The AgentX Protocol Implementation For AGENT++

Version 1.4.8

Frank Fock

1 Contents

2 System Requirements

To use AgentX++ v1.4.2 you need SNMP++ v3.2 and AGENT++ v3.5.6 (or later) installed. AgentX++ can be compiled without changes on Linux, Solaris 7, and Windows NT (Visual C++ 6.0)[1]. As AgentX++ is available as source code license, it may be ported to other operating systems as well, provided that an ANSI C++ compiler is available.

If context support is a requirement then SNMP++ v3.1 is needed and the SNMP++ library as well as the AGENT++ library has to be compiled with the _SNMPv3 compilation flag set[2]. The following table gives an overview of the requirements.

AgentX++ Feature	Supported OS	Required Libraries
Non-default contexts	Linux, Solaris 7, Windows NT	SNMP++v3.2, AGENT++v3.5.6 (or later)
Default context only	Linux, Solaris 7, Windows NT	SNMP++v3.2*, AGENT++v3.5.6 (or later)
TCP socket connections	Linux, Solaris 7, Windows NT	-
UNIX domain socket connections	Linux, Solaris 7	-

* _SNMPv3 can be undefined in *snmp++/include/config.h* if context and SNMPv3 support is not required.

3 Installation

Use any decompression tool capable of unpacking GNU zipped tape archive files (TAR), for example WinZIP or gtar to unpack the downloaded **agentX++.tar.gz** file. Unpack the file in the same directory where you installed SNMP++ and AGENT++. After installation, the installation directory should look like:

```
installdir/
        agent++                     AGENT++ installation
        agentX++                    Installed AgentX++ package
            examples
                master
                subagent
            include
            src
        snmp++                      SNMP++ installation
```

After installation the AgentX++ library can be compiled by changing the working directory to installdir/agentX++/src and executing

[1] Solaris and Windows NT are trademarks of SUN Microsystems and Microsoft respectively.

[2] The _SNMPv3 flag has to be set consistently for SNMP++ in snmp++/include/config.h and for AGENT++ in agent++/include/agent++.h.

make –f Makefile.<platform>

4 What Is AgentX++?

AgentX++ is a C++ API that adds support of the AgentX protocol to the AGENT++ API. can be used in conjunction with the AGENT++ and SNMP++ API to create SNMP agents supporting the AgentX protocol. The Agent Extensibility (AgentX) Protocol[3] defined by the Internet Society is a standardized protocol for the communication between SNMP master and subagents. The motivation for AgentX is given by RFC 2741 as follows:

> New MIB modules that extend the Internet-standard MIB are continuously being defined by various IETF working groups. It is also common for enterprises or individuals to create or extend enterprise-specific or experimental MIBs.
>
> As a result, managed devices are frequently complex collections of manageable components that have been independently installed on a managed node. Each component provides instrumentation for the managed objects defined in the MIB module(s) it implements.
>
> The SNMP framework does not describe how the set of managed objects supported by a particular agent may be changed dynamically.
>
> This very real need to dynamically extend the management objects within a node has given rise to a variety of "extensible agents", which typically comprise
>
> - a "master" agent that is available on the standard transport address and that accepts SNMP protocol messages
> - a set of "subagents" that each contain management instrumentation
> - a protocol that operates between the master agent and subagents, permitting subagents to "connect" to the master agent, and the master agent to multiplex received SNMP protocol messages amongst the subagents.
> - a set of tools to aid subagent development, and a runtime (API) environment that hides much of the protocol operation between a subagent and the master agent.
>
> The wide deployment of extensible SNMP agents, coupled with the lack of Internet standards in this area, makes it difficult to field SNMP-manageable applications. A vendor may have to support several different subagent environments (APIs) in order to support different target platforms.
>
> It can also become quite cumbersome to configure subagents and (possibly multiple) master agents on a particular managed node.
>
> Specifying a standard protocol for agent extensibility (AgentX) provides the technical foundation required to solve both of these problems. Independently developed AgentX-capable master agents and subagents will be able to interoperate at the protocol level. Vendors can continue to differentiate their products in all other respects.

[3] The AgentX Protocol Version 1 is defined by RFC 2741 and updates thereof.

Readers that are not familiar with the AgentX specification are encouraged to read the corresponding RFCs before proceeding. Knowledge of the AgentX protocol is required in order to understand and use all features of AgentX++.

AgentX++ extends AGENT++ without changing the interface between AGENT++ and the management instrumentation. Thus, an existing AGENT++ agent can be migrated to an AgentX master or subagent within minutes by only changing a few lines of code within the agents main routine. The system architecture of an AgentX++ master or subagent is shown by Figure 1.

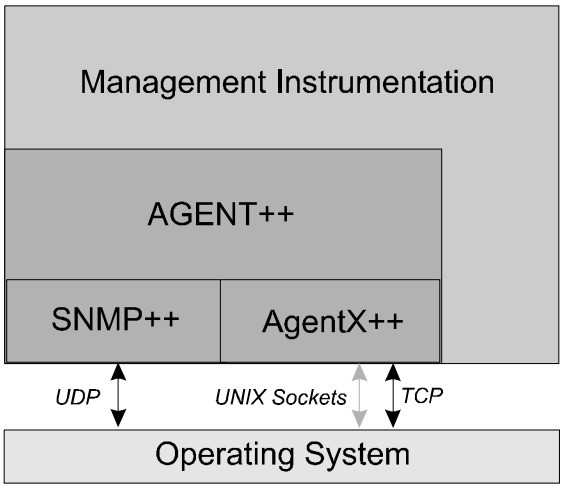

Figure 1: AgentX++ System Architecture

The interface between AGENT++ and the management instrumentation is defined through the AGENT++ classes *MibLeaf* and *MibTable*. These classes are used unchanged with AgentX++. What changes is the *Mib* class. It is subclassed by the *MasterAgentXMib* class and the *SubAgentXMib* class like it is shown by Figure 1.

The AgentX++ API user decides by choosing one of these classes, whether she implements an AgentX master or an AgentX subagent. The transport layer dependent functions, represented by the classes *AgentXMaster* and *AgentXSlave,* are separated by aggregation from the classes *MasterAgentXMib* and *SubAgentXMib* respectively. This facilitates porting AgentX++ to new operating systems or adding new transport protocols.

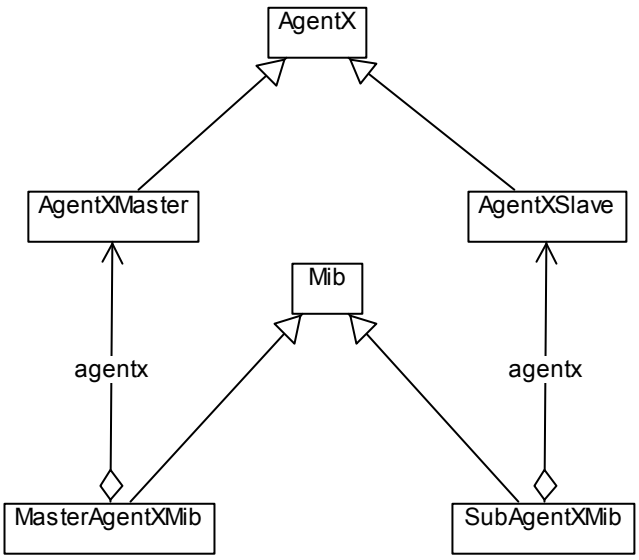

Figure 2: UML Class View of AgentX++ Mib class extensions

5 How AgentX++ Works

This chapter explains how the AgentX protocol extension for AGENT++ is implemented and why this implementation meets the requirements of the AgentX protocol specification. Readers who are familiar with the AgentX protocol and interested in a quick start may skip this chapter and proceed with chapter 6.

5.1 The Master Agent

The AgentX specification requires for an AgentX master agent, that it is capable of performing the following functions:

- Establishment of AgentX sessions with subagents.

- Registration of MIB regions by subagents. A MIB region is given by an object identifier that names a subtree[4]. Regions registered by different subagents may overlap. Regions may be registered with different priorities. When dispatching a SNMP request, the master agent determines the authoritative region by

    1. choosing the region that was originally registered with the most specific region,
    2. if there is still more than one applicable region, then the one with the lower priority value is chosen.

[4] There are other special regions that do not name a single subtree. Instead those regions specify rows in tables. See the AgentX Protocol specification for details.

- Application of MIB views, access control policy for managed nodes, and implementation of any MIB objects relevant to any administrative framework the agent supports, particularly the MIB objects defined in RFC 1907.

- Sending and receiving AgentX protocol messages to access management information, based on the current registry of MIB regions.

- Forwarding notifications on behalf of subagents.

The AGENT++ framework provides already most of the above functions. What is missing is the ability to register overlapping regions and dispatching SNMP requests to AgentX subrequests. The following sections describe how

- sessions are managed by an AgentX++ master agent
- region registration is implemented
- requests are dispatched
- notifications are forwarded
- index allocation is implemented.

5.1.1 AgentX Session Management

An AgentX++ master agent has dedicated threads for processing AgentX messages. Thus, multi-threading support must be available and activated[5] with an AgentX++ master agent.

When a subagent requests an UNIX domain socket or TCP connection, the master usually accepts it and establishes the connection. On success, the master adds an *AgentXPeer* class instance to the internal list of connected peers. It also adds the appropriate entries into the *agentxConnectionTable* of the AgentX MIB.

The next request on the newly established connection has to be an AgentX OPEN PDU. Whenever such a PDU is received an *AgentXSession* instance is created and added to the list of open sessions. In addition, an appropriate row in the *agentxSessionTable* is created. Each *AgentXSession* instance has a reference to the *AgentXPeer* describing the connection over which the OPEN PDU has been received. Figure 3 shows how the AgentX++ classes involved in session management are related to each other.

[5] Multi-threading is activated by defining the _THREADS macro in AGENT++'s *agent++.h* file. It is activated by default.

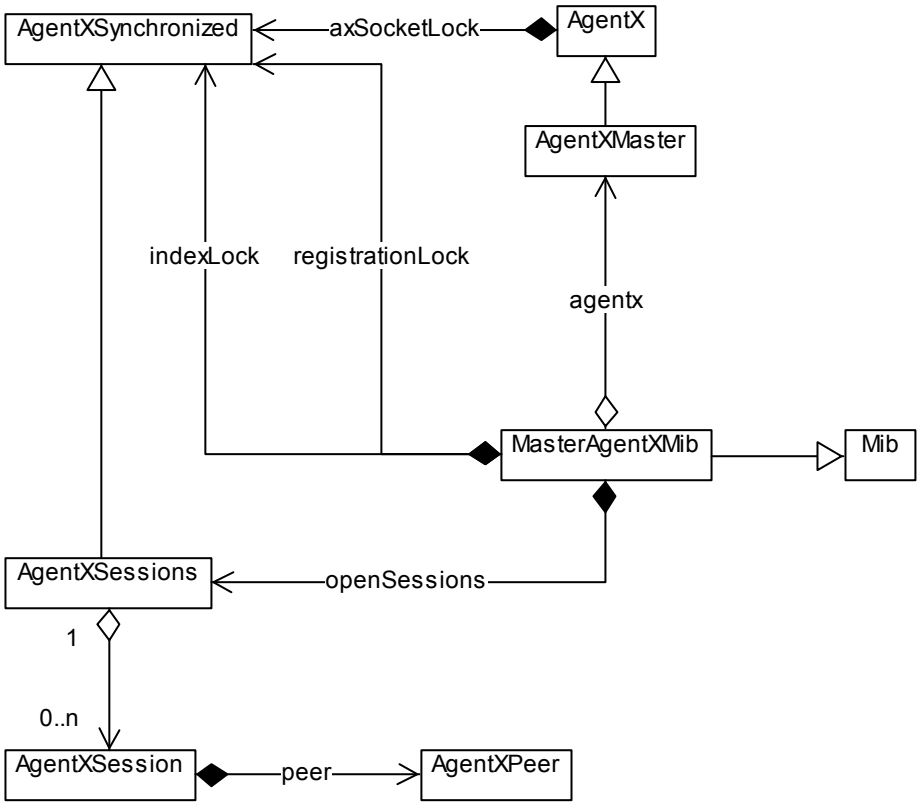

Figure 3: UML Class View of master agent related AgentX++ classes

When a session is closed by a subagent, the corresponding *AgentXSession* object is removed from the list of opened sessions. All the registrations of index values and regions allocated on behalf of this session are removed, as well as all corresponding entries within the AgentX MIB.

When a connection to a subagent is dropped, then all sessions opened on behalf of this connection are closed.

When the master agent is stopped it closes all open sessions and thus sends a session CLOSE PDU to all sessions.

5.1.2 AgentX++ Region Registration

With AGENT++ is capable of registering regions that have a lower and an upper bound. Such a region registration is done by adding a *MibEntry* instance to the agent's *Mib* instance. A *Mib* instance has an AVL tree for each supported context which holds registered MIB objects (regions). Each MIB object has a key, which is normally its object identifier, and a maximum key. The maximum key is returned by the *MibEntry::max_key()* method and returns the non-including upper bound of the registered region.

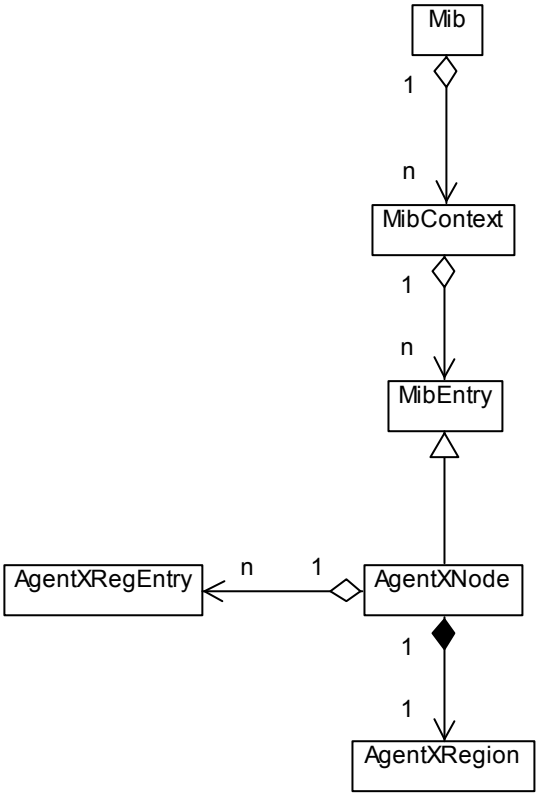

Figure 4: UML Class View of the AgentXNode and related classes.

AgentX++ uses the ***AgentXNode*** class to register its AgentX regions. *AgentXNode* is a subclass of *MibEntry* and thus has a lower and a non-including upper bound. Therefore, an *AgentXNode* instance can be used to register an **atomic** region.

How the *AgentXNode* class is related to other classes of AgentX++ and AGENT++ shows Figure 4. Each *AgentXNode* contains exactly one *AgentXRegion*. It holds to lower and the upper bound of the region represented by that node. Each *AgentXNode* has one or more registration entries represented by *AgentXRegEntry* instances. These entries are sorted by length of their registration object identifier in descending order and their priority value in ascending order. When a SNMP request is dispatched to an *AgentXNode* it uses the first entry of the registration list. This entry points to the session and thus to the connection, that should be used to forward the request using AgentX. Figure 5 shows an *AgentXNode* holding registration information about the region from ifIndex.2 (1.3.6.1.2.1.2.2.1.1.2) to ifIndex.3 (1.3.6.1.2.1.2.2.1.1.3).

The question raised by the example is, how overlapping regions could be registered when an *AgentXNode* may only be used for atomic regions? The solution for this dilemma is splitting existing registered regions (*AgentXNodes*), when they overlap with the newly registered one. Figure 6 shows the consecutive processing of four regions and the resulting *AgentXNode* registration.

Each time a new registration is requested and it affects existing regions, it is processed as follows:

- If the **new region equals an existing one**, then the new registration entry will be added to the registration list of the existing one, if the new registration has a different

priority to all existing entries in that list. If it has the same priority as another entry, then the registration will be denied because of a duplicate registration.

- If the **new region covers the existing ones**, then the existing regions will be subtracted from the new one. The resulting regions will be added to current MIB context by creating corresponding AgentXNode instances, which will contain a single registration entry. This original registration entry will also be added to the registration lists of the affected existing *AgentXNodes*.

- If the **new region is covered by an existing region**, then the existing one will be split into two or three parts. There will be three parts if both regions do not share a boundary. The new registration entry will be then added to the split parts.

- If the **new region overlaps with any non-AgentX region** (i.e., *MibLeaf*, *MibTable*, etc.), then the registration request will be denied.

If **no region is affected by the new one**, then a new *AgentXNode* is created with representing the new registration. Particularly, this is the case, when a subagent registers a region in a new context. A *MasterAgentXMib* instance may be configured to allow a subagent to implicitly create contexts in the master agent by simply registering a region for a new context.

Is **auto context creation** enabled (by calling *MasterAgentXMib::set_auto_context)*, the master agent will also add new contexts to the View Access Control Model (VACM) MIB. By default, auto context creation is disabled and a subagent may not register any regions for contexts that are not known by the *MasterAgentXMib* instance.

In either case, appropriate configuration entries in the VACM MIB have to be added to map a user/context combination to an access group. This configuration may be done via SNMP or by the master agent. Enable auto context creation and overwrite the *MasterAgentXMib::add_new_context* method in the latter case.

When existing regions are split, the lower bounds of the corresponding *AgentXNodes* are never changed, because AGENT++ uses an AVL tree to store its MIB objects and the lower bound of an *AgentXNode* is at the same time the key used by that AVL tree. Would this key be changed while it is registered, the MIB's lookup routines will stop working correctly.

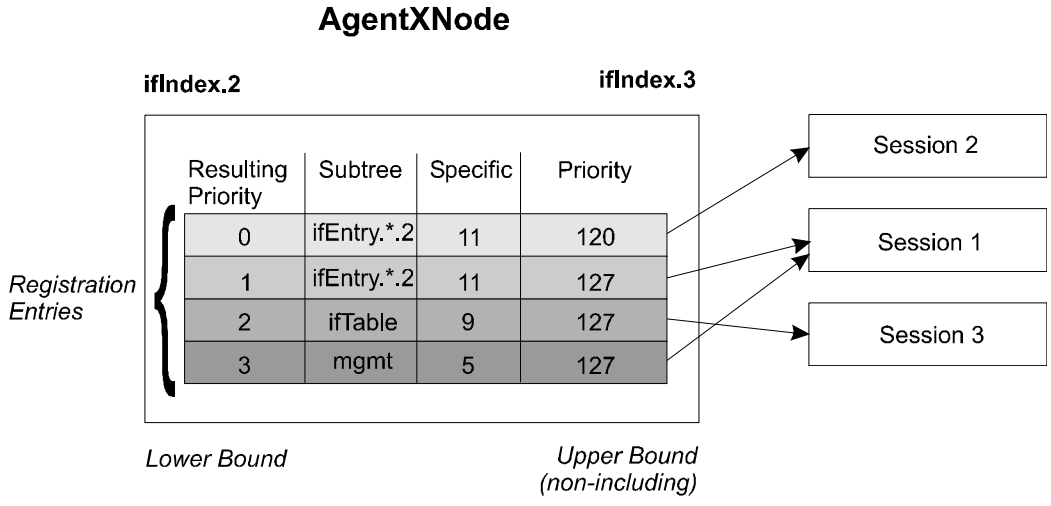

Figure 5: Example AgentXNode

The **unregistration process** is realized in a similar way to the registration process. When a unregistration takes place, all *AgentXNodes* are searched for the registration entry referenced by the unregister request. If an *AgentXNode* contains that entry, it will be removed from that node. If the node then does not have any registration entry, the node will be removed from the MIB too. If the node has exactly one registration entry and the predecessor AgentXNode also has exactly the same entry and both regions border on each other, then the second node will be removed from the corresponding MIB context and the predecessor's region will be expanded in order to include the union of both regions.

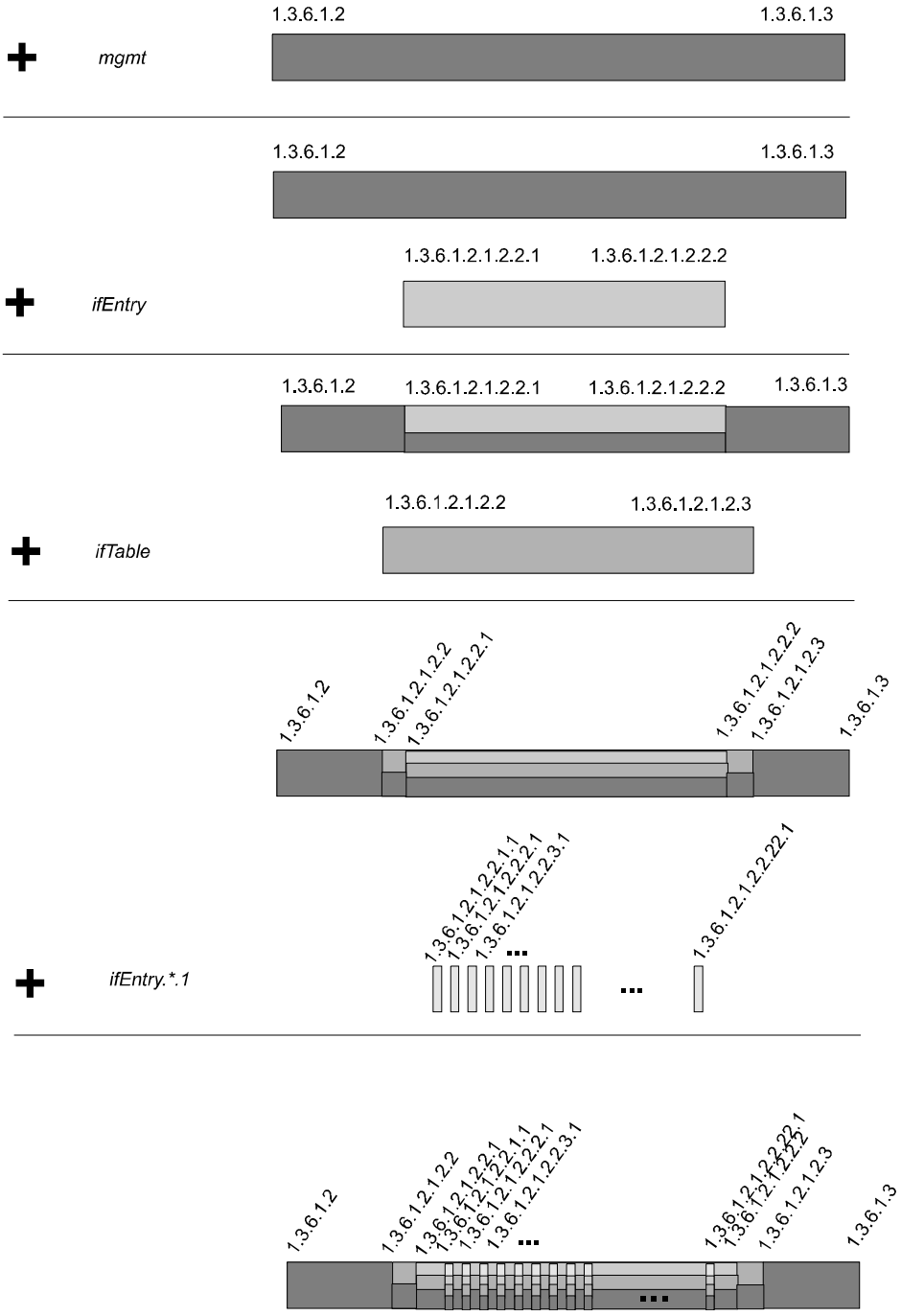

Figure 6: Example of a registration process

5.1.3 AgentX Message Dispatching

AGENT++ dispatches SNMP messages to method routines using a conceptual dispatch table. It is called conceptual, because it is not a table, but an AVL tree of AVL trees. The first one is indexed by SNMPv3 contexts[6], the second one is indexed by management information identifiers (object identifiers) and contains *MibEntry* instances. A *MibEntry* instance represents a managed object. The *MibEntry* class is abstract and thus it does only define an interface to the method routines of a managed object.

When AGENT++ processes a SNMP request, it selects the AVL tree to be used to dispatch the request by the request's context information. AGENT++ then searches for each subrequest the authoritative managed object. If such an object can be found its appropriate method routine will be called depending on the type of the SNMP request.

Similarly, AGENT++ dispatches SNMP subrequests where an *AgentXNode* is the authoritative managed object. AGENT++ calls the appropriate method routine of the AgentXNode. It then propagates the subrequest to the AgentX subagent, which is owner of the authoritative registration entry within that node. But this is not done subrequest by subrequest like the SNMP proxy managed objects *MibProxy* or *MibProxyV3* of AGENT++ do it. In fact, **AgentX++ ensures that for each incoming SNMP request it sends at most one AgentX request to each affected AgentX session[7]**, which gains performance and which enables the master agent to **perform SET requests atomically**.

If AgentX++ had handled each subrequest individually, it would have been impossible for an AgentX subagent to determine whether two or more of those subrequests are related. Only if the subagent knows that two SET subrequests are related (are in the same PDU), it will be able to determine whether those subrequests are mutually exclusive.

So, how does it actually work? When an *AgentXNode*'s method routine (e.g., *AgentXNode::prepare_set_request*) is called it computes all information needed to create an AgentX request addressed to the authoritative session of that node. It then calls the *add_get_subrequest* or *add_set_subrequest* method of the *MasterAgentXMib* instance, respectively. These methods actually create and queue AgentX requests. If there is not already an AgentX request queued (but not sent) for the current transaction and target AgentX session, then a new request will be created and queued, otherwise the subrequest will be appended to the existing one. Since the index of a subrequest within the resulting AgentX requests do not necessarily correspond to the index of the original subrequest, a reference is stored within the queued AgentX request, which enable the master agent to map AgentX responses back to the original subrequests.

Queued AgentX request are sent when the virtual *MasterAgentXMib::finalize(Request*)* method is being called. If there are any queued AgentX requests for the SNMP request being finalized, the request will not be answered, instead the queued AgentX request will be sent. Hence, all subrequest that can be processed locally will be processed before any AgentX request is sent. This avoids unnecessary communication if an error occurs while processing the subrequests with local scope.

---

[6] For SNMPv1 and SNMPv2c there is only a single context, which is called the default context.

[7] This statement does not apply to GETNEXT or GETBULK request, because of the search facilites of these requests.

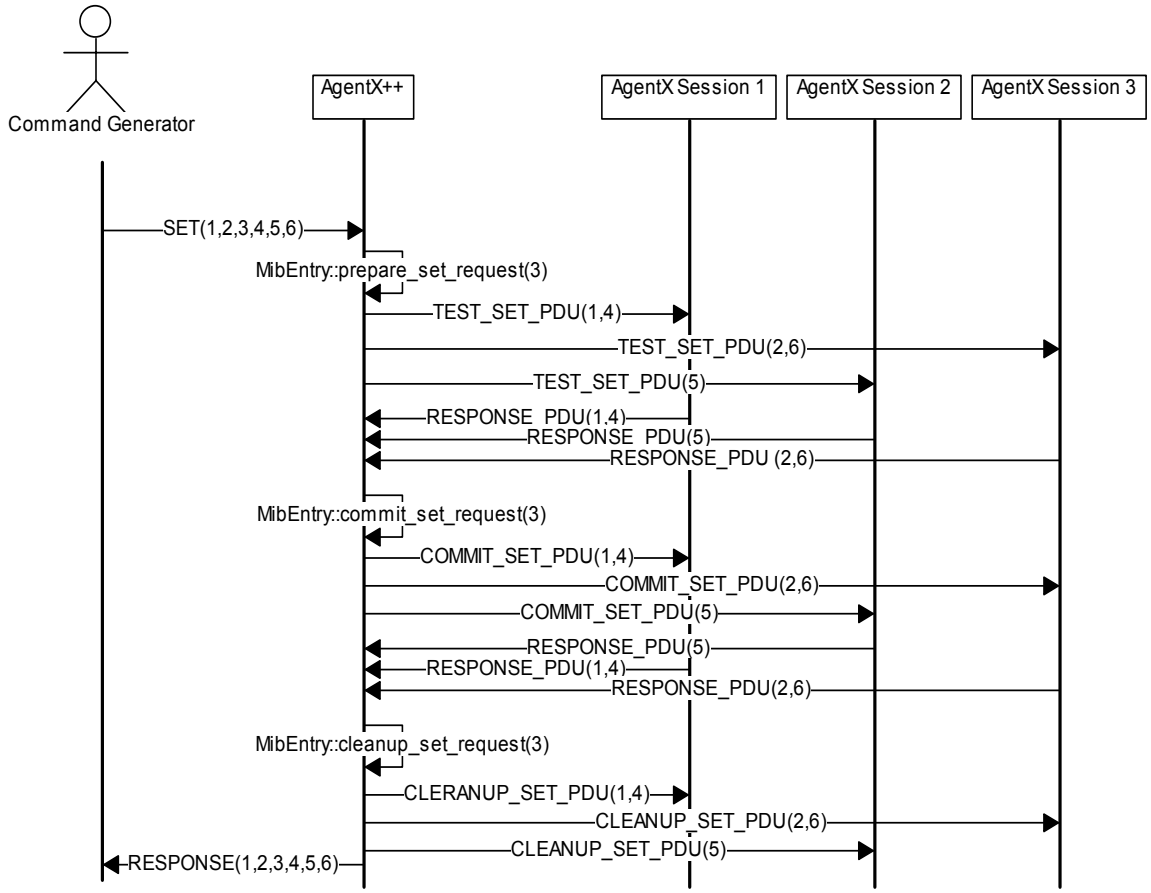

Figure 7: Sequence Diagram of a successful SET-request (values in brackets represent SNMP subrequest indexes)

Figure 7, for example, shows how AgentX++ dispatches a SNMP SET request with six subrequests, which were all processed successfully. In the shown example, the master agent sends nine AgentX request and receives six responses. A master agent that does not group subrequests would have sent fifteen AgentX request and would have received ten responses. That is a communication increase of 67 percent.

Dispatching search requests like GETNEXT or GETBULK is more complicated than dispatching the above discussed GET and SET requests. Dispatching a search request may involve trying several regions before an object can be found that matches the search criteria. Thus, when a search on a particular region fails, the subrequest is reprocessed with the next registered region or MIB object. The Reprocessing is realized by calling the Mib::do_process_request() method again with the partly processed request and the search OID of the failed subrequest replaced by the OID of the upper bound of the empty region.

AgentX++ dispatches BULK requests as described in section 7.2.1.3 of RFC 2741 (italic text indicates AgentX++ specific behavior):

Each variable binding in the request PDU is processed as follows:

1. Identify the authoritative target region and target session, exactly as described for the agentx-GetNext-PDU (see section 7.2.1.2, "agentx-GetNext-PDU").

2. If this is the first variable binding to be dispatched over the target session in a request-response exchange entailed in the processing of this management request:

   - Create an agentx-GetBulk-PDU for the session, with the header fields initialized as described above (see section 6.1, "AgentX PDU Header").

3. Add a SearchRange to the end of the target session's agentx-GetBulk-PDU for this variable binding, as described for the agentx-GetNext-PDU. If the variable binding was a non-repeater in the original request PDU, it must be a non-repeater in the agentx-GetBulk-PDU.

The value of g.max_repetitions in the agentx-GetBulk-PDU may be less than (but not greater than) the value in the original request PDU.

The AgentX++ master agent makes such alterations due to simple sanity checking, optimizations for the current iteration based on the registry, and the maximum possible size of a potential Response-PDU.

5.1.4 Response Processing

Since version 1.4, AgentX++ uses thread pools instead detached threads. Using thread pools improves performance because threads are only created once and the number of threads concurrently running is limited. In addition, an agent can be shut down more cleanly when no detached threads are used.

AgentX++ uses the following four thread pools:

- A thread pool with the default size of four threads that is used to process incoming SNMP requests. This is the same thread pool as present in AGENT++.
- A thread pool with a single thread (default) that processes AgentX requests, not including AgentX responses received from subagents.
- A thread pool with a single thread (default) that processes AgentX responses on GET, GETNEXT, and GETBULK requests.
- A thread pool with a single thread that processes AgentX responses on SET related requests.

SET responses are handled in a different thread because SET requests have to be processed atomically by a SNMP agent. Thus, AgentX++ locks all objects (resources) required to answer a SET request before processing begins and releases these locks not before processing of the SET request has finished. Unfortunately, SET requests are not processed in a single step with AgentX. For this reason it is necessary to process SET related responses in a different thread than for example GETBULK responses. Otherwise, the processing of a GETBULK request could hit an object locked by a pending SET request. This would block the GETBULK resulting in a dead lock because the next response related to the pending SET request cannot be processed until the GETBULK leaves the response thread. One could argue now, that it would be sufficient to increase the size of the response thread pool. But it would not solve the problem on principle.

When a request needs reprocessing, for example a GETBULK or SET request, then sending the AgentX subsequent requests must be done from the thread pool dedicated to AgentX request processing.

5.1.5 Notification Forwarding

AGENT++ includes a class named *NotificationOriginator*, which provides services to generate notifications (traps). This class uses the MIB objects defined in the SNMP-NOTIFICATION-MIB, SNMP-TARGET-MIB and SNMP-COMMUNITY-MIB to determine target destinations and their parameters as well as to control notification filtering.[8]

AgentX++ forwards each received AgentX notification by calling the *notify* method it inherits from the *Mib* class. By default, the *Mib::notify* method uses a *NotificationOriginator* instance. Hence, AgentX++ uses the same MIBs to control the generation of notifications as AGENT++ does.

5.1.6 Index Allocation

The index allocation service is provided by an AgentX++ master agent to support sharing conceptual tables among subagents. The master maintains a database of index objects (OIDs), and, for each index, the values that have been allocated for it. That database is implemented by the *AgentXIndexDB* class. The relation between an index object and its allocated values, represented by the *AgentXIndexEntry* class, is represented by the *AgentXIndex* class. Figure 8 shows how these classes are related with the *MasterAgentXMib*. An index database is automatically created for a context when the first index for that context is being allocated. A subagent may request allocation of

1. a specific index value (allocate index)
2. an index value that is not currently be allocated (any index)
3. an index value that has never been allocated (new index)

In the first case, the subagent chooses an index value, in the second and third case the master agent creates an appropriate value and returns it to the subagent. AgentX++ supports the allocation of a specified index for any type of valid index objects, which are:

- Integer32
- Timeticks
- Gauge32
- OCTET STRING
- BITS
- Opaque
- IpAddress
- OBJECT IDENTIFIER

AgentX++ is able to create new index values for Integer32 and Gauge32 (which is undistinguishable from UInteger32) index objects only. This not a crucial restriction, because index allocation is necessary only when the index in question is an arbitrary value, and the subagent has no other reasonable way to determine which index value to use.

[8] The SNMP-NOTIFICATION-MIB and the SNMP-TARGET-MIB are defined in RFC 3413. The SNMP-COMMUNITY-MIB is specified in RFC 2576.

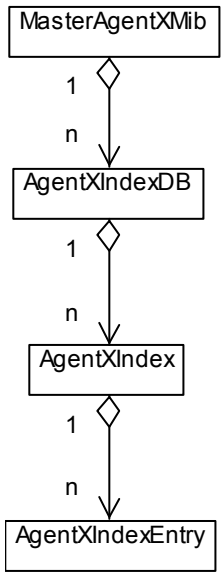

Figure 8: UML class view of index allocation related classes

An index allocation may allocate values for more than one index object at once. All three types of index allocations are executed atomically. That is, if the allocation or creation of an index object fails, then AgentX++ will not allocate any value for the other index objects of that request either.

5.2 The Subagent

The architecture of an AgentX++ subagent is similar to an AGENT++ agent. All SNMP transport specific functions are replaced by their AgentX counterparts. This is achieved by subclassing the AGENT++ classes *Mib*, *RequestList*, and *Request* as shown in Figure 9.

View Access Control is not needed by AgentX subagents. Hence, an AgentX++ subagent does not use the *Vacm* class for view access control. Instead it uses the *NoVacm* class, which always permits access.[9]

Notifications are sent by an AgentX subagent to its master agent rather than directly to any notification receivers. Thus, an AgentX++ subagent should not implement the SNMP-TARGET-MIB, SNMP-NOTIFICATION-MIB, and SNMP-COMMUNITY-MIB. Consequently, the *NotificationOriginator* class cannot be used for generating notifications. Instead, the *SubAgentXMib* class overrides the *notify* method of *Mib*. By calling the *SubAgentXMib::notify* method, a notification is propagated to the master agent, which then forwards the notification itself to the notification receivers determined by evaluating the SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB.

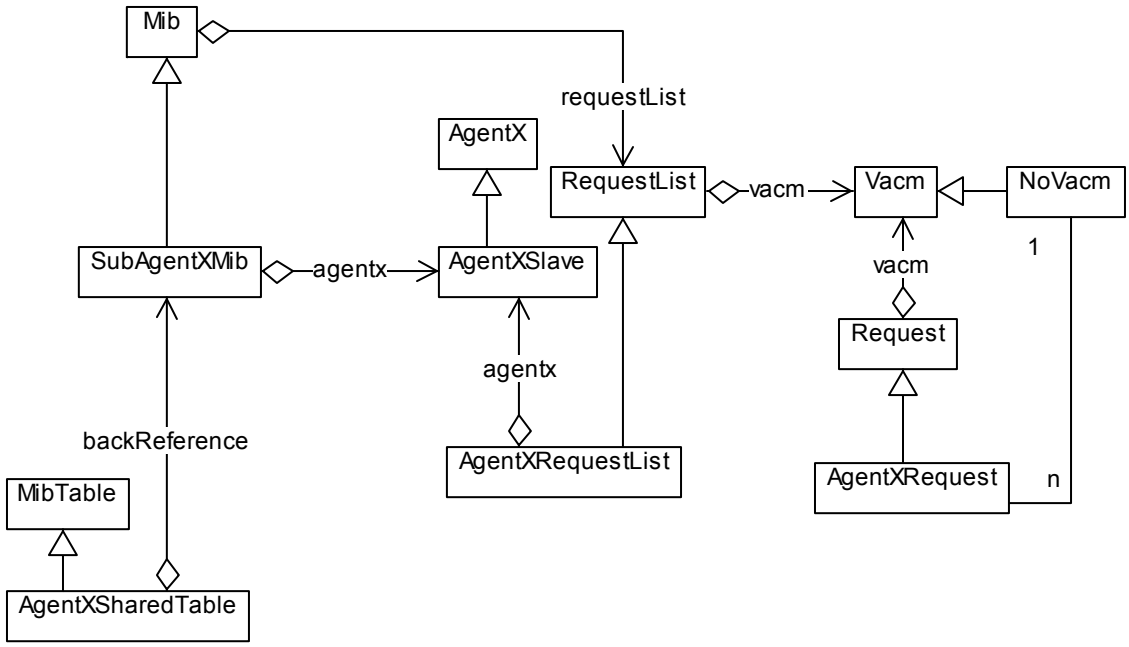

Figure 9: Simplified UML class model of subagent related AgentX++ classes

AgentX shared tables extend SNMP tables by an additional characteristic, which allows such tables to be shared between different subagents. AGENT++'s *MibTable* class has no means to support this. The *AgentXSharedTable* extends the *MibTable* class in order to provide these means for AgentX++ subagents.

[9] If SNMPv3 support is disabled (see section 1), then AgentX++ will not use the *NoVacm* class nor the *Vacm* class.

5.2.1 Region Registration

An AgentX subagent has to register its MIB objects at its master agent (see section 5.1.2), before the master agent will be able to dispatch any messages to the subagent. Crucial point for efficiency is the granularity of the registered regions. The less specific a region is registered, the more efficient it is accessed through the master agent. Nevertheless, it is not advisable to register a single region like the *mgmt* subtree (1.3.6.1.2), because it might then be difficult or impossible for other subagents to override or add other regions within such a region (subtree).

AgentX++ gives the API user full control over the registration granularity. With the *MibGroup* class from AGENT++ MIB objects can be logically grouped. In the first place, AGENT++ uses this class to logically group MIB objects, which should be made persistent at once.

Each *MibGroup* has an object identifier (key) that should denote the most specific subtree that covers all MIB objects within that group. AGENT++ checks that only MIB objects with an object identifier within that subtree may be added to a *MibGroup*.

AgentX++ registers its MIB objects during the initialization of the *SubAgentXMib* as follows:

- A *MibGroup* is registered as a single region with the master. The region registered is the subtree denoted by the group's key. All MIB objects added to that group are not registered explicitly.

- Each MIB object, that is not part of any *MibGroup,* is registered with its own region. *MibLeaf* objects are registered as single instance regions, whereas *MibTable* as well as *MibComplexEntry* objects are registered as subtree regions.

MIB objects added to the *SubAgentXMib* after its initialization are registered as follows:

- *MibGroup* instances or MIB objects other than *AgentXSharedTable* instances are registered as described above. *AgentXSharedTable* instances should be added through a special method that suppresses registration of the whole table. This should be done for better dispatching performance, but is optional.

- An *AgentXSharedTable* registers each row that is added by calling its *add_row* method by a range registration (see section 6.2.3 of RFC 2741). **Consequently, rows must not be added before the *SubAgentXMib* is initialized**, because the registration process needs an open AgentX session. It is recommended to use only consecutive sub-identifier values for the columns of a shared table. Since rows of a shared table are registered using range registration (see 5.1.2), gaps are processed as if those columns where actually present. Thus, performance of retrieving the table goes down straight proportional to the size of the gap. If large gaps are needed (for example because the MIB specification cannot be changed), then the method *AgentXSharedTable::add_row* should be overwritten to replace the range registration by an individual registration of the column instances.

The registered regions are unregistered whenever a MIB object or *MibGroup* is removed from the *SubAgentXMib.* A row of an *AgentXSharedTable* instance is removed when the *remove_row* method is called.

If a region registration is acknowledged by the master agent, the *registration_success* method of the *SubAgentXMib* will be called. Except for shared table row registrations, no further action is performed by this method. An API user who needs to trigger any actions on a successful registration will have to override this virtual function.

After a failed registration attempt, the *registration_failed* method will be called, which does not perform any actions by default, because a general reaction cannot be defined. An API user who wants to react on a failed registration should override this method.

Contexts need not to be registered explicitly. The master agent may reject the registration of a region for an unknown or unsupported context (see also 5.1.2).

5.2.2 Shared Tables

Conceptual tables that may be shared among AgentX subagents are represented by the *AgentXSharedTable* class. It extends AGENT++'s *MibTable* class by means for dynamically allocating and deallocating index values as well as registering and unregistering rows.

Instead using the *add_row* method directly to add a new row to a shared table, one of the three methods for index allocation should be used. They send an index allocation request to the master and return immediately. When the acknowledgement from the master is received, the row is added and a region registration request is sent to the master (see Figure 10). Only when the acknowledgement of the region registration is received the table's *row_added* method is called.

The *add_row* method uses range registration (see 5.1.2) to register a row with the master agent. This approach can be ineffective if the columns of a shared table are not consecutively numbered. In such a case, overwriting the *add_row* method to replace the range registration by individual registrations is recommended.

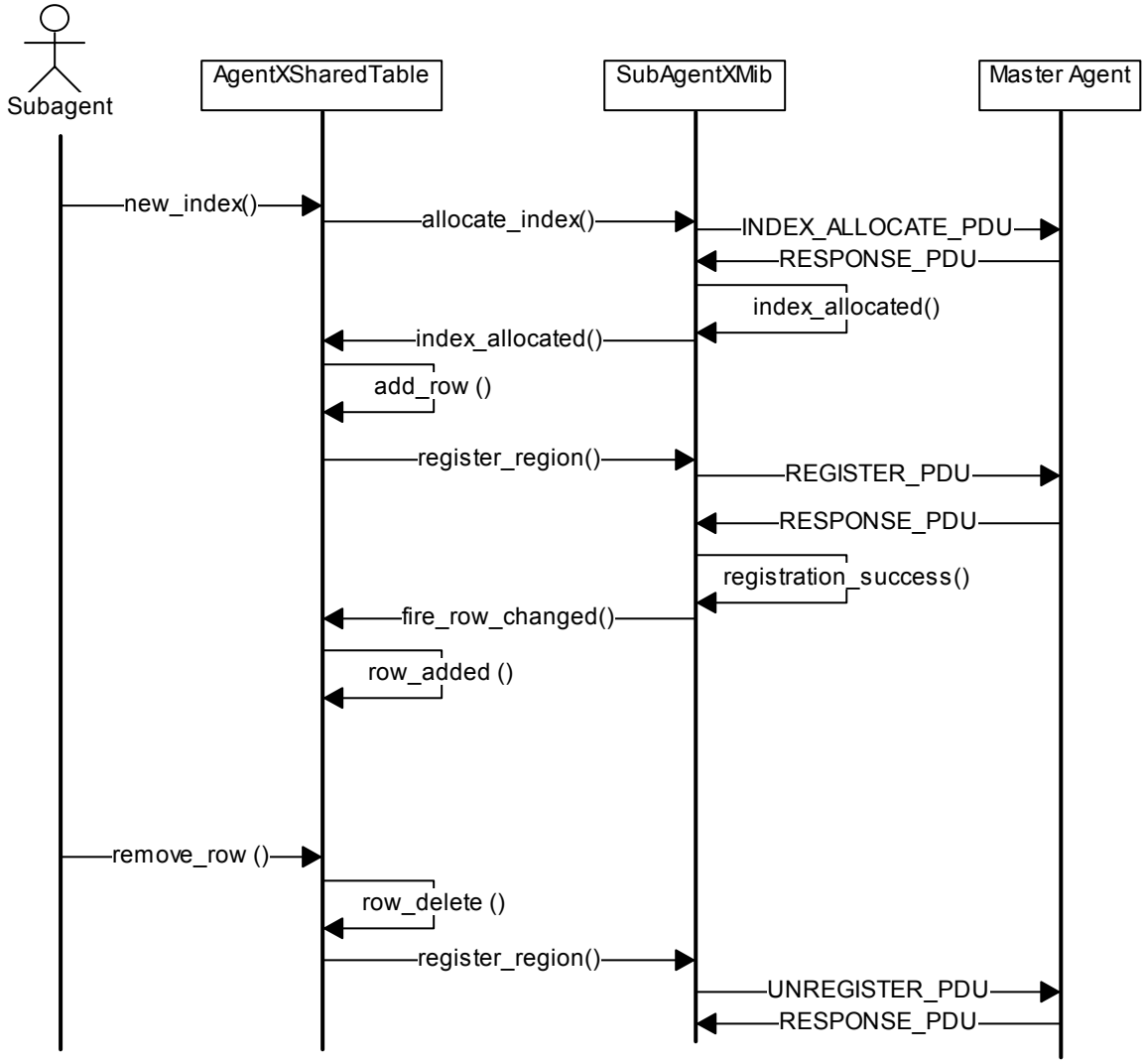

Figure 10: Successful row creation / deletion sequence diagram

In case of an error, i.e. if the index allocation failed, the table's *index_allocated* method will be called with the corresponding error code and error index. The error index denotes the failed index object. A failed region registration is ignored by default (see section 5.2.1).

6 Using AgentX++

One of the goals of AgentX++ is making the migration from an AGENT++ agent to an AgentX master or subagent as simple as possible. Consequently, the AgentX++ API hides most of the AgentX protocol specific characteristics. Thus, for those who are already familiar with AGENT++, using AgentX++ should be easy.

The sections 6.1 and 0 describe in detail how an AGENT++ agent is converted into an AgentX master or an AgentX subagent, respectively. The sections are structured according to the general structure of an AGENT++ agent, which is shown by Figure 11.

Figure 11: Structure of an AGENT++ agent (SNMPv3 components are gray)

Since the understanding of an AGENT++ agent's structure is a prerequisite for being able to use AgentX++, a short overview of the agent's components is given below:

1. **Initialize SNMP**
   Creates a *Snmpx* instance with the SNMP port the agent will listen on:

   ```
   Snmpx snmp(status, port);
   ```

2. **Initialize Message Processing**
   Initializes the SNMPv3 Message Processing component of a SNMPv3 agent with its SNMP message processing component (from 1.), its engine ID, and its boot counter:

   ```
   mpInit(&snmp, engineId, snmpEngineBoots);
   ```

3. **Creating the MIB**
   Creates the *Mib* instance that will hold all MIB information of the agent, and that will dispatch SNMP requests:

   ```
   mib = new Mib();
   ```

4. **Register the Request List**
   Registers a *RequestList* instance with the Mib instance created under 3. The request list receives, sends, and queues requests. It is thus responsible for managing requests.

   ```
   reqList = new RequestList();
   mib->set_request_list(reqList);
   ```

5. **Initializing the Request List**
   The request list needs the SNMP message processing component in order to be able to send and receive SNMP messages. The initialization of the request list is done by registering the SNMP message processing component created under 1.

   ```
   reqList->set_snmp(&snmp);
   ```

6. **Initialize Signals**
   An agent runs as a daemon process, which runs forever. It is stopped by sending a SIGTERM signal. In order to be able to shutdown the agent properly (i.e., storing MIB objects to disk), the agent has to catch this and other signals to perform any necessary actions, for example deleting the *Mib* instance created under 3.

7. **Adding MIB Objects**
   Adds (registers) all MIB objects (and groups) that should be statically supported by the agent. Dynamic MIB objects should be added to the MIB within the Main Loop component.

   ```
   ...
   ```

```
mib->add(new snmp_target_mib());
mib->add(new snmp_notification_mib());
mib->add("myContext", new if_mib());
...
```

8. Adding User Security Model & View Access Control Model MIB objects

A SNMPv3 agent must implement the USM and VACM MIBs. This component initializes a basic set of objects of those MIBs and adds them to the *Mib* instance created under 3. Besides, the VACM MIB needs to be registered with the request list created under 4. in order to be able to control the access to the MIB objects using the VACM information.

```
UsmUserTable *uut = new UsmUserTable();
...
// add non persistent USM statistics
mib->add(new UsmStats());
// add the USM MIB - usm_mib MibGroup is used to
// make user added entries persistent
mib->add(new usm_mib(uut));
// add non persistent SNMPv3 engine object
mib->add(new V3SnmpEngine());
mib->add(new MPDGroup());

Vacm* vacm = new Vacm(*mib);
reqList->set_vacm(vacm);
```

9. Initializing the MIB

The last action, that has to be executed before entering the main loop, is initializing the *Mib*. This includes, for example, loading MIB data from persistent storage as well as initializing MIB objects with management information got from the underlying managed systems. Afterwards, a cold start trap/notification may be sent.

```
mib->init();
...
coldStartOid coldOid;
mib->notify("", coldOid, 0, 0);
```

10. Main Loop

The main loop is responsible for processing incoming requests, monitoring the managed systems (that includes sending notifications), and updating management information that is not updated on demand (i.e., when a request for that information is pending).

```
Request* req;
while (run) {

       req = reqList->receive(2);

       if (req) {
           mib->process_request(req);
       }
       else if ((reqList->size() == 0) &&
                 (mib->get_thread_pool()->is_idle())) {
           mib->cleanup();
       }
```

```
}
mib->delete_thread_pool();
delete reqList;
delete mib;
delete agentx;
```

The example main loop above loops every two seconds. The timeout parameter for the *receive* method of the *RequestList* may be set to 0, which makes the *receive* method non blocking. With the *Snmpx::get_session_fds* method the API user can get the socket descriptor used for incoming SNMP requests, which enables the user to use a *select()* outside the API to control the agent's IO.

6.1 Creating an AgentX Master Agent

The way an AgentX++ master agent is implemented is only slightly different from an AGENT++ agent. Figure 12 shows, that an AgentX++ master agent needs an additional component which initializes the AgentX protocol stack. The additional component is marked by an *. Besides, the *Mib* used must be an instance of the *MasterAgentX* class. Consequently, the *Mib* instance creation component has to be changed too.

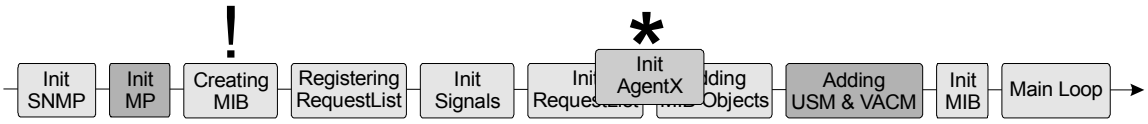

Figure 12: Differences between an AgentX++ master and an AGENT++ agent

The following list explains in detail the modifications that have to be made in order to create an AgentX++ master agent from an AGENT++ agent:

1. **Creating the Mib**
   Before the master agent's Mib instance can be created, the *MasterAgentXMib* and associated classes have to be included into the main program by

   ```
   #include "agentx_master.h"
   ```

   Instead creating an instance of the *Mib* class as described by component 3 of section 6, an AgentX++ master agent creates a *MasterAgentXMib* instance that will dispatch SNMP requests and AgentX requests:

   ```
   mib = new MasterAgentXMib();
   ```

2. **Initializing the AgentX protocol**
   An additional component that initializes the AgentX protocol stack has to be inserted after the initialization of the *RequestList*. With the initialization it can be determined whether the master agent should use UNIX domain socket connections and/or TCP connections for AgentX communication. This is done by using the *set_connect_mode*

method of the *AgentX*(*Master*) class. The file *agentx_def.h* defines two integer values, *AX_USE_UNIX_SOCKET* and *AX_USE_TCP_SOCKET*, which can used as parameter for that method. In order to support both connection types, the values can be ored.
The UNIX port location is set by *set_unix_port_loc* and the TCP port is set by *set_tcp_port.* The default values are shown by the below example:

```
AgentXMaster* agentx = new AgentXMaster();
#ifdef AX_UNIX_SOCKET
agentx->set_connect_mode(AX_USE_UNIX_SOCKET|AX_USE_TCP_SOCKET);
agentx->set_unix_port_loc("/var/agentx/");
#else
agentx->set_connect_mode(AX_USE_TCP_SOCKET);
agentx->set_tcp_port(705);
#endif
mib->set_agentx(agentx);
```

The last statement of this component should be registering the AgentX protocol stack with the *MasterAgentXMib* instance.
Please note that the UNIX domain socket related functions may not available on non UNIX systems. Thus, they should be used only enclosed with *#ifdef AX_UNIX_SOCKET*.

6.1.1 Configuration Options

When the *MasterAgentXMib* instance is created it automatically registers the AgentX MIB. Most of the managed objects defined in the AgentX MIB are informational only, i.e. they cannot be changed, except the *agentxSessionAdminStatus* object. It can be used to close a session with a subagent, by setting its value to *down(2)*. By default, the *agentxSessionAdminStatus* object is read-write accessible. In order to make it read-only, the *set_session_admin_status_writable* method of *MasterAgentXMib* has to be called:

```
mib->set_session_admin_status_writable(false);
```

Creating an AgentX Subagent

The implementation of an AgentX++ subagent is in some ways easier that implementing an AGENT++ agent, which on the other hand is not very difficult as well. Even if the master agent will support SNMPv3, the corresponding AgentX subagent needs not to deal with SNMPv3 related stuff at all. Therefore, all SNMPv3 related components have to be removed (omitted) from an AgentX++ subagent as it is illustrated by Figure 13.

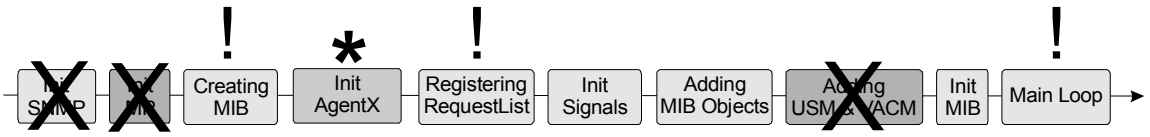

Figure 13: Differences between an AgentX++ subagent and an AGENT++ agent

Since a subagent does not handle SNMP requests directly, the initialization of the SNMP protocol stack as well as the initialization of the *RequestList* is not needed. The place of the latter one is taken by the initialization of the AgentX protocol stack. This and the other changes needed are listed in detail below:

1. **Creating the Mib**
   Before the subagent's Mib instance can be created, the *SubAgentXMib* and associated classes have to be included into the main program by

   ```
   #include "agentx_subagent.h"
   ```

   Instead creating an instance of the *Mib* class as described by component 3 of section 6, an AgentX++ subagent creates a *SubAgentXMib* instance that will dispatch the AgentX requests received from the master:

   ```
   mib = new SubAgentXMib();
   ```

2. **Initializing the AgentX Protocol**
   Because an AgentX subagent responds to AgentX requests, the AgentX protocol must be initialized before the agent's request list:

   ```
   AgentXSlave* agentx = new AgentXSlave();
   #ifdef AX_UNIX_SOCKET
   agentx->set_unix_port_loc("/var/agentx/");
   agentx->set_connect_mode(AX_USE_UNIX_SOCKET|AX_USE_TCP_SOCKET);
   #else
   agentx->set_connect_mode(AX_USE_TCP_SOCKET);
   #endif
   ```

   The AgentX protocol of the subagent is initialized in a similar way to a master agent (see list item 2 of section 6.1). The only difference is that the *AgentXSlave* class is used instead of the *AgentXMaster* class, which would be used with a master agent.

3. Registering the RequestList

Since the subagent does not handle SNMP requests but AgentX requests, the *RequestList* class cannot be used to receive, send, and queue the requests. Instead, the *AgentXRequestList* must be used, which takes the AgentX protocol stack from above as parameter.

```
reqList = new AgentXRequestList(agentx);
mib->set_request_list(reqList);
```

4. Initialize MIB

MIB initialization can be done likewise to AGENT++. The SubAgentXMib::init() method loads any persistent data from disk and then opens the connection to the master agent. In contrast to Mib::init() it may thus return FALSE if the connection could not have been established. This behavior can be used to implement a subagent that tries to reconnect to its master agent once a connection got broken.

```
mib->init();
```

5. Main Loop

When the connection to the master agent is cut off, the subagent may react in two possible ways. First, it may stop running, and second it may try to reestablish the connection to the master agent. An example for the main loop for the first case is shown below:

```
Request* req;
while (!mib->get_agentx()->quit()) {
      req = reqList->receive(20000);
      if (req) {
          mib->process_request(req);
      }
      else {
        mib->ping_master();
      }
      ...
}
```

The example subagents coming with AgentX++ stop running when the connection to the master is broken. This behavior may be modified to allow the subagent to try to reestablish the connection:

```
do {
      while (!mib->get_agentx()->quit()) {

            req = reqList->receive(40000);

            if (req) {
                  mib->process_request(req);
            }
            else {
                  mib->cleanup();
            }
      }
```

```
        mib->save_all(); // save persistent data to disk
        retries = 0;
        // Try to reconnect 10 times every 10 seconds
        while ((retries++ < 10) && (!mib->init())) {
             sleep(10);
        }
    }
    while (retries < 10); // stop after 10 consecutive failures
```

6.1.2 Using Shared Tables

Unlike normal tables, a shared table is not registered as a whole, but row by row (for more details see section 5.2.2). By it, several subagents can share a table. But since the subagents are supposed to be independent from each other, table sharing will not work reliable without index allocation (see section 5.1.6). Nevertheless, an *AgentXSharedTable* may be used without index allocation too, even though it does not make much sense. Therefore the following paragraph describes only the usage of shared tables with index allocation.

Like its super class *MibTable,* an *AgentXSharedTable* may be added to a *MibGroup* instance, in order to make it persistent and group it with other MIB objects. Although it is then registered with the master as part of the *MibGroup*, it will be able to register its rows separately, because less specific registrations (the group) are overruled by more specific registrations (the shared table's rows).

If persistency is not needed for a specific shared table, it may be not part of a *MibGroup* and it should be then added to the *SubAgentXMib* instance by its *add_no_reg* method after its initialization, in order to avoid the registration of the table as a whole. The registration of the whole table would not cause any damage, but it would slightly decrease performance.

When an *AgentXSharedTable* instance is created, the following information is needed:

- the object identifier of the table's entry object definition
- the index structure information as an array of *index_info* structures (one element for each index object)
- for each index object its object identifier
- a reference to the *SubAgentXMib* instance of the subagent, which will be used to process index allocation and registration requests
- the context name of the table's context

The below examples illustrates the creation of a shared *ifTable* (1.3.6.1.2.1.2.2) for the default context:

```
const index_info indIfEntry[1] = {
   { sNMP_SYNTAX_INT, FALSE, 1, 1 } }
const Oidx indIfEntryOIDs[1] = { "1.3.6.1.2.1.2.2.1.1" };

ifEntry::ifEntry(const OctetStr& context, SubAgentXMib* mib):
   AgentXSharedTable("1.3.6.1.2.1.2.2.1", // table OID
                     indIfEntry, 1,       // index info
```

```
                    indIfEntryOIDs,      // index obj OIDs
                    mib,                 // back reference
                    context)             // the context
{
   add_col(new ifIndex(colIfIndex));
   add_col(new ifDescr(colIfDescr));
   ...
}

AgentXSharedTable ifTable = new ifEntry("", mib);
```

The shared table created above could then be added to the default context of the subagent's MIB by:

```
mib->add_no_reg("", ifTable);
```

As describe in section 5.2.2, a new row is added to a shared table by first allocating an index value for the row and then, after the successful index allocation and row registration, setting the values of the row's objects. The index allocation can be done in three ways:

1. Allocating a new index value that has never been allocated before[10]:

   ```
   ifTable->new_index();
   ```

2. Allocating a new index value that is currently not used otherwise[11]:

   ```
   ifTable->any_index();
   ```

3. Allocating a specific index value:

   ```
   ifTable->allocate_index("42");
   ```

When the index allocation has been successfully completed, the row is registered and the table's *row_added* method is called (see Figure 7). Thus, the *row_added* method has to be overwritten, in order to set (initialize) the values of the new row. This can be accomplished for example by:

```
void ifEntry::row_added(MibTableRow* row,
                        const Oidx& index, MibTable*)
{
   // The row 'row' with 'index' has been added to the table.
   // Thus, the row's index is successfuly registered as well as
   // the row's region.
   // Now you may fill it with values
   row->get_nth(0)->replace_value(new SnmpInt32(index[0]));
   row->get_nth(1)->replace_value(new OctetStr("eth42"));
   ...
}
```

[10] Within the same context and during the runtime of the master agent.
[11] Within the same context.

A row is removed from the subagent by calling the *remove_row* method of the shared table. The index of the row is then deallocated and the row is deregistered. Both requests are sent independently to the master agent. When both requests are sent, the *row_delete* event is fired[12] and the row is removed from the table afterwards.

A complete example of subagents sharing a table comes with the AgentX++ source code. Please read also section 5.2.2 for more information on shared tables and on how to effectively use/implement row registration.

[12] A MibTable can fire five events, which are *row_added*, *row_delete*, *row_activated*, *row_deacrtivated*, and *row_init*. These events are named after the table's methods that are called when such an event is fired. Other tables can register for these events by calling the *add_listener* of a table.