AGENT++

An Object Oriented
Application Programmers Interface
for Development of SNMP Agents
Using C++ and SNMP++

Version 2.1

Frank Fock

Contents

1 Introduction

Various Simple Network Management Protocol (SNMP) Application Programmers Interfaces (APIs) exist which allow the creation of network management applications. SNMP++ is such an API but in contrast to many others it offers the advantages of object oriented programming. An object oriented approach to SNMP network programming provides many benefits including ease of use, safety, portability and extensibility. The SNNP++ source code (C++) is freely available from the Hewlett Packard Company WWW server[1] as long as their copyright notice is preserved. SNMP++ is designed to support the development of SNMP manager entities in the first place. AGENT++ extends the basic concepts of SNMP++ to support the development of SNMP agents and SNMP entities playing a dual role.

If you are looking for an introduction to SNMP and SNMP MIBs, I recommend [Perkins97].

1.1 What Is AGENT++?

AGENT++ is a set of C++ classes which provides a complete protocol engine and dispatch table for the development of SNMP agents. Besides the AGENT++ API provides various C++ classes which implement base classes for scalar and table SNMP managed objects that can be customised by derivation. An additional class supports the development of proxy agents. AGENT++ is based on an extended version of SNMP++ 2.5f. For convenience AGENT++ includes ready to use classes which implement the MIB II´s system and snmp group.

1.2 Objectives of AGENT++

1.2.1 Ease of Use

AGENT++ has been designed to make the development of *simple* network management protocol agents *simple*. Using the AGENT++ API the programmer does not need to be concerned with details about SNMP protocol engine and dispatch table. The programmer can focus on implementing method routines and management instrumentation. The OO approach encapsulates and hides many internals like management of incoming SNMP requests, looking up mappings in the dispatch table, calling the appropriate method routines, sending responses and traps. Details concerning the simple network protocol itself are encapsulated by SNMP++ (see the SNMP++ specification for details).

[1] http://rosegarden.external.hp.com/snmp++

1.2.2 Rapidity of Implementation

The AGENT++ API undertakes nearly any task of a SNMP agent implementation that can be generalised. Therefore the implementation of SNMP agents with AGENT++ saves a lot of time and money. The OO approach of AGENT++ supports the generalisation of the key tasks of a SNMP agent efficiently without making the API difficult to use.

1.2.3 Extensibility

Extensions to the AGENT++ API can be done in many ways. This includes supporting new versions of SNMP (e.g. version 3) and their administrative framework, base classes for additional types of managed objects, and adding new features. Through C++ class derivation, users of AGENT++ can inherit what they like and override what they wish to overload:

- A key concept of AGENT++ is the sub-classing of managed object base classes by the user. The user overrides virtual member functions of these base classes when she or he needs a specialisation to the default behaviour of the respective class member function.

1.2.4 Portability

AGENT++ offers a maximum of portability due to the use of pure C++ without any operating system specific calls. So its only portability limitation is SNMP++ which is available for many Unix derivates and Microsoft Windows. Nevertheless if you want to implement a multi-threaded SNMP agent AGENT++ is limited to POSIX threads compatible UNIX operating systems, e.g. Solaris, Digital Unix, and Linux. The public interface of the AGENT++ API remains the same across any platform. *A programmer who codes to AGENT++ does not have to make changes to move it to another platform.*

2 An Introductory Example

To get an idea of what benefits AGENT++ offers to a SNMP agent programmer, here is an example that shows the implementation of an agent for a coffee-percolator. For now, a single scalar managed object which represents the temperature of the coffee in the coffee-pot will do. Additional managed objects e.g. those that demonstrate how SNMP tables can be implemented with AGENT++ will follow later on.

2.1 Explanation of Introductory Example

The main procedure of an AGENT++ SNMP agent (shown in figure 1) can be divided into three sections:

1. **Setting up the SNMP protocol engine**

   First of all an extended version of the SNMP++ Snmp class (Snmpx) is used to create a SNMP session that will be used for incoming SNMP requests. The example uses the standard SNMP port 161 to listen on, although any (available) UDP port can be chosen. If the session has been created successfully the session object snmp has to be registered to the static class RequestList which queues and manages SNMP requests.

2. **Create and register the MIB objects**

   The mib object is created which represents the conceptual database containing the management information of the agent (Management Information Base). Then all objects that should be supported by the agent´s MIB are added. Each such object contains one or more instances of managed objects related to a subtree of the management information tree. The classes systemGroup, snmpGroup, and trapDestGroup are provided by AGENT++, which include the managed objects of the MIB II´s system and snmp group, as well as managed objects needed for trap destination registration.

3. **Loop forever for incoming SNMP requests**

   The receive method of the RequestList object is used to wait for incoming SNMP request. The method takes as argument the maximum time to wait for a request in seconds. Note: If the agent uses multi-threading the RequestList can contain more than one request. A SNMP request is processed through the process_request method which propagates each sub-request to the appropriate MIB object.

How a simple managed object like coffeeTemperature can be implemented using the AGENT++ API is shown by figures 2 and 3. The class coffeeTemperature is derived from MibLeaf which is itself derived from MibEntry. MibLeaf is the base class for all instances of managed objects that are leafs of the management information tree, also called scalar managed objects. The constructor of coffeeTemperature calls the constructor of its base class MibLeaf with three arguments:

- the object identifier of the managed object´s *instance* the MIB object is representing.
- the access rights to be used (coffeeTemperature is read only)

```
...

main (int argc, char* argv[])
{
        int status;
        Snmpx snmp(status, 161);  // create SNMP++ session

        if (status != SNMP_CLASS_SUCCESS) {      // if fail print error
                cout << snmp.error_msg(status);  // message
                exit(1);
        }
        RequestList::set_snmp(&snmp);   // register the Snmpx object to
        // be used by the global RequestList for incoming SNMP requests

        Mib mib;                  // create the agents MIB

        mib.add(new systemGroup());  // add sysGroup and snmpGroup
        mib.add(new snmpGroup());    // (provided with AGENT++) to mib
        mib.add(new coffeeTemperature());     // add temperature MO
        mib.add(new coffeeSchedulingTable()); // add scheduling table
        ...

        Request* req;             // pointer to an incoming SNMP request
        for (;;) {                // loop forever (agent is an daemon)

                // wait for incoming request max 120 sec
                // and then just loop once
                req = RequestList::receive(120);

                if (req) mib.process_request(req); // process the request
        }
}
```

Figure 1: An example for a main routine of a SNMP agent to manage a coffee-percolator.

- the syntax in form of a from SnmpSyntax derived object that represents an object oriented view of Structure of Management Information (SMI) Abstract Syntax Notation (ASN.1) data types which are used by SNMP (see figure 4 or the SNMP++ specification for sub-classes of SnmpSyntax).

Because coffeeTemperature has not a constant value, we need to redefine the virtual member function get_request inherited from MibLeaf to get the actual temperature of the coffee from the management instrumentation. To translate between the different management information representations used by the management instrumentation (temperature measured in Fahrenheit) and the SNMP MIB definition (Celsius) the method get_coffee_temperature is used.

The coffeeTemperature::get_request method first gets the actual temperature and stores it in value which is a pointer to the SnmpSyntax object that has been given as third parameter to the MibLeaf constructor. Then MibLeaf::get_request is called to actually answer the SNMP request using the value of value. That´s all! Everything else is done by AGENT++.

```
class coffeeTemperature: public MibLeaf {
public:
        coffeeTemperature();
        void   get_request(Request*, int);
private:
        int    get_coffee_temperature();
};
```

Figure 2: Class definition for the managed object representing the coffee temperature

3 AGENT++ Features

- **Power and Flexibility**

  AGENT++ (in conjunction with SNMP++) provides a power and flexibility which would otherwise be difficult to implement and manage. A programmer using the AGENT++ API can concentrate his affords on the management instrumentation and method routines. He or she needs nothing to know about a SNMP protocol engine or dispatch table. Even the tricky task to manage dynamic SNMP tables that follow the SMIv2 row status textual convention is undertaken by AGENT++.

```
coffeeTemperature::coffeeTemperature():
        MibLeaf("1.3.6.1.3.100.2.1.0", READONLY, new SnmpInt32(0)) { }

int coffeeTemperature::get_coffee_temperature()
{
        // (CoffeePercolator is assumed as a static class and contains
        //  the management instrumentation for the coffee-percolator)
        // get temperature from instrumentation and
        // convert it from Fahrenheit to Celsius
        return (CoffeePerculator::get_temperature()-32)*5/9;
}

void coffeePercolator::get_request(Request* req, int ind)
{
        // store the actual temperature in value (derived from MibLeaf)
        *((SnmpInt32*)value) = (long)get_coffee_temperature();

        MibLeaf::get_request(req, ind); // use default behaviour of
        // scalar managed objects for get_requests which answers the
        // snmp get/getnext request using value
}
```

Figure 3: Class implementation of the managed object representing the coffee temperature

- **Multi-Threaded Request Processing (New!)**

  A MIB object´s method routines and thus its management instrumentation can run in an extra thread (pseudo) parallel executed to the agent´s main process. So time intensive calculations or information retrievals do not block the agent. This could increase the availability and reliability of the agent.

  Version 1.x of AGENT++ supported multi-processing. The use of multi-processing has had the following disadvantages:

  - Multi-processing needs a significant larger amount of system resources (i.e., memory and CPU) than multi-threading.
  - Processes need to communicate with each other. For this purpose System V Inter Process Communication (IPC) has been used by AGENT++ 1.x. A lot of coding had to be done to get this communication working, nevertheless the IPC could not completely hidden. As a result, the user interface was not as simple as it could have been.

    Consequently, AGENT++v2.0´s user interface, as well as its code, is much easier to understand.
  - Multi-threading offers a (pseudo) parallel working management instrumentation access to the memory of the whole agent and its MIB. A child process has only a copy of the agents memory at the child's birth time, so it cannot recognise intermediate updates to the agent´s MIB.

  What is new: Agent++ Version 2.0 and greater supports the execution of the management instrumentation[2] *as multiple threads. Each SNMP request can be separately executed as a thread. For each instance of a MIB object (MibLeaf or MibTable) there can be only a single thread executing one of its method routines at the same time. Thus, you do not have to worry about synchronising memory reads/writes within your instrumentation methods.*

  As multi-processing is no longer supported, the methods MibEntry::update, MibEntry::result, Mib::busy, *and* Mib:process_updates *known from version 1.0x have been removed.*

- **MIB II´s System and SNMP Group**

  For the programmer´s convenience an implementation of the MIB II´s system and snmp group is provided with AGENT++.

[2]The management instrumentation is represented by the ::get_request(..), ::commit_set_request(..), etc. methods of MIB objects.

- **Proxy Support for Arbitrary Management Information Subtrees**

  An arbitrary subtree of another agent´s management information tree can be included in an agent´s MIB simply be give the other agent´s management transport address and the subtree´s object identifier when defining such an proxy MIB object. As long as the proxy subtrees are disjunctive any combination of such proxy MIB objects can be used at the same time with local MIB objects.

- **Support for Persistent MIB Objects**

  Scalar and table managed objects can be saved to disk and reloaded from it. This enables a user to easily save management information persistently.

- **Support for SNMPv1 and SNMPv2c**

  SNMP++ supports SNMPv1 and SNMPv2c thus AGENT++ likewise supports both versions. A support for SNMPv3 and its user and view based administrative model will follow in the near future.

- **Detailed Logging**

  AGENT++ has a detailed logging mechanism supporting five log classes (error, warning, info, debug, event) with each supporting up to 16 levels. This makes debugging of your agent very easy and gives you full control about what is happening when your agent is running.

4 An Introduction to SNMP++

SNMP++ is a set of C++ classes which provide SNMP services to a network management application developer. An overview over these C++ classes gives figure 4. For further documentation about SNMP++ please read the corresponding specification which is freely available[3] from the Hewlett Packard company.

AGENT++ comes with a set of subclasses for some classes of SNMP++ which are shown in figure 4. Those subclasses provide extended functionality to their SNMP++ base classes, so their class names contain an appended "x". Significant extensions and improvements have been made to Snmp by Snmpx, to Vb by Vbx, and to Oid by Oidx.

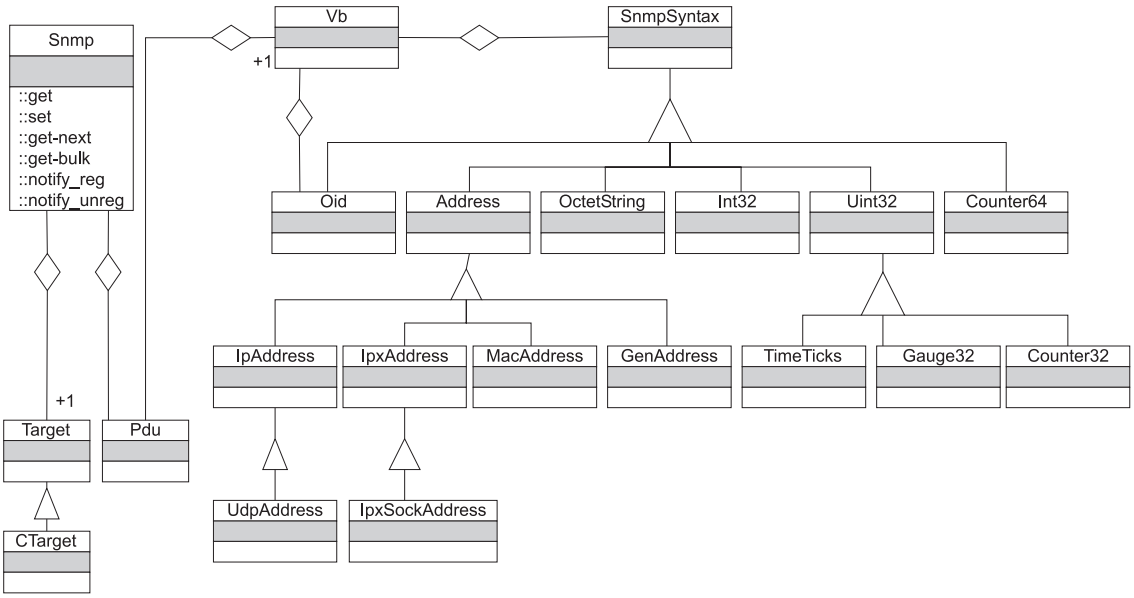

Figure 4: Object Modelling Technique (OMT) view of the SNMP++ Framework [Mellquist96]

5 The AGENT++ Classes

5.1 Mib Class

The AGENT++ Mib class represents a central part of any agent - the Management Information Base (MIB). The MIB of a SNMP agent is a conceptual database rather than a real database. An Agent has exactly one MIB, thus the Mib class is a singleton. Mib has three functional areas:

1. **Registration of MIB objects**

   Use the add member function to add a MIB object (any C++ object derived from MibEntry) to the agent´s MIB. Use the remove member function to remove a MIB object from the agent´s MIB. Both functions can be used while the agent is running.

2. **Receive and Process SNMP Requests**

   Incoming SNMP requests are accepted by using the receive member function. The receive function waits for such a request until a given timeout is reached. The timeout is given in seconds and if it is zero receive looks for a pending request and returns it immediately or if is not such a request receive returns the null pointer.

[3]http://rosegarden.external.hp.com

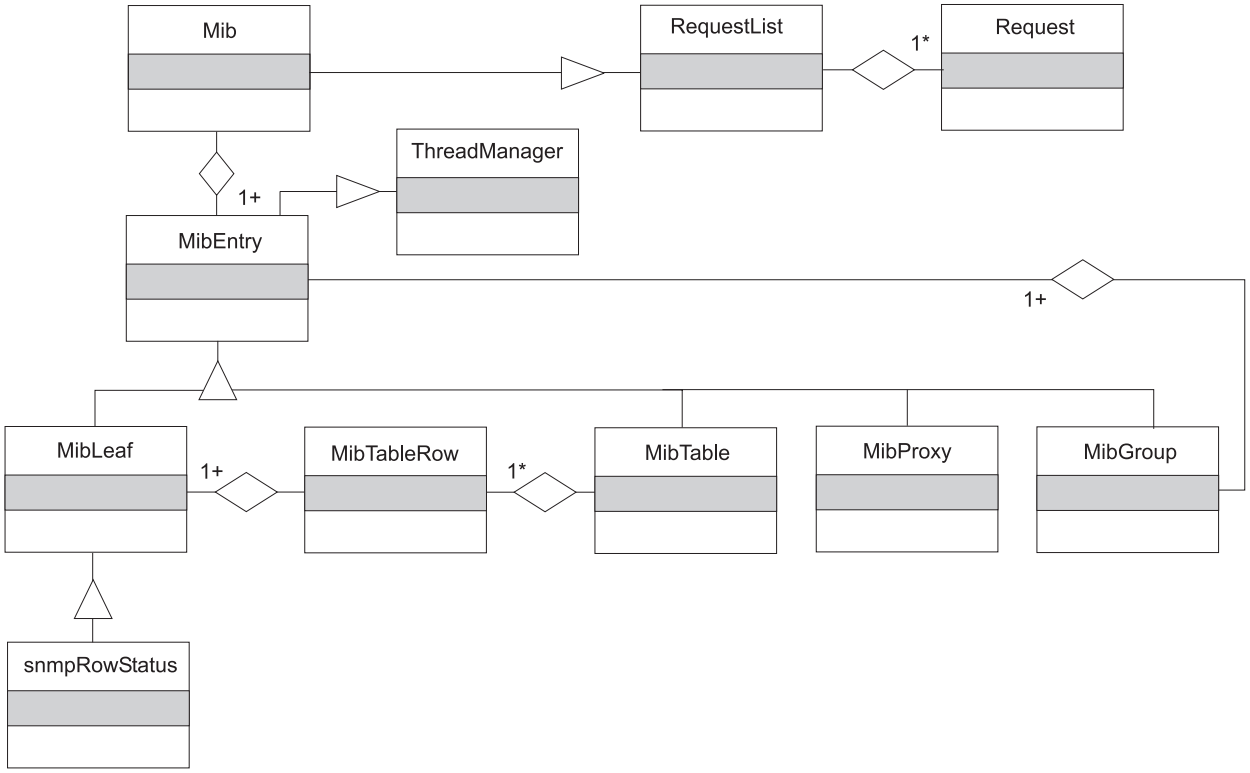

Figure 5: Object Modelling Technique (OMT) view of the AGENT++ Framework

A request is processed by the agent by calling the process_request member function of the agent´s MIB. Depending of the request type the get_request, get_next_request, or prepare_set_request, commit_set_request, undo_set_request, and cleanup_set_request of each target MIB object is called. Because the targets of SNMP requests are managed objects, but the MIB contains only MIB objects (see figure 6) the mib object determines which MIB object manages which managed object. So, process_request calls the appropriate of the above mentioned functions for each target managed object.

When the agent ist multi-threaded the method routines called by a request are executed within the same thread. This thread is then different from the master thread accepting new request.

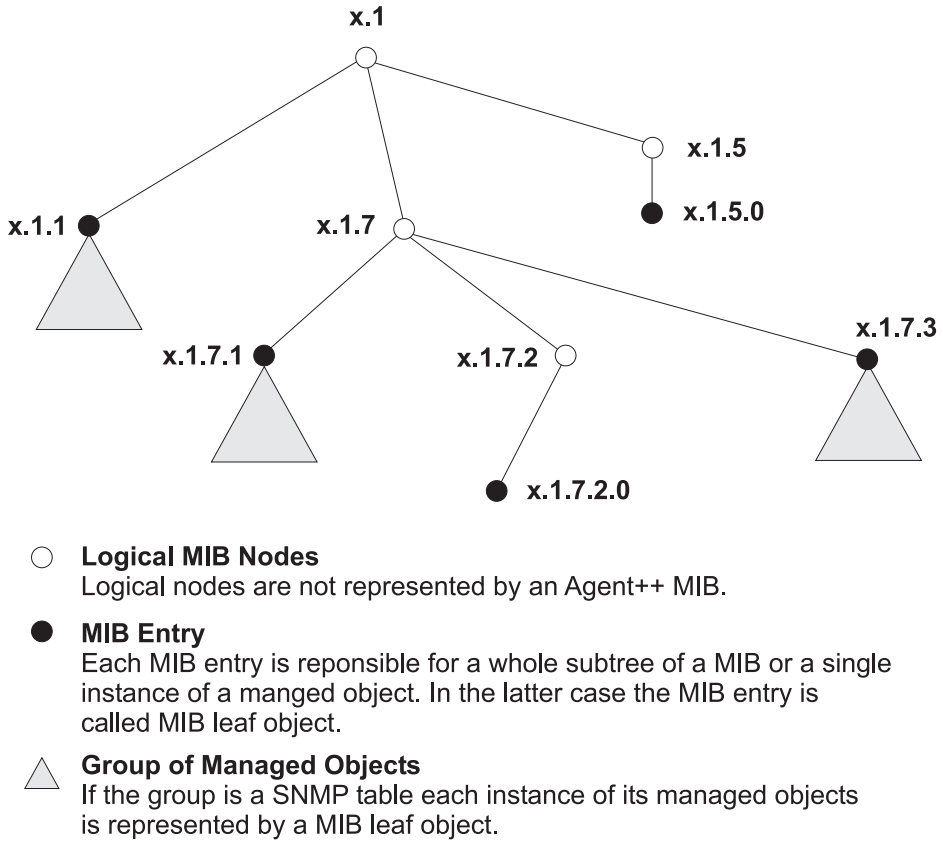

Figure 6: Example of an AGENT++ Mib Structure

The structure of an AGENT++ MIB slightly differs from the corresponding SNMP MIB structure. An Example of an AGENT++ MIB structure shows figure 6. Generally

managed objects with SNMP syntax NO-ACCESS contain no management information and are therefore not represented by AGENT++ MIB objects. These managed objects are logical nodes of the management information tree and can not be instantiated. All instances of SNMP managed objects are represented by MIB leaf objects (derived from MibLeaf).

To concentrate the management of a whole management information subtree in a single MIB object the managed object at the root of the subtree is represented by a special MIB object. This MIB object is then responsible to process SNMP requests for all the instances of managed objects within its subtree. This may be done by propagating the requests to the appropriate MIB leaf object (see MibTable class description in 5.5), but the MIB object also may answer such a request by itself.

An AGENT++ MIB thus contains two kinds of MIB objects:

1. MIB leaf objects, which represents a single instance of a managed object, and
2. MIB objects, which are responsible for the managed objects of a whole management information subtree to whom they are the root.

To propagate a GET-NEXT request to the right MIB object the mib class has to know which is actually the last managed object instance managed by a particular MIB object. Therefore each MibEntry has the member function max_key that should return the object identifier of that instance. For example: the max_key function of MibLeaf objects returns its own object id, the max_key function of MibTable objects return the oid of the last object in its table. When you create your own MIB object class do not forget to override the virtual member function max_key!

5.1.1 Mib Class Member Functions

5.1.1.1 Constructors

- Mib::Mib()

  Construct an empty MIB - persistent objects will be stored in "config/".

- Mib::Mib(const char* path)

  Construct an empty MIB - persistent objects will be stored in path.

5.1.1.2 Destructor

- Mib::~Mib()

  Destroy the MIB and all contained objects.

5.1.1.3 Configuration

- static const char* Mib::get_persistent_objects_path()

  Return the path persistent MIB objects are saved to and loaded from.

5.1.1.4 Registration of MIB Objects

- Mib::add(MibEntry* obj)

  Add the MIB entry obj to the MIB.

- Mib::remove(Oidx* oid)

  Remove the MIB object with object identifier oid from the MIB.

5.1.1.5 Request Processing

- Mib::process_request(Request* req)

  Process the request req. If multi-threading is enabled the request is processed asynchronously.

5.1.1.6 Inherited Member Functions

- Request* RequestList::receive(int sec)

  Wait (block) for the next incoming SNMP request and return it. If there is no such request within the next sec seconds return NULL.

- void RequestList::set_snmp(Snmpx* snmp)

  Register snmp to be the SNMP session that listens on incoming requests.

5.2 MibEntry Class

Any AGENT++ MIB object class has to be derived from the abstract MibEntry class.

5.2.1 MibEntry Class Member Functions

5.2.1.1 Constructors

- MibEntry::MibEntry()

  Construct an empty MIB object.

- MibEntry::MibEntry(const Oidx& oid, mib_access access)

  Construct an empty MIB object with object id oid and access rights access.

- MibEntry::MibEntry(const MibEntry& other)

  Construct a MIB object from another, copy constructor.

5.2.1.2 Destructor

- MibEntry::~MibEntry()

  Destroy the MIB object.

5.2.1.3 Object Handling

- mib_type MibEntry::type()

  Return the type of the MIB object.

- MibEntry* MibEntry::clone()

  Return a pointer to a copy of the MIB object.

5.2.1.4 Request Processing

- void MibEntry::get_request(Request* req, int ind)

  Perform the GET operation requested by the sub-request of req with index ind. When this MIB object is "multi-threaded" this method is executed as thread.

- void MibEntry::get_next_request(Request* req, int ind)

  Perform the GETNEXT operation requested by the sub-request of req with index ind. When this MIB object is "multi-threaded" this method is executed as thread.

- int MibEntry::prepare_set_request(Request* req, int ind)

  Test if the SET operation requested by sub-request of req with index ind could be executed. If so return SNMP_ERROR_SUCCESS and prepare the operation (if needed), otherwise return the appropriate SNMP++ error code.

- int MibEntry::commit_set_request(Request* req, int ind)

  Commit the prepared SET operation requested by the sub-request of req with index ind. Returns SNMP_ERROR_SUCCESS on success and SNMP_ERROR_COMMITFAIL on failure.

- int MibEntry::undo_set_request(Request* req, int ind)

  Undo the changes by a (failed) commit_set_request operation executed to process the sub-request of req with index ind. Returns SNMP_ERROR_SUCCESS on succes and SNMP_ERROR_UNDO_FAIL on failure.

- void MibEntry::cleanup_set_request(Request* req, int ind)

  Release the resources allocated for undoing the prepare/commit of sub-request req with index ind.

5.2.1.5 MIB Object Communication

The following methods can be used to automatically notify MIB objects of a change in the management information of another MIB object.

- MibEntry::add_change_notification(MibEntry* entry)

  Register MIB object entry to be notified of changes in the management information of MibEntry.

- void MibEntry::change_notification(const Oidx& oid, mib_change type)

  This method of a MIB object is automatically called if the object has been registered for notifications with add_change_notification and the management information of the instance identified by oid has changed in a way indicated by type.

5.2.1.6 Input & Output

- void MibEntry::save_to_file(const char* path)

  Save the management information contained in MibEntry (if there is any) to the file specified by path.

- void MibEntry::load_from_file(const char* path)

  Try to load management information into MibEntry that has been formerly saved by MibEntry::save_to_file into the file path. If the file cannot be found or is corrupt the actual management information of MibEntry will not be changed.

5.2.1.7 Miscellaneous

- Oidx* MibEntry::key()

  Return a pointer to the object identifier of MibEntry.

- Oidx* MibEntry::max_key()

  Return a pointer to an oid representing the last managed object instance MibEntry is managing.

- mib_access MibEntry::get_access()

  Return the maximum access rights of MibEntry.

5.3 MibLeaf Class

Generally the MibLeaf class represents an instance of a managed object, but it also can represent a columnar object of a SNMP table. A columnar object defines the behaviour of the managed object instances in a particular column of a SNMP table. The columnar object itself is not accessible in contrast to the instances derived from it (see section 5.5 for details).

MibLeaf is a sub-class of the abstract MibEntry class. As MibLeaf represents a managed object *instance* a MibLeaf object contains management information. So each MibLeaf object contains a pointer value to that management information which can be any object derived from SnmpSyntax.

5.3.1 MibLeaf Member Functions

5.3.1.1 Constructors

- MibLeaf::MibLeaf()

  Construct an empty MIB leaf object.

- MibLeaf::MibLeaf(const Oidx& oid, mib_access access, SnmpSyntax* val)

  Construct a MIB leaf object with object id oid and access rights access. The MIB leaf object´s management information will be stored in the SnmpSyntax object referenced by val. This reference will be stored in the MibLeaf::value variable.

- MibLeaf::MibLeaf(const Oidx& oid, mib_access access, SnmpSyntax* val, boolean has_default)

  Construct a MIB leaf object with object id oid and access rights access. The MIB leaf object´s management information will be stored in the SnmpSyntax object referenced by val. If has_default is true and if the MibLeaf object represents a columnar object of a SNMP table the initial value of the object referenced by val will be the default value for instances of that columnar object. The reference val will be stored in the MibLeaf::value variable.

- MibLeaf::MibLeaf(const MibEntry& other)

  Construct a MIB leaf object from another, copy constructor.

5.3.1.2 Destructor

- MibLeaf::~MibLeaf()

  Destroy the MIB leaf object.

5.3.1.3 Overloaded Member Functions

- mib_type MibLeaf::type()

  Return LEAF as type of the MIB object.

- MibEntry* MibLeaf::clone()

  Return a pointer to a copy of the MIB leaf object. *Attention! You have to refine the* clone() *method in each subclass of* MibLeaf *especially if its instances shall be used as objects within a* MibTable. *Be sure you create an instance of your subclass and return it as* MibEntry*. *Otherwise the method routines of your subclass are not called (instead the* MibLeaf::get_request(..), MibLeaf::commit_set_request(..), *etc. methods are called) when used in* MibTable *objects.*

- void MibLeaf::get_request(Request* req, int ind)

  Perform the GET operation requested by the sub-request of req with index ind by returning the management information referenced by MibLeaf::value.

- void MibEntry::get_next_request(Request* req, int ind)

  Perform the GETNEXT operation requested by the sub-request of req with index ind by returning the management information referenced by MibLeaf::value.

- int MibLeaf::prepare_set_request(Request* req, int ind)

  If the maximum access right of MibLeaf is at least READWRITE and MibLeaf::value_ok returns true for the value to be set, then return SNMP_ERROR_SUCCESS, otherwise return an appropriate SNMP++ error code.

- int MibLeaf::commit_set_request(Request* req, int ind)

  Commit the prepared SET operation requested by the sub-request of req with index ind by setting the SnmpSyntax object referenced by MibLeaf::value to its new value and saving its old value. Returns SNMP_ERROR_SUCCESS on success and SNMP_ERROR_COMMITFAIL on failure.

- int MibEntry::undo_set_request(Request* req, int ind)

  Undo the changes by a (failed) commit_set_request operation executed to process the sub-request of req with index ind by setting the SnmpSyntax object referenced by MibLeaf::value to its old value. Returns SNMP_ERROR_SUCCESS on succes and SNMP_ERROR_UNDO_FAIL on failure.

- void MibEntry::cleanup_set_request(Request* req, int ind)

  Free the saved old value of MibLeaf::value allocated for undoing the prepare/commit of sub-request req with index ind.

- MibLeaf::add_change_notification(MibEntry* entry)

  See 5.2.1.5.

- void MibLeaf::change_notification(const Oidx& oid, mib_change type)

  See 5.2.1.5.

- void MibLeaf::save_to_file(const char* path)

  See 5.2.1.6

- void MibLeaf::load_from_file(const char* path)

  See 5.2.1.6

- Oidx* MibLeaf::key()

  Return a pointer to the object identifier of MibLeaf.

- Oidx* MibLeaf::max_key()

  Return a pointer to the object identifier of MibLeaf.

- mib_access MibLeaf::get_access()

  Return the maximum access rights of MibLeaf.

5.3.1.4 Value Manipulation

- Vbx MibLeaf::get_value()

  Return a Vbx object containing the MibLeaf´s object identifier and value (a copy of the SnmpSyntax object referenced by MibLeaf::value).

- int MibLeaf::set_value(const Vbx& vb)

  Set the object referenced by MibLeaf::value to the value portion of vb if the value types are compatible and the object identifier of vb and MibLeaf are equal. If MibLeaf::value can be set return SNMP_ERROR_SUCCESS, otherwise an appropriate SNMP++ error code.

- void MibLeaf::replace_value(SnmpSyntax* val)

  Delete the SnmpSyntax object referenced by MibLeaf::value and set the reference to the object referenced by val.

5.3.1.5 Request Processing

- boolean MibLeaf::value_ok(const Vbx& vb)

  Return true if the value portion of vb can be accepted as the new value for the MibLeaf´s management information. The default implementation of the value_ok member function MibLeaf offers always returns true. Override this function in your own sub-class of MibLeaf if do not you want any value to be accepted.

  Note: value_ok is called by MibLeaf::prepare_set_request to check if a SNMP SET operation can be executed.

- void MibLeaf::set(const Vbx& vb)

  Set the management information of MibLeaf to the value portion of vb. The default implementation of set simply calls set_value. Override the set function if you want your sub-class of MibLeaf to perform additional or other actions.

  Note: set is called by MibLeaf::commit_set_request to set the MibLeaf´s management information to its new value.

- int MibLeaf::unset()

  Set the management information of MibLeaf stored in MibLeaf::value to its old value stored in MibLeaf::undo. Returns SNMP_ERROR_SUCCESS on success and SNMP_ERROR_UNDO_FAIL on failure.

  Note: unset is called by MibLeaf::undo_set_request to set the MibLeaf´s management information to its old value.

5.3.1.6 Miscellaneous

- MibTable* MibLeaf::get_reference_to_table()

  If the MibLeaf object is part of a table return a pointer to the appropriate MibTable object, otherwise return 0.

- MibTableRow* MibLeaf::get_reference_to_row()

  If the MibLeaf object is part of a table return a pointer to row object of this table MibLeaf belongs to, otherwise return 0.

5.4 MibTableRow Class

The MibTableRow class is a container class for MibLeaf objects. A MibTableRow represents a row of a SNMP table. The MibTableRow class provides functions to add MibLeaf objects to a row and functions to find and get them again. Normally a user of the AGENT++ API does not have to be concerned with MibTableRow as the MibTable class provides corresponding wrapper member functions for the above listed operations on rows. But the member functions listed in the following section can be useful anyhow.

5.4.1 MibTableRow Class Member Functions

- MibLeaf* MibTableRow::get_nth(int n)

  Return a pointer to the nth columnar object instance of this row. If such an object does not exists return 0.

- MibLeaf* get_element(const Oidx& oid)

  Return a pointer to the columnar object instance of this row with object id oid. If such an object does not exists return 0.

- snmpRowStatus* MibLeaf::get_row_status()

  Return a pointer to the row status object of this row. If the table does not contain a row status column return 0.

5.5 MibTable Class

The MibTable class is a container class for MibTableRow objects, but seen from the users view point a MibTable seems to contain only MibLeaf objects. A MibTable object must be initialised by adding to it a set of MibLeaf objects called *columnar objects*. This is best done in the constructor of your sub-class of MibTable by using the add_col member function (see section 5.5.1.3). Each columnar object is the master copy for any columnar object instance of its column. Whenever a new row is created MibTable will clone each columnar object once to build the new row. *Hence, it is necessary to redefine the* clone *method of every class derivied from* MibLeaf.

MibTable automatically sets the object identifiers of every object of a new row. If the columnar objects that are part of the index are scalar and the index has a fixed length

MibTable can set the values of the index objects of a row accordingly to its index value. Figure 7 shows an example with a fixed index length of two, so the first two columnar objects are part of the index. Because each sub-identifier of the index corresponds to its scalar columnar object the automatic index generation can be used in the shown example.

As soon as all columnar objects have been added, rows can be added to the empty table by using the add_row member function. The add_row member function needs an object identifier representing the row´s index as single parameter. The MibLeaf objects cloned from the columnar objects are then responsible for answering SNMP requests (see section 5.3.1.3). Rows can be added automatically by SNMP SET requests if the MibTable object contains a snmpRowStatus columnar object. See section 5.8 for more information about the SMIv2 row status mechanism. Rows can be removed with the remove_row member function.

How a SNMP table can be coded to AGENT++ is shown in figures 8 and 9.

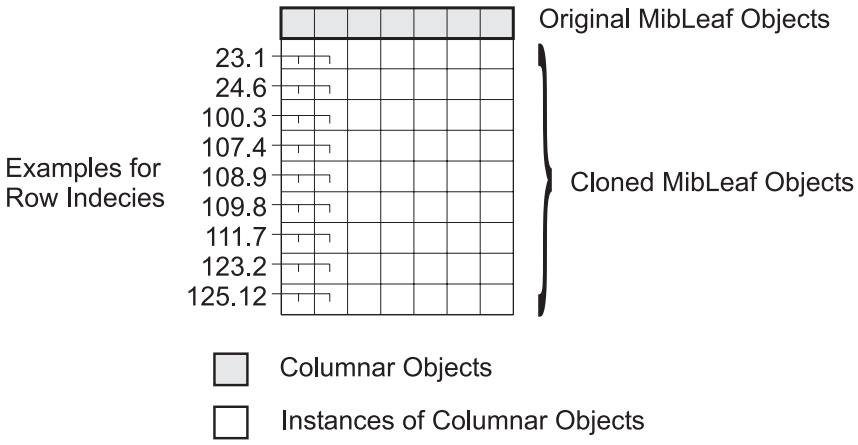

Figure 7: Structure of MibLeaf objects contained in a MibTable object

5.5.1 MibTable Member Functions

5.5.1.1 Constructors

- MibTable::MibTable(const MibTable& other)

  Create a MibTable object from another table. All columnar objects and their instances are cloned.

- MibTable::MibTable(const Oidx& oid)

  Create a empty MibTable object with object identifier oid. The given oid refers to the object identifier of the corresponding SNMP table entry´s managed object

```
...
CoffeeTime::CoffeeTime(const Oidx& o, mib_access access):
            MibLeaf(o, access, new TimeTicks(0)) { }

CoffeeCups::CoffeeCups(const Oidx& o, mib_access access):
            MibLeaf(o, access, new SnmpInt32(4), TRUE) { }

boolean CoffeeCups::value_ok(const Vbx& vb)
{
        if (vb.get_syntax() == value.get_syntax()) {
                SnmpInt32 i;
                vb.get_value(i);
                return ((i>=2) && (i<=12));
        }
        return FALSE;
}

CoffeeStatus::CoffeeStatus(const Oidx& o): snmpRowStatus(o,
        READCREATE) { }

CoffeeSchedule::CoffeeSchedule():
        MibTable("1.3.6.1.3.100.2.2.1", 1, TRUE)
{
        // first column takes the automically generated index
        add_col(new MibLeaf("1", READONLY, new SnmpInt32(0)));
        // time at which coffee should be ready (no default)
        add_col(new CoffeeTime("2", READWRITE));
        // number of cups to be made (default is 4 cups)
        add_col(new CoffeeCups("3", READWRITE));
        // row status
        add_col(new CoffeeStatus("4"));
}
```

Figure 8: Example for a Coffee-Percolator Scheduling Table (Constructors)

```
void CoffeeCups::set(const Vbx& vb)
{
        set_value(vb);
        CoffeePercolator::set_cups(my_row->get_index().first(), *value);
}

void CoffeeTime::set(const Vbx& vb)
{
        set_value(vb);
        CoffeePercolator::set_time(my_row->get_index().first(), *value);
}

void CoffeeStatus::set(const Vbx& vb)
{
        set_value(vb);
        switch (get()) {
        case rowActive: {
                CoffeePercolator::
                    add_schedule(my_row->get_index().first(),
                                 *my_row->get_nth(1)->value,
                                 *my_row->get_nth(2)->value);
                break;
        }
        case rowDestroy:
        case rowNotInService: {
                CoffeePercolator::
                    remove_schedule(my_row->get_index().first());
                break;
        }
}

boolean CoffeeSchedule::ready_for_service(Vbx**, int)
{
        // check if number of active schedules won´t be passed over
        return (active_schedules <= MAX_SCHEDULES);
        // may also check here for overlapping schedules...
}
```

Figure 9: Example for a Coffee-Percolator Scheduling Table (Method Routines)

(< *tableoid* >.1). The created table object supports arbitrary index length > 1 within the table, so an index first sub-identifier must contain the length of the index.

- MibTable::MibTable(const Oidx& oid, unsigned int ilen)

  Create a empty MibTable object with object identifier oid and a fixed index length of ilen. The given oid refers to the object identifier of the corresponding SNMP table´s entry managed object (< *tableoid* >.1). The created table object supports only index lengths of ilen and the values of columnar object instances participating in the index are not set automatically.

- MibTable::MibTable(const Oidx& oid, unsigned int ilen, boolean auto)

  Create a empty MibTable object with object identifier oid and a fixed index length of ilen. The given oid refers to the object identifier of the corresponding SNMP table´s entry managed object (< *tableoid* >.1). The created table object supports only index lengths of ilen and the values of columnar object instances participating in the index are set automatically if auto is true.

5.5.1.2 Destructor

- MibTable::~MibTable()

  Destroy the MibTable object and all contained columnar objects and their instances.

5.5.1.3 Configuration

- void MibTable::add_col(MibLeaf* obj)

  Add the columnar object obj to the table. The obj´s object identifier must be of length one and should refer to the column number.

- void MibTable::add_col(snmpRowStatus* rs)

  Add the snmpRowStatus object rs to the table. A table can contain at most one such a row status object. The snmpRowStatus object makes it easy to control the creation and deletion of rows caused by SNMP SET requests. See section 5.8 for details.

5.5.1.4 Table Operations

- MibTableRow* MibTable::add_row(const Oidx& index)

  Add a row with index to the table and return a pointer to the corresponding MibTableRow object. The programmer is responsible for not adding rows with same index value.

- void MibTable::remove_row(const Oidx& index)

  Remove the row with index from the table.

- void MibTable::remove_obsolete_rows(OrderedList< $Oidx$ >& confirmed)

  Remove all rows from the table which index value is not contained in the list of confirmed index values.

- MibLeaf* MibTable::get(int col, int row)

  Return a reference to the MibLeaf object at the position (col, row) within the table.

- boolean MibTable::find(const Oidx& oid, int& col, int& row)

  Determine the position of the columnar object instance with object identifier oid. If found set col and row accordingly (starting from 0) and return true.

- boolean MibTable::find_next(const Oidx& oid, int& col, int& row)

  Determine the position of the next columnar object instance with an object identifier >= oid. If found set col and row accordingly (starting from 0) and return true.

- MibTableRow* MibTable::find_index(const Oidx& index)

  Return a pointer to the MibTableRow object containing the row with index index. If such a row does not exist return 0.

- Oidx MibTable::index(const Oidx& oid)

  Return the index portion of the object identifier oid which must be an object id within the table.

- Oidx MibTable::base(const Oidx& oid)

  Return the object identifier of the columnar object corresponding columnar object instance with object id oid which must be an object id within the table.

5.5.1.5 Request Processing

- boolean MibTable::ready_for_service(Vbx** pvbs, int sz)

  Check if a row with the values contained in the array referenced by pvbs and of size sz can be set active (in service). This member function of a table with snmpRowStatus object is called whenever someone tries to set the table´s row status to rowActive using a SNMP SET request. MibTable´s ready_for_service member function checks only if all required columns of a row are set. Overload this function if your MibTable object needs a more clever check.

- Oidx MibTable::get_next_avail_index() const

  If the table has a fixed index length of one return the next free index value, that can be used to create an new row. This can be used to support (following the SMIv2 row status textual convention) managed objects that give a manager a hint for the next index value which can be used for a new row in a particular SNMP table.

5.5.1.6 Overloaded Member Functions

- void MibTable::get_request(Request* req, int ind)

  Propagate the GET operation requested by the sub-request of req with index ind to the appropriate MibLeaf object within the table.

- void MibTable::get_next_request(Request* req, int ind)

  Propagate the GETNEXT operation requested by the sub-request of req with index ind to the appropriate MibLeaf object within the table.

- int MibTable::prepare_set_request(Request* req, int ind)

  Propagate the SET operation requested by the sub-request of req with index ind to the appropriate MibLeaf object within the table. Return SNMP_ERROR_SUCCESS if the SET request could be performed, otherwise return the appropriate SNMP++ error code.

- int MibTable::commit_set_request(Request* req, int ind)

  Propagate the commit of the SET operation requested by the sub-request of req with index ind to the appropriate MibLeaf object within the table.

- int MibTable::undo_set_request(Request* req, int ind)

  Propagate the undo of the SET operation requested by the sub-request of req with index ind to the appropriate MibLeaf object within the table.

- void MibTable::cleanup_set_request(Request* req, int ind)

  Propagate the clean-up of the SET operation requested by the sub-request of req with index ind to the appropriate MibLeaf object within the table.

- Oidx* MibTable::max_key()

  Determine the last columnar object instance currently contained in the table and return a reference to its object identifier.

- mib_type MibTable::type()

  Return TABLE as type of the MIB object.

- MibEntry* MibTable::clone()

  Return a pointer to a copy of the table object.

5.6 MibProxy Class

The MibProxy class provides an easy way to proxy a management information subtree. All SNMP requests addressed to a managed object within that subtree are forwarded to a given SNMP agent. The MibProxy class asynchronously processes the responses to the forwarded requests and answers the original requests. Figure 10 shows an example implementation of a simple proxy agent. This agent holds the snmp group of MIB II locally and the system, interfaces and extended interfaces group are imported from a remote agent. The remote MIB groups are imported from the same source agent, but this is not a must.

5.6.1 MibProxy Class Member Functions

5.6.1.1 Constructors

- MibProxy::MibProxy(const MibProxy& other)

  Create a MibProxy object from another (copy constructor).

- MibProxy::MibProxy(const Oidx& root, mib_access max_access, const UdpAddress& source)

  Create a MibProxy object for a management information subtree with the given root. The maximum access rights for all managed objects within the subtree are specified by max_access. These rights may be more strict than those of the remote managed objects, but less strict access rights have no effect. Thus the minimum of both access rights will be applied. All managed objects belonging to the given subtree are retrieved from a SNMP agent with the management interface address source.

5.6.1.2 Destructor

- MibProxy::~MibProxy()

  Delete the MibProxy object.

5.6.1.3 Configuration

- void MibProxy::set_community(access_types access, const OctetStr& community)

  Set the community for the access type (READING or WRITING) specified by access to community.

```
...
main (int argc, char* argv[])
{
        u_short     port = 161;
        UdpAddress  source("127.0.0.1");
        source.set_port(161);               // set default SNMP port

        if (argc>1) port = atoi(argv[1]);   // set listen port
        if (argc>2) source = argv[2];       // set source agent IP address
        if (argc>3) source.set_port(atoi(argv[3])); // set source agent UDP
                                                    // port
        int status;
        Snmpx snmp(status, port);
        if (status == SNMP_CLASS_SUCCESS) {
                LOG_BEGIN(EVENT_LOG | 1);
                LOG("main: SNMP listen port");
                LOG(port);
                LOG_END;
        }
        else {
                LOG_BEGIN(ERROR_LOG | 0);
                LOG("main: SNMP port init failed");
                LOG(status);
                LOG_END; // program exits on fatal error log 0!
        }
        RequestList::set_snmp(&snmp);

        Mib mib;
        mib.add(new snmpGroup());   // add local snmp group
        // add remote system, interfaces, and extended interfaces group
        mib.add(new MibProxy("1.3.6.1.2.1.1",  READCREATE, source));
        mib.add(new MibProxy("1.3.6.1.2.1.2",  READCREATE, source));
        mib.add(new MibProxy("1.3.6.1.2.1.31", READCREATE, source));
        for (;;) {
                Request* req = RequestList::receive(120);
                if (req) mib.process_request(req);
        }
}
```

Figure 10: Example of a proxy agent implementation

5.6.1.4 Overloaded Member Functions

- mib_type MibProxy::type()

  Return PROXY as type of the MIB object.

- MibEntry* MibProxy::clone()

  Return a pointer to a copy of the MIB proxy object.

- Oidx MibProxy::max_key()

  Return the maximum object id value that may be within the management information subtree managed by MibProxy. Because this object id value cannot be determined exactly $root.2^{32} - 1$ is returned as an approximate value.

- void MibProxy::get_request(Request* req, int ind)

  Forward the by req and ind given GET sub-request to the source SNMP agent and store its response in req.

- void MibEntry::get_next_request(Request* req, int ind)

  Forward the by req and ind given GETNEXT sub-request to the source SNMP agent and store its response in req.

- int MibLeaf::prepare_set_request(Request* req, int ind)

  If the maximum access right of MibProxy is at least READWRITE return SNMP_ERROR_SUCCESS, otherwise return SNMP_ERROR_NO_ACCESS.

- int MibLeaf::commit_set_request(Request* req, int ind)

  Forward the by req and ind given SET sub-request to the source SNMP agent and store its response in req.

- int MibLeaf::undo_set_request(Request* req, int ind)

  Nothing is done. Returns SNMP_ERROR_SUCCESS.

- void MibLeaf::undo_set_request(Request* req, int ind)

  Nothing is done.

5.7 MibGroup Class

The MibGroup class is an encapsulation for a collection of MibEntry objects. MibGroup can be used to group a collection of MIB objects logically. If such a MibGroup object is added to a Mib instance it is flattened, which means each MibEntry object within that group object will be added to the Mib instance and then the group object itself will be deleted. As MibGroup is derived from MibEntry a MibGroup object can contain other MibGroup objects.

5.7.1 MibGroup Class Member Functions

5.7.1.1 Constructor

- MibGroup::MibGroup(const Oidx& group_id)

  Create a MibGroup object with group_id as the group´s object identifier. Any object added to this group must have an object id belonging to the subtree specified by group_id.

5.7.1.2 Destructor

- MibGroup::~MibGroup()

  Destroy the MibGroup object and all its contained MibEntry objects.

5.7.1.3 Configuration

- MibEntry* MibGroup::add(MibEntry* object)

  Add object to MibGroup. If the object´s object id is not in the group´s subtree the object will not be added. In any case a pointer to object is returned.

5.7.1.4 Overloaded Member Functions

- mib_type MibGroup::type()

  Return GROUP as type of the MIB object.

5.8 snmpRowStatus Class

The snmpRowStatus class is derived from MibLeaf and provides functionality to control the manipulation of MibTable rows. The snmpRowStatus class is an encapsulation of the SMIv2 row status textual convention. Figure 11 demonstrates the states which the snmpRowStatus object can traverse.

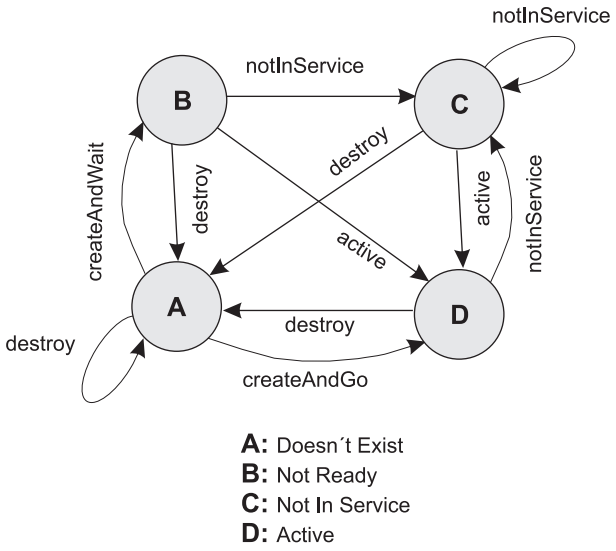

Figure 11: State Diagramm of RowStatus interaction

5.8.1 snmpRowStatus Class Member Functions

5.8.1.1 Constructor

- snmpRowStatus::snmpRowStatus(const Oidx& column, mib_access max_access)

  Create a snmpRowStatus object with object id column, which will be in most cases the number of the last column of a SNMP table. Usually the maximum access right is READCREATE, but there may be situations where the user should not be able to create new rows. In these cases max_access may be set to READWRITE or READONLY. If a READCREATE snmpRowStatus object is used in a table all the other columnar objects should have an maximum access right of at most READWRITE to assure that the row status mechanism is used to create new rows.

5.8.1.2 Destructor

- snmpRowStatus::~snmpRowStatus()

  Destroy the snmpRowStatus object.

5.8.1.3 Request Processing

- boolean snmpRowStatus::check_state_change(const Vbx& new_status)

  Return true if the row can be set to the row status given by new_status. If the current status is rowNotReady or rowNotInService and the desired status is rowActive

snmpRowStatus calls the ready_for_service method of the table with the values of the row´s MibLeaf objects to check if this operation is permitted.

5.8.1.4 Miscellaneous

- long snmpRowStatus::get()

  Return the status of the row to which snmpRowStatus belongs. This can be rowNotReady, rowNotInService, or rowActive.

5.8.1.5 Overloaded Member Functions

- MibEntry* snmpRowStatus::clone()

  Return a pointer to a copy of the snmpRowStatus object. Always refine the clone method in derived classes in order to make sure the derived class is called within a table rather than its base MibLeaf.

- boolean snmpRowStatus::value_ok(const Vbx& vb)

  Return true if the row status can be set to vb´s value. The snmpRowStatus::value_ok member function checks only if the desired way to change its state is valid.

References

[Mellquist96] Peter E. Mellquist. *SNMP++, An Open Specification for Object Oriented Network Management Development Using C++*. Hewlett Packard Company, 1996.

[Perkins97] David Perkins and Evan McGinnis. *Understanding SNMP MIBs*. Prentice Hall PTR, 1997.