data is altered. Finally, a simple thus complete management interface for kernel subsystems would enable developers to implement and maintain any network management extension to their subsystem.

This paper is structured as follows: The next Section gives some background information about the subagent protocol AgentX. Section 3 describes the design and implementation of the kernel subagent architecture. It also gives some examples. Section 4 explains the implemented MIB modules and Section 5 gives a short evaluation of the presented architecture. Finally, Section 6 concludes the paper.

2 Agent Extensibility Protocol

Developed as a protocol to dynamically extend SNMP agents, Agent eXtensibility (AgentX) was published in January 2000 as an IETF Proposed Standard [2] and advanced to Draft Standard in 2002. The AgentX framework splits the role of an agent into two separate entities: A master agent, which is a traditional SNMP agent but with little or no direct access to management information, and a set of subagents, which have access to a mostly disjunct set of management information and no knowledge about SNMP. Master agent and subagents communicate through the AgentX protocol. The master agent thereby acts as a multiplexer and SNMP/AgentX protocol translator for the subagents. AgentX is transparent to SNMP managers and SNMP independent, which means that AgentX subagents can be combined with SNMPv1, SNMPv2c and SNMPv3 master agents. The upper half of Figure 2 illustrates this concept.

The design of a single master agent to which one or more subagents connect requires that the master agent is already running when the subagents are initialized.

The AgentX architecture was designed to be simple in respect of authorization, privacy and encoding. It completely leaves the first two points to the master agent, which has to ensure that only allowed mangers can access or change management information. Because subagents usually reside on the same machine as the master agent a native byte-order encoding is used instead of the BER/ASN.1 encoding of SNMP.

While it was tried to keep AgentX simple, it offers full support for data retrieval (Get, GetNext and GetBulk) and data modification operations (Set) as well as notifications (Traps). AgentX uses a multiphase-commit so that SNMP Set operations remain atomic even if the Set request comprises objects located at several subagents. Figure 1 shows the different states in an AgentX set transaction.

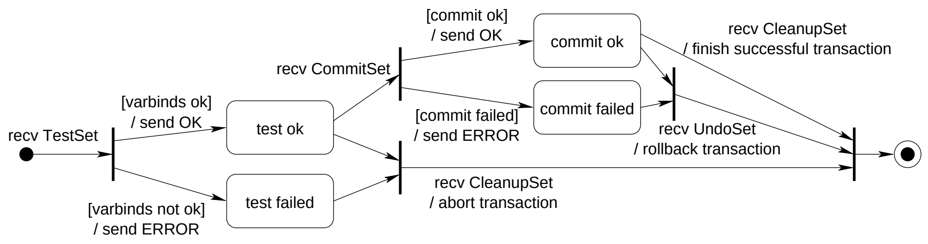

Fig. 1. AgentX Set transaction

AgentX transport mappings are specified for Unix domain sockets and TCP. Any other transport mechanism is likewise conceivable. Every AgentX connection is split up into several sessions, which in turn can convey several transactions.

There are also other subagent protocols: The SNMP multiplexing protocol SMUX [3] and the Distributed Program Interface DPI [4, 5] can be regarded as predecessors of AgentX. Furthermore, there are proprietary agent toolkits, such as EMANATE by SNMP Research Inc., that use their own master/subagent architecture.

3 Design and Implementation

Similar to traditional user space AgentX subagent toolkits such as NET-SNMP [6] and JAX [7], we propose a kernel subagent architecture which comprises two major parts: (a) a generic AgentX layer, which is MIB-unaware but omniscient in regard to the AgentX protocol and its variable types and (b) one or more management entities (kernel subagents) which implement the Managed Objects. There are no requirements for the master agent, thus any existing AgentX master agent that runs on the target platform can be used. Figure 2 gives an overview of the SNMP/AgentX framework with kernel space extensions. In the following, we focus on the kernel elements in the lower right part of the figure, the master agent and user space subagents are not regarded any further.

3.1 The AgentX Layer

The AgentX layer is a mediator between the master agent and the kernel subagents. It implements the AgentX protocol to talk to the master agent in user space and it maintains the knowledge of all available kernel subagents which supply management information. The general idea is to put everything into the AgentX layer that can be done in a generic way. The AgentX layer is divded into the following three parts, which are also shown in the grey box of Figure 2.

The socket layer forwards all AgentX PDUs between the protocol layer and the master agent. Its interface is very small and can easily be exchanged or modified to support other AgentX transport mappings.

The protocol layer is able to parse AgentX PDUs received from the master agent and to create AgentX PDUs, which are then sent through the socket layer to the master agent.

The session layer decides which kernel subagent is responsible for a request that has been parsed by the protocol layer. Similarly, the session layer assigns responses and notifications from kernel subagents to the right AgentX sessions and passes them up to the protocol layer. For this purpose, every kernel subagent has to register with the session layer prior to any AgentX communication. These registrations result in AgentX registrations.

When the master agent sends an AgentX request to the kernel, it is received by the socket layer and passed to the protocol layer which divides it into smaller pieces, which are simpler to process by the subagents. E.g., a GetNext request may contain more than one SearchRange so that multiple object instances can be retrieved by one request. This is handled as follows: The protocol layer creates an empty Response PDU. Then it takes the first SearchRange and dispatches it via the session layer to the corresponding kernel subagent. The response is appended to the Response PDU. This procedure is repeated until all SearchRanges were processed.

In contrast to userspace subagents where it is easy to ensure that the master agent is started first, the kernel and many of its subsystems are obviously initialized before the master agent program can be started. The implemented solution to this problem is to decouple the registration of kernel subagents to the AgentX layer from the AgentX registration to the master agent. This allows the kernel AgentX layer to delay the AgentX connection. Once the master agent is running a signal has to be sent to the kernel AgentX layer in order to setup the connection. The upper part of the sequence diagram in Figure 3 illustrates this procedure.

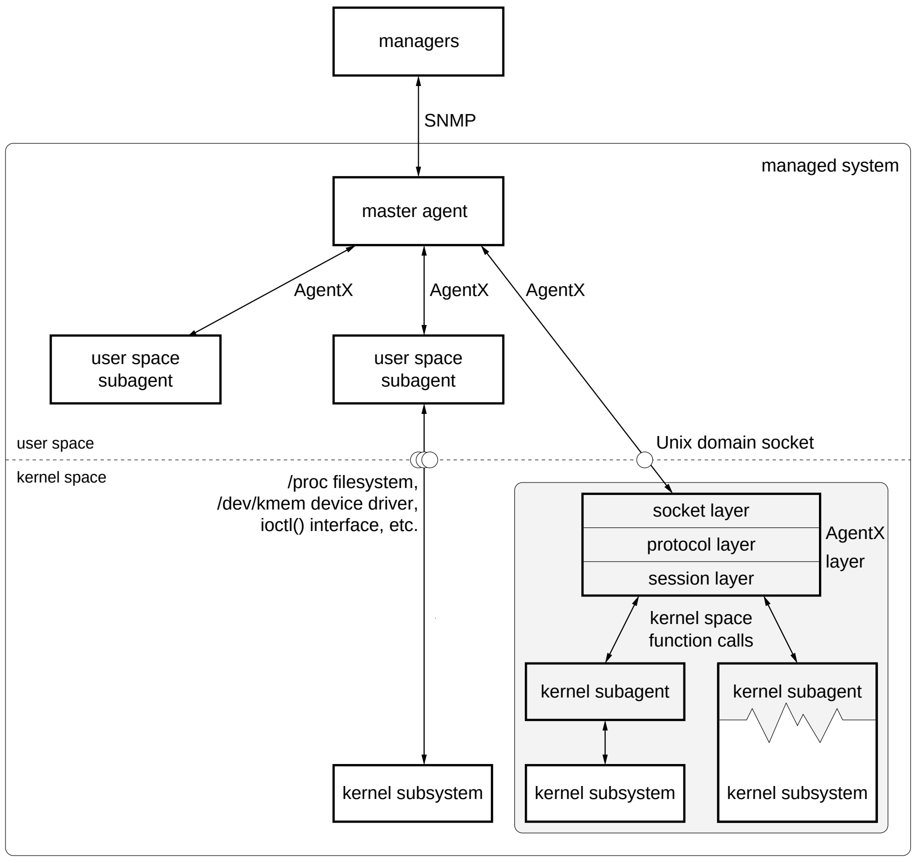

Fig. 2. SNMP/AgentX architecture with kernel space subagents

3.2 Kernel Subagents

Kernel subagents are closely attached to the kernel subsystems for which they implement the Managed Objects. Figure 2 shows the two different approaches of kernel subagents: The subagent can either be closely integrated within the kernel subsystem code, or the kernel subagent can be implemented separated from the kernel subsystem, e.g., as a separate kernel module, if the subsystem supports appropriate kernel level interfaces.

In both variants, a kernel subagent contains notification emitting functions and request callback functions. The notification emitting functions are called by other kernel functions upon certain events, so that they can construct a notification message and pass it to the AgentX layer. If the subagent is integrated with the managed kernel subsystem, notifications can easily be triggered from those functions that actually process kernel data in a way that should raise a notification. On the other hand, if the kernel subagent is implemented as a separate module, it depends on the kernel subsystem to offer hooks so that the subagent can register for events that potentially raise notifications. However, such hooks are more likely to be available within kernel space than for feedback to traditional user space agents.

As described in Section 3.1, callback functions are registered with the AgentX layer at startup. They are called for all Get and Set requests. The signature of a callback function is defined as follows:

```
errorcode Callback(in oid, in method, in context, out result)
```

A callback function gets a single request *oid*, a *method* specifier (which is a set of named flags) and the SNMP *context* as input arguments. A buffer in which the callback functions returns a *result* is passed as the fourth argument. Every callback function returns an *errorcode*. The *oid* is the starting OID of a SearchRange or the exact OID of a variable binding (varbind). It always lies within the range for which the callback function was registered to the AgentX layer. When the AgentX layer receives an AgentX GetNext PDU and dispatches a SearchRange, it compares the starting OID of the SearchRange to the callback functions' registration points. If the starting OID is a lexicographical predecessor compared to the registration point of a callback function, it uses the registration point as the value for the *oid* argument and sets the INCLUDE flag in the *method* argument.

AgentX PDU	*method* parameters
Get	GET & EXACT
GetNext	GET or GET & INCLUDE
GetBulk	GET or GET & INCLUDE
TestSet	TESTSET
CommitSet	COMMITSET
UndoSet	UNDOSET
CleanupSet	CLEANUPSET

Table 1. AgentX PDU type to method translation

Callback functions are called with one out of five different methods, which is specified by an according flag in the *method* argument. There is one GET method for Get, GetNext and GetBulk requests, and four methods for the phases of Set transactions (TESTSET, COMMITSET, UNDOSET, and CLEANUPSET). Additionally, the *method* argument can hold two flags which describe the interpretation of the *oid* argument: The flag INCLUDE signals that the search range includes *oid* itself, if it is not set *oid* is excluded. The flag EXACT signals that the request is for the exact *oid* and not for any successor. AgentX GetBulk requests with Repeaters are split by the AgentX layer into several invocations of callback functions. This streamlines and simplifies the architecture, because for

GetBulk Repeaters the results may come from different callback functions. It is not a performance issue, since it affects only neglectable local function calls and not PDUs that would have to be transmitted. Table 1 shows how all AgentX request PDUs are translated into *method* parameters to callback functions.

3.3 Processing Get/GetNext/GetBulk Requests

The middle part of Figure 3 shows a sequence diagram that illustrates the processing of a GetNext request which is similar to the processing of Get or GetBulk requests. Some details of the socket layer and the protocol layer are omitted here.

The function `ax_dispatch_getnext()` in the protocol layer dispatches every SearchRange contained in the GetNext request. It then forwards the request for every single SearchRange to `ax_dispatch_sr()` in the session layer. This function now iterates over all callback functions for the session and compares their registration OIDs to the starting and ending OID in the given SearchRange. Every matching callback function is invoked until a callback returns a valid result.

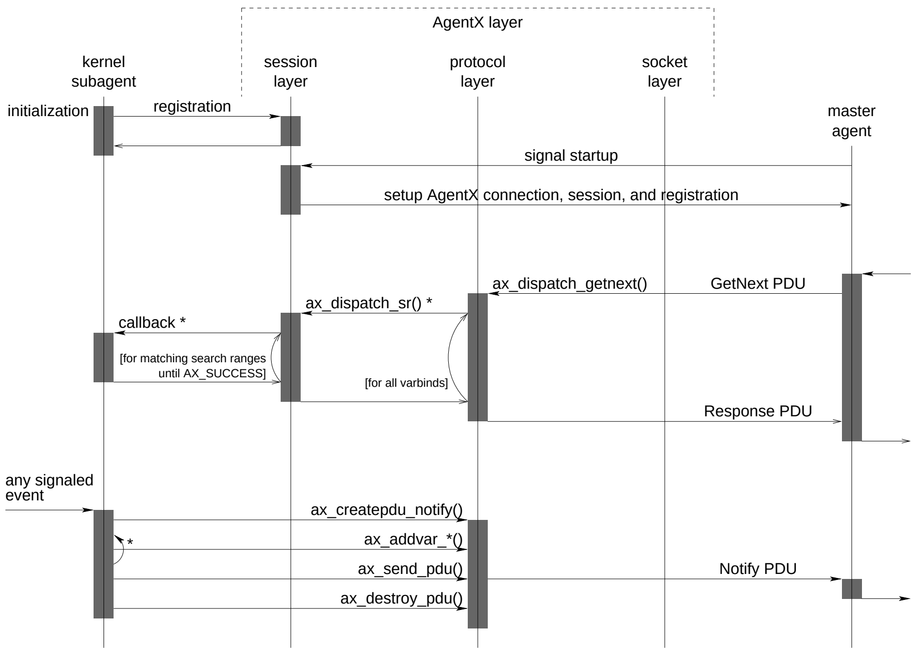

Fig. 3. Sequence diagrams: (a) startup procedure, (b) processing a GetNext request, (c) emitting a notification

3.4 Processing Set Transactions

The AgentX protocol accomplishes a successful Set transaction in three phases (see Figure 1). This is done to preserve the atomic nature of an SNMP Set request that may span multiple objects even at multiple subagents. However for the AgentX layer, Set transactions are very similar to Get requests, because varbinds are dispatched in the same way as SearchRanges. In case of success, a Set transaction consists of three request PDUs: a TestSet PDU to prepare the write access, a CommitSet PDU to actually write the data and a CleanupSet PDU to complete the transaction. The next example describes how set transactions are safely dispatched to the kernel subagents.

An AgentX TestSet PDU initiates a Set transaction. Every varbind from the TestSet PDU is dispatched to callback functions as explained before for SearchRanges in Get requests. Please notice that varbinds from the TestSet PDU may be dispatched to different callback functions. The callback functions check the varbinds for valid write access, type correctness, and legal values. When the checks have been succeeded, the AgentX layer saves all varbinds from the TestSet PDU because they are needed later. In the second phase, the master agent sends a CommitSet PDU to actually trigger the data change. Because the CommitSet PDU does not contain any varbind data, the AgentX layer refers to the previously saved varbind list to dispatch the CommitSet PDU to all corresponding callback functions. Finally in the third phase, the master agent sends a CleanupSet PDU. Similar to the commit phase, it is dispatched to the callback functions based on the saved varbind list to release any remaining temporary data of the transaction.

3.5 Sending Notifications

While Get and Set requests are handled by callback functions, notifications are initiated by kernel subagents. Hence, the AgentX protocol layer provides functions to kernel subagents to create a Notify PDU, to add an arbitrary number of varbinds of specific base types to the PDU, and to send the PDU to the master agent. Finally the kernel subagent has to release the PDU data from the protocol layer. This procedure is illustrated in the lower part of Figure 3.

4 Implemented MIB Modules

In order to evaluate the feasibility of the presented kernel subagent architecture it was implemented for the open-source Linux operating system. In addition to the AgentX layer, two MIBs were partially implemented: The `ifTable` and `ifXTable` of the Interfaces Group MIB [8] and a newly defined MIB [9] for the Linux Netfilter subsystem [10, 11].

4.1 The Interfaces Group MIB

The Interfaces Group MIB (IF-MIB) defines objects for managing network interfaces. Our implementation accesses existing kernel data structures directly or through functions already provided by the Linux networking code. It can notice changes of network interfaces through an already existing notification hook, which makes the design of a separated subagent module (see the left part of the grey box in Figure 2) feasible. Hence, the kernel IF-MIB subagent was implemented as a Linux kernel module.

However, there are two objects for which no equivalent variables exist in the kernel: a timestamp for the last status change of an interface (`ifLastChange`) and the value of `ifLinkUpDownTrapEnable`. The latter is to specify whether to send a notification if an interface changes its status. Because a kernel module cannot extend existing data structures, the IF-MIB module introduces an interface shadow list where these values are stored. The elements of this list are created on demand so that the list contains only interfaces which have non-default values on any of these two objects. In addition to readable objects, this module implements the `linkUp` and `linkDown` notifications and write access to the `ifLinkUpDownTrapEnable` and `ifAdminStatus` objects.

```
int if_mtu(ax_oid *oid,
        ax_method method,
        char* context,
        ax_variable *res)
{
        u_int32_t ifid;
        const ax_oid IFMTU =
                {10, {1, 3, 6, 1, 2, 1, 2, 2, 1, 4}};

        if (!(m & GET))
                return AX_NOTWRITABLE;

        ifid = get_ifid(oid, method, 4);
        if (!ifid)
                return AX_NOSUCHOBJECT;

        res->oid                = ax_oid_addint(IFMTU, ifid);
        res->type               = AX_VB_INT;
        res->value.integer      = getmtu(ifid);

        return AX_SUCCESS;
}
```

Fig. 4. IF-MIB: Callback function for `ifMtu`

Figure 4 gives an example of a callback function. It covers the table row `ifMtu`, the Maximum Transfer Unit. The IF-MIB defines this as a read-only object, hence this function returns an error on Set transactions. The function `get_ifid()` finds the correct interface ID for the requested OID. The function `getmtu()` is called to retrieve the MTU of the interface. Finally the full instance OID, the type and the value are stored to the result structure, before the callback function is terminated successfully.

4.2 The Netfilter MIB

The second implemented MIB is the experimental TUBS IBR Linux Netfilter MIB [9]. Netfilter is the Linux subsystem for packet filtering, mangling and network address translation (NAT).

The Linux Netfilter subsystem consists of so called tables. As of Linux 2.4.20 there are three tables in the Linux kernel: The packet filtering table, the network address translation table and a mangle table for packet alteration. Each table contains a number of built-in chains and may additionally have user-defined chains. The Netfilter subsystem currently has five hooks at five different points where an IP packet can traverse (INPUT, FORWARD, OUTPUT, PREROUTING and POSTROUTING). So each table has up to five different built-in chains. Every chain consists of a list of rules where each rule consists of one or more conditions (matches) and an action (target). If a packet matches all conditions, the according target is applied. Each built-in chain has a default policy which decides the fate of an IP packet that does not match any rule.

The Linux Netfilter subsystem is divided into several modules, which are responsible for different tasks. One module handles all table and chain management (`ip_tables.c`) so that the according network management functionality has been implemented there. This module defines data structures for Netfilter tables and chains but it, e.g., lacks methods to access a specific chain. This is done in user space by the Netfilter configuration tool *iptables(1)*. This approach is usually preferred because it keeps the kernel code small, but on the other hand, it makes the task of adding network management code to the kernel more difficult. So existing user space functions had to be rewritten and put into the kernel. Furthermore, the Netfilter MIB contains "LastChange" timestamp objects for all Netfilter elements, which the original Netfilter subsystem does not support. For this reason the data structures for Netfilter tables and chains had to be extended. This raised a problem, because as mentioned before, these data structures are used by the user space tool as well, so that adding timestamps to tables and chains breaks compatibility with the Netfilter user space tool. The solution to this problem is either to recompile the user space tool with the new structure definitions or to drop the "LastChange" objects. Finally, both approaches were implemented and the system administrator can decide at kernel compile time.

5 Evaluation

A brief evaluation was done in order to see if the presented kernel subagent architecture has measurable impact on the performance in contrast to a monolithic SNMP agent. For this purpose we used a Pentium 200MMX host with 64 MB memory running either a standalone NET-SNMP agent or the presented kernel

AgentX prototype with a NET-SNMP daemon as the AgentX master agent. All SNMP requests were issued by a remote manager over a local area network.

	kernel subagent		monolithic agent	
	mean	std.dev.	mean	std.dev.
snmpwalk ifTable	323 ms	11 ms	351 ms	7 ms
10× snmpget ifNumber	737 ms	20 ms	739 ms	34 ms

Fig. 5. Performance comparison

Figure 5 shows the results of the evaluation which presents no distinctive difference in performance. However, the implementation of `ifTable` is not equivalent in both approaches: the NET-SNMP implementation of `ifTable` supports 17 columnar objects while the kernel subagents supports 20 columns. Therefore in the snmpwalk test the kernel implementation returned 81 object instances for four table rows compared to 69 in case of the NET-SNMP implementation.

The implementation costs for kernel subagents turned out to be relatively small. E.g., the `ifTable` and `ifXTable` implementation presented in Section 4.1 comprise of about 900 lines of C code.

6 Conclusion

This paper states that the concept of distributed SNMP agent implementation by means of the AgentX subagent architecture is well applicable not only to modular devices and host services in user space, but also to kernel space subsystems. It has been argued that this allows MIB implementors to retrieve and manipulate data that lives in kernel space without the indirection of a potentially changing and limited kernel interface. Furthermore, this way it is easier to handle events in the kernel space in order to emit notifications for which user space agent implementations would often need to poll data frequently from the kernel due to the lack of appropriate trigger mechanisms.

To evaluate the concept of kernel AgentX subagents, two MIBs have been implemented for the Linux 2.4.x kernel: the IETF IF-MIB and a newly designed MIB module for the Linux packet filtering subsystem Netfilter. It has been shown that the effort to instrument typical kernel subsystems with subagent functionality is low and that the intervention to the existing kernel code could be restricted to a small interface. Furthermore it has been proved that the performance is at least comparable to traditional agent implementations.

The downside of implementing management agent functionality inside the kernel is the increased size of static kernel code and the general fact that kernel level code development is a delicate task, because bugs affect system stability more severely than user space programs.

The major benefit of the presented approach is the fact that the development of a kernel subsystem and its management instrumentation can be closely integrated. The development of a MIB implementation is put in the hands of the maintainer of the subsystem which is to be managed through the MIB. This expertise helps to ensure more accurate MIB implementations and eases to keep track of changes in a subsystem which affect the management portion. At the same time the AgentX layer releases the developer from the necessity to know SNMP in detail.

References

1. J. Case, R. Mundy, D. Partain, and B. Stewart. Introduction to Version 3 of the Internet-standard Network Management Framework. RFC 2570, SNMP Research, TIS Labs at Network Associates, Ericsson, Cisco Systems, April 1999.
2. M. Daniele, B. Wijnen, M. Ellison, and D. Francisco. Agent Extensibility (AgentX) Protocol Version 1. RFC 2741, Digital Equipment Corporation, IBM T. J. Watson Research, Ellison Software Consulting, Cisco Systems, January 2000.
3. M. Rose. SNMP MUX Protocol and MIB. RFC 1227, Performance Systems International, May 1991.
4. G. Carpenter and B. Wijnen. SNMP-DPI: Simple Network Management Protocol Distributed Program Interface. RFC 1228, T.J. Watson Research Center, IBM Corp., May 1991.
5. B. Wijnen, G. Carpenter, K. Curran, A. Sehgal, and G. Waters. Simple Network Management Protocol Distributed Protocol Interface Version 2.0. RFC 1592, IBM T.J. Watson Research Center, Bell Northern Research Ltd., March 1994.
6. The NET-SNMP home page. WWW Page. http://www.net-snmp.org.
7. F. Strauß, J. Schönwälder, and S. Mertens. JAX - A Java AgentX Subagent Toolkit. In *Proc. 1st IEEE Workshop on IP-oriented Operations & Management*, Cracow, September 2000.
8. K. McCloghrie and F. Kastenholz. The Interfaces Group MIB. RFC 2863, Cisco Systems, Argon Networks, June 2000.
9. F. Strauß and O. Wellnitz. The experimental TUBS-IBR Linux Netfilter MIB. http://www.ibr.cs.tu-bs.de/arbeiten/strauss/kagentxd/TUBS-IBR-LINUX-NETFILTER-MIB, 2002.
10. Pat Eyler. *Networking Linux: A Practical Guide to TCP/IP.* New Riders Professional Library, 2001.
11. K. Wehrle, F. Pählke, H. Ritter, D. Müller, and M. Bechler. *Linux Network Architecture.* Prentice Hall, 2004.