The Simple Times™

THE QUARTERLY NEWSLETTER OF SNMP TECHNOLOGY, COMMENT, AND EVENTS
VOLUME 6, NUMBER 1 MARCH, 1998

The Simple Times is an openly-available publication devoted to the promotion of the Simple Network Management Protocol. In each issue, *The Simple Times* presents technical articles and featured columns, along with a standards summary and a list of Internet resources. In addition, some issues contain summaries of recent publications and upcoming events.

In this Issue:

The Simple Times is available as an online journal in HTML, PDF and PostScript. New issues are announced via an electronic mailing list. For information on subscriptions, see page 21.

An Overview of the AgentX Protocol

Matt White, Carnegie Mellon University
Smitha Gudur, Management Consultant

This article provides an introduction to the AgentX (Agent eXtensibility) protocol that was published as an IETF Proposed Standard in January 1998. Much of the information in this article is distilled from RFC 2257, which details the AgentX protocol, while the rest has been gleaned from experiences implementing the protocol. The AgentX protocol allows multiple subagents to make MIB information available in a way that is transparent to SNMP management applications.

AgentX is the first IETF standard-track specification for extensible SNMP agents. Prior to the publication of AgentX, users were forced to either use non-standard solutions or to run multiple SNMP agents on different UDP ports, probably using proxies to access them via a standard UDP port. Both approaches have problems. The lack of a standard often requires that subagent implementors have to support multiple subagent protocols if the same subagent is supported on different operating systems. Proxies are not transparent for the management station and thus require more intelligence on the managers side. In addition, running a number of subagents on a device usually costs less resources than running the same number of full-featured SNMP agents. The AgentX protocol provides a standard solution for the agent extensibility problem. It is designed to be independent from any particular SNMP version. An AgentX subagent will work with an SNMPv1, SNMPv2c and/or SNMPv3 master agent without any changes. A good overview of the various non-standard agent extensibility solutions and the AgentX problem space can be found in Volume 4, Number 2 of *The Simple Times*.

The AgentX protocol specifies a method for subagents to advertise to the master agent the information for which they are willing to take responsibility. Each AgentX subagent can operate in its own process space, providing a more robust alternative to monolithic SNMP agents. Additionally, processes can provide access to their internal state via the AgentX protocol, which is then accessible from a management station via SNMP.

With the complexity of server and application processes increasing daily, this last point becomes extremely important. Without a standard way to access the current state and historical data of server processes, large software systems quickly become unmanageable. By making this information available through AgentX, we can use standard SNMP management tools to administer software systems.

Architecture, Protocol Features and Issues

An AgentX SNMP environment consists of two types of processes: a master agent and one or more subagents, communicating with each other either over TCP or a Unix domain socket. The master agent speaks both AgentX and SNMP. It is the master's job to maintain a table of which subagents are responsible for which MIB regions. When the AgentX master receives a request via SNMP, it finds the subagent(s) responsible for the requested MIB region and dispatches appropriate AgentX requests. The master itself contains almost no management information with the possible exception of information regarding currently connected agents and allocated MIB regions as described below in the AgentX MIB.

The AgentX subagents are responsible for providing access to management information. When a subagent is started, it contacts the master and registers the various MIB regions for which it has information. The subagent has no concept of SNMP or even other subagents so the AgentX master must arbitrate all conflicts between subagents. As part of this arbitration service, the master provides an index allocation service and resolves overlaps in MIB region registrations in a deterministic manner. In addition to providing a means with which to dispatch SNMP requests to subagents, the AgentX protocol also defines how the master should go about avoiding and, if avoidance fails, resolving conflicts.

Features

The feature set of the AgentX protocol is designed to allow the protocol to be transparent to the SNMP management station and to eliminate any need for subagent to subagent communication. There are several features specified in the protocol that make this transparent operation possible. Arguably the most important of these features are two-phase commits, index allocation and registration conflict arbitration.

- *Sets operations maintain atomic nature.* SNMP Sets are atomic all-or-nothing operations. AgentX preserves this property, even when the OIDs specified in a Set request are registered by different subagents. This functionality is obtained through the use of two-phase commits.
- *Index allocation.* The master agent is responsible for providing non-conflicting table indices so that multiple subagents can provide access to management information in separate rows of the same table. As long as all subagents use the master to allocate indices in a table before registering them, we can guarantee that there will be no registration conflict.
- *Registration conflict arbitration.* The master agent is also responsible for resolving any registration conflicts that arise by subagents attempting to register duplicate and/or overlapping MIB regions. The resolution process gives preference to subagents registering the most specific region and then by a priority variable that is included in the registration PDU.

The AgentX MIB

The AgentX MIB is not a part of RFC 2257 but is rather an Internet Draft being discussed within the AgentX working group. It is therefore possible that the MIB changes before it goes on the standards track.

The AgentX MIB allows managers to identify the number of subagent sessions that are open with the master agent. It can identify the MIB regions or MIB objects that a subagent implements. Managers can gather statistics and operational parameters such as the timeout interval for responses from a subagent and can determine the priority at which a subagent registered a particular MIB region.

The AgentX MIB is organized into four groups with three tables.

- The agentxGeneral group provides information about the master agent's AgentX support, including the protocol version and transport mechanisms.
- The agentxConnection group provides information describing the current set of connections capable of carrying AgentX sessions. The heart of the agentxConnection group is the agentxConnectionTable, which is defined as follows:

```
AgentxConnectionEntry ::= SEQUENCE {
  agentxConnIndex              Unsigned32,
  agentxConnOpenTime           TimeStamp,
  agentxConnTransportDomain    TDomain,
  agentxConnTransportAddress   TAddress,
  agentxConnSessions           Gauge32
}
```

- The agentxSession group provides information describing the current set of AgentX sessions. The heart of the agentxSession group is the agentxSessionTable:

```
AgentxSessionEntry ::= SEQUENCE {
  agentxSessionIndex       Unsigned32,
  agentxSessionObjectID    OBJECT IDENTIFIER,
  agentxSessionDescr       Utf8String,
  agentxSessionAdminStatus INTEGER,
  agentxSessionOpenTime    TimeStamp,
  agentxSessionAgentXVer   INTEGER,
  agentxSessionTimeout     INTEGER
}
```

  Entries in the session table exist in a many-to-one relationship with entries in the connection table.

- The agentxRegistration group provides information describing the current set of registrations. The heart of the agentxRegistration group is the agentxRegistrationTable:

```
AgentxRegistrationEntry ::= SEQUENCE {
  agentxRegIndex    Unsigned32,
  agentxRegContext  OCTET STRING,
  agentxRegStart    OBJECT IDENTIFIER,
  agentxRegEnd      OBJECT IDENTIFIER,
  agentxRegPriority Unsigned32,
  agentxRegTimeout  INTEGER,
  agentxRegInstance TruthValue
}
```

  Entries in the registration table exist in a many-to-one relationship with entries in the session table.

The relationships between these tables are expressed in the indexing structure. The agentxConnectionTable is indexed by agentxConnIndex, the agentxSessionTable by agentxConnIndex and agentxSessionIndex, and the agentxRegistrationTable by agentxConnIndex, agentxSessionIndex and agentxRegIndex.

Security Issues

The AgentX protocol has no built in access control method to control the registration process. That means that any agent that connects to the master can register whatever region it desires. A malicious user could exploit this lack of access control to provide false information to the SNMP management console. Where the transport itself provides access control, such as Unix domain sockets, this is not an issue since there is adequate security provided at a lower layer. When TCP is used as the transport for AgentX there is no reliable way to prevent malicious subagents from connecting to the master.

AgentX has a notion of context just as SNMP does. Even though there is no way to prevent a malicious subagent from connecting to a master when TCP is the transport, unauthorized subagents can be prevented from providing access to misinformation for a manager. Using contexts to prevent malicious registrations provides some weak security as long as care is taken not to advertise contexts.

Fortunately, there are implementations of Unix domain sockets for all major platforms, including Microsoft Windows(tm). Unix domain sockets provide access control and an authentication mechanism (the subagent socket is owned by the subagent process' user). Since AgentX is primarily a means of inter-process communication, Unix domain sockets provide an appropriate transport with sufficient security from unauthorized connections.

Other Considerations

AgentX is primarily a protocol for inter-process communication between the master and its subagents. It is not intended to be used over a network wire, although it can be. There are a number of design decisions that reflect this fact that are noted in RFC 2257. The most notable of these is the protocol default of using machine byte ordering instead of network byte ordering. The subagent defaults to its platforms' native byte ordering and then communicates this choice to the master.

Other decisions that reflect the nature of this protocol are the 32-bit alignment of data and large PDU sizes. The large PDU sizes in particular provide lower protocol overhead in a way that might not be practical over a local area or wide area network.

AgentX Protocol Flow

In this section we will discuss the interaction between the AgentX master and subagent in more detail. This is not intended to be an in depth explanation of the workings of AgentX and, as such, does not detail every eventuality in an AgentX session. Interested readers should read RFC 2257 for more information.

PDU Description

Each AgentX PDU contains a 20-byte header. The first 32 bits of the header contain the protocol version number (currently 1), the PDU type, an 8 bit flags field and an 8 bit reserved field. The flags currently in use are `INSTANCE_REGISTRATION`, `NEW_INDEX`, `ANY_INDEX`, `NON_DEFAULT_CONTEXT` and `NETWORK_BYTE_ORDER`. The AgentX Register PDU uses `INSTANCE_REGISTRATION`; the AgentX IndexAllocate PDU uses `NEW_INDEX` and

ANY_INDEX. The NETWORK_BYTE_ORDER flag is used by the subagent to communicate its platforms' native byte ordering to the master.

The NON_DEFAULT_CONTEXT flag alerts the recipient that an octet string follows immediately after the header. The use of this context field is implementation dependent but may refer to an SNMP context or some mapping between contexts done by the master agent. The implementation of non-default contexts is optional.

The remaining 16 bytes are organized into four 32-bit words. These fields are sessionID, transactionID, packetID and payload length. The sessionID identifies a session created by the subagent. The transactionID refers to the SNMP request that this AgentX PDU is servicing. The packetID is used to pair AgentX request and response PDUs. The payload length is simply the length of the remainder of the PDU, in bytes.

Following the PDU header is a payload of some sort. The length and makeup of this payload depends on the type of AgentX PDU being sent. An exact description of each of the 18 AgentX payload types is beyond the scope of this article. Interested readers may find this information in RFC 2257.

Opening a Session

Once a connection has been established between a subagent and master, the subagent is free to establish one or more sessions with the master. Sessions are opened with the AgentX Open PDU. The Open PDU contains a default timeout value for the session, an OID to associate with the session and a description of the session. The session identifier in an Open PDU is ignored. The use of the session OID and description fields are implementation specific, but may be used to populate the sysORTable RFC 1907.

When an AgentX master receives an Open PDU from an attached subagent, it assigns an unused session identifier and creates an AgentX Response PDU with the session identifier set to the new session number. The response is sent to the subagent and the session is considered open.

At any time a subagent can send an AgentX Close PDU in order to disconnect a session from the master agent. When the master agent receives an AgentX Close PDU, it flushes its pending queue for that session, de-allocates all indices, unregisters all OID regions and then closes the session. The master agent may send an AgentX Close PDU to a subagent as the result of an SNMP management request. In this case, the master agent sends an AgentX Close PDU to the subagent with the reason set to 'reasonByManager'.

Index Allocation

Index allocation is a service provided by the master to allow registration of the rows of a conceptual table within a MIB. For instance, if the subagent wishes to register a row of ifTable (RFC 2233), it might request an index allocation for the column ifIndex. Depending on the parameters of the request, the master agent will either attempt to issue a specific index, a currently unused index or a never used index in this table. If an index of the type requested is available, it is returned to the subagent in an AgentX Response PDU.

Multiple indices may be requested in a single AgentX IndexAllocate PDU. In the case of multiple allocations, all indices must be successfully processed before any indices are allocated. If the allocation fails, the master returns an AgentX Response PDU indicating the offending allocation request.

Presumably after a subagent has successfully allocated an index, it will then follow up with a registration request. Since registration does not look at index allocation, it is possible that a registration of an allocated index will fail due to a poorly behaved subagent registering an index that it has not first allocated. In this case, the subagent should attempt to allocate another index and repeat the registration process.

If a subagent does not continue to require control of an index, it may be released with an AgentX IndexDeallocate PDU. This releases the index back into the pool that may be allocated but does not unregister any OIDs that are within the subagent's authoritative region. Therefore, the subagent should unregister all regions underneath an index before releasing the index itself.

In practice, none of this is usually necessary. A subagent will generally connect to a master and will remain connected and relatively static for the remainder of its life. A well-behaved subagent will close sessions before shutting down, but that obviates the need to deallocate indices and unregister regions because that is taken care of as part of the Close PDU's processing.

OID Registration

When a subagent wishes to declare itself authoritative for a set of OIDs, it sends an AgentX Register PDU to the master agent. Depending on the OID range being registered, it may be necessary for the subagent to allocate the index being registered. A registration succeeds if it does not cause ambiguity as to which subagent is authoritative for a region. Authoritativeness is determined as follows:

1. The most specific registration is authoritative. That is, the registration with the longest OID. Ranges do not count towards specificity. For example, 1.3.6.1.2.1.2.2.1.[1-14].1 is no more specific than

1.3.6.1.2.1.2.2.1.[1-22].1 and may be rejected if authoritativeness cannot be determined via rule 2.

2. When authoritativeness cannot be determined from rule (1), the registration with the lowest priority value is considered authoritative. If a subagent attempts to register a region having the same specificity and priority as an existing region, the new registration is rejected as a 'duplicateRegistration'.

Once the master has determined that a new registration request will not cause an ambiguity, the new region is entered into the master's dispatch table and an affirmative response is sent to the subagent. Only one region may be registered in a single PDU, however multiple OIDs may be specified in a region through the use of the range subid, reducing the number of Register PDUs that must be sent. If a subagent wishes to be authoritative for more than one region, multiple PDUs must be sent.

As with index allocation, there is an inverse to the registration process. If a subagent no longer wishes to respond to an OID region, it should send an AgentX Unregister PDU with the same OID range as the original registration. If the region specified by the unregister request exactly matches a region allocated to the session making the unregister request, that region is unregistered and an affirmative response is sent to the subagent. If the region being unregistered does not correspond to a registered region or if that region is not registered to the session making the unregister request, the request fails and a 'notRegistered' response is sent to the subagent.

SNMP Get, GetNext and GetBulk

The AgentX Get, GetNext and GetBulk PDUs are the primary means of servicing the SNMP requests of the same name. When the master receives an SNMP request that results in it issuing one or more AgentX PDUs it will generate a unique transaction ID to identify the SNMP request. The generated transaction ID is included in each AgentX PDU sent. All AgentX PDUs generated as the result of an SNMP request are sent to the session(s) that is currently deemed authoritative for the OID(s) being requested.

All AgentX Get, GetNext and GetBulk PDUs make use of search ranges. A search range contains a starting OID, an ending OID and an include field which may either be 0 or 1. There may be other fields in the AgentX Get, GetNext or GetBulk PDU, depending on the PDU type.

An AgentX Get PDU may contain one or more search ranges. If the starting value is instantiated by the receiving subagent, the value for this instantiation is written into the response packet. If the OID lookup fails on the subagent, a response of either 'noSuchObject' or 'noSuchInstance' is returned for that variable in the response packet. In either case, processing continues with the next search range in the AgentX Get PDU.

An AgentX GetNext PDU may also contain one or more search ranges. If the 'include' field of the starting field is set to 0, then the value returned for that search range is the closest lexicographical successor to the starting OID, non-inclusive. If the include field is set to 1, then the search is inclusive. In either case, if the ending OID is not null then the object whose value is returned must lexicographically precede the ending OID. If no variable is instantiated within the search range, the returned value is set to 'endOfMibView'.

SNMP GetBulk requests can either be processed as multiple AgentX GetNext PDUs or through AgentX GetBulk PDUs. AgentX GetBulk PDUs allow for more efficient processing of SNMP GetBulk requests. The AgentX GetBulk PDU is processed similarly to the GetNext PDU, but has additional fields that allow the master to fine-tune the information returned by the request.

SNMP Set

SNMP Sets are implemented through the use of the AgentX TestSet, CommitSet, UndoSet and CleanupSet PDUs. These four PDU types provide the two-phase commits necessary to provide atomic SNMP Sets across multiple subagents.

The first stage of SNMP Set processing involves sending one or more AgentX TestSet PDUs to the subagents specified. The subagents extract the values and test whether the Set operation would succeed. Since a single TestSet PDU may include several variables to set, each one must be checked for validity. If all variables may be set, then a response is sent back to the master that the processing for the SNMP Set operation may proceed. If one of the variables cannot be set, then an error is returned to the master indicating the offending variable.

The next stage of SNMP Set processing depends on whether all TestSet PDUs were successful. If all TestSet requests were successful then the master sends a set of CommitSet PDUs, which informs each subagent that they should actually commit the changes to memory. CommitSet operations should almost always succeed. However, there is still a chance that they may fail. In the event that a CommitSet operation fails, the master agent should send an UndoSet PDU to undo the changes made by the previous CommitSet operation. The receipt of an UndoSet PDU by the subagent indicates the end of SNMP Set processing. If an UndoSet fails, the master takes no action, which may leave the managed resource in an inconsistent state. Implementation designers should take care that this is an extremely unlikely event although there are some instances when this is unavoidable.

If either a TestSet fails or a CommitSet succeeds, the master agent follows up with a CleanupSet PDU, indicating the end of SNMP Set processing. No response is sent to the CleanupSet PDU.

SNMP Traps and Informs

The AgentX Notify PDU provides the ability for a subagent to emit event reports for sending SNMP Traps. The subagent simply sends a Notify PDU to the master. What the master agent does with this event report is implementation specific. A master may send an SNMP Trap to a configured set of hosts or perform some other action, like discarding the notification. The SNMPv3 target and notification MIBs (RFC 2273) can be used to configure target hosts and to setup a notification filter for these targets. An SNMPv3 Inform may also be sent. However, there is no way within the AgentX protocol to inform the subagent whether or not the SNMPv3 Inform reached its destination(s).

Other AgentX administrative PDUs

The AgentX AddAgentCaps PDU provides a method for a subagent to modify its capabilities in the sysORTable (RFC 1907) for a given context. As with the other registration PDUs, there is an inverse operation, RemoveAgentCaps, which can be used to remove agent capabilities for a context. Agent capabilities are on a per-session-per-context basis.

The remaining AgentX PDU is the Ping PDU, which allows a subagent to test a master's ability to respond to AgentX requests. If a master fails to respond to a subagent's ping, the subagent will have to close and reopen the session with the master. Due to the multiple process nature of AgentX, the ability to re-open and reregister OIDs is an important one for the subagent to possess.

The CMU AgentX Implementation

One of the authors of this article is also involved with developing a reference implementation of the AgentX protocol. A brief description of this work is given here merely to give a taste of things to come. In the coming months, we will likely see the release of several excellent implementations of the protocol, each targeted at a different customer base. This is one reason why a standards based approach to agent extensibility is preferred over proprietary standards.

Carnegie Mellon University has long been active in the SNMP world. It should come as no surprise then that we are also interested in AgentX, being a natural evolution of SNMP. We are currently developing an AgentX master (which is a Unix daemon process) and a subagent library under Solaris. The finished product is expected to run under most modern operating systems that support Unix domain sockets and POSIX threads. Target systems include: Solaris, Linux and IRIX, but we also hope to have code running under FreeBSD, NetBSD and Windows NT.

The goal of the CMU subagent library is the painless instrumentation of already existing code. At CMU, we SNMP instrument just about everything we can manage and hope to extend that practice with the introduction of AgentX. When our AgentX library is initialized, it spawns a thread to handle incoming requests from an AgentX master and then returns. The subagent is then free to open sessions with the master, register OIDs and map OIDs to locations in memory. Some synchronization routines are provided where there are sequential dependencies.

When the AgentX thread receives an AgentX PDU from the AgentX master in response to an SNMP Get, GetNext, GetBulk or Set request, it looks up each OID from the PDU in a tree structure stored in memory. The data in this structure contains a number of user provided callback functions. These callback functions fill in the values for Get and GetNext requests. Similarly, Set processing is done through a series of callback functions. There will be a set of generic callback functions provided in the AgentX library for mapping objects to memory locations.

By providing a simple way of mapping OIDs to locations in memory, we provide an uncomplicated way to make accessible whatever variables the system designer wishes. This interface emphasizes CMU's focus of instrumenting server processes, which would be difficult if the AgentX instrumentation required constant attention throughout the server code.

Since this is an implementation in progress, this information is subject to change at any time. For the most recent information regarding the CMU implementation of the AgentX protocol, please visit the Web page: http://www.net.cmu.edu/projects/agentx/.

Carnegie Mellon's plans for AgentX

CMU currently runs SNMP instrumented versions of BIND and our own DHCP server. These SNMP agents respond to SNMP requests on alternate UDP ports. With AgentX, we will no longer take this approach but will rather incorporate AgentX subagents in these processes. It is also quite likely that we will instrument other server processes in order to increase our network information gathering capabilities.

Another place we will make extensive use of AgentX is our own homegrown network monitoring system called

NADINE. NADINE consists of a hub process, some number of monitors and any number of clients. The monitors post information about what they are monitoring via SNMP. This information is then read by the hub process and correlated so that related network events are grouped together in threads. These threads are then read by the clients, which filter out information that the user does not care about and leaves only the perceived root causes of those network situations that the user does care about. Other information can be read about a thread by expanding it and viewing the underlying events that make up the thread.

With the advent of AgentX, all the monitors running on a system will be able to coexist on the same port, allowing for more automatic configuration. In addition, doing an SNMP walk of a machine running monitors will reveal what events those monitors are currently tracking on the network. By using our AgentX API, we will eliminate the current need to constantly service SNMP requests within our monitor code because this task is handled automatically by the AgentX library.

Conclusion

The AgentX protocol provides a framework for providing access to management information via SNMP without drastically increasing the complexity of the individual agents. AgentX is invisible to the management station and so existing tools can be used to gather information from the subagents. The plug-in architecture keeps agent design simple while having subagents running in separate process spaces improves the robustness of the SNMP system running on a given network device.

While AgentX is not and was never intended to be capable of supporting every possible subagent configuration, it does support the vast majority of configurations. By not attempting to do everything, the protocol designers have come up with a protocol that does most things extremely well while not unduly increasing implementation difficulty. Hopefully this will lead to a plethora of implementations, both public domain and commercial.

As the AgentX protocol heads into the last legs of the standards process, we will begin to see AgentX implementations appear on various FTP sites as well as within commercial products. This summer there will likely be an AgentX interoperability test of different AgentX implementations. The AgentX working group is planning to gather implementation experiences and to complete the AgentX MIB at the 42nd IETF meeting.

Hopefully, with the introduction of AgentX, we will see a drastic increase in the amount of management information provided by network devices as well as increased reliability of those SNMP agents.

WinSNMP v2.0 - Evolution of an industry-standard API

*Bob Natale, ACE*COMM*

The WinSNMP API has progressed quite a bit since *The Simple Times* published a “sneak peek” overview of the early design discussions in Volume 2, Number 3 (May/June 1993). This article will outline the overall progress and detail certain key aspects of the current version of the API. The presentation is targeted at SNMP developers and includes a fair amount of technical detail. For those who may not want to digest the details right away, a quicker reading will still give you a good taste of how WinSNMP works. Terms that have a special meaning in WinSNMP are shown in italics when first used in this article.

Introduction

The WinSNMP Industry Forum is an open, ad hoc group of technical contributors interested in the development of standard SNMP application programming interfaces. While the subset of active participants has varied from time to time since its inception in 1993, at crucial points in the evolution of the WinSNMP specifications the make-up of that subset has included a significant number of knowledgeable and experienced SNMP developers. Virtually all major SNMP “platform” vendor organizations (notably HP, Cabletron, and IBM) have played important roles, as has a large body of application developers. User organizations have generally been under-represented.

The group released the first production version (1.1) of the WinSNMP API in June of 1994. A revised version (1.1a) – with additional clarifying text comprising the majority of the changes – followed in August of 1995. This latter version achieved wide deployment, initially in Win16 environments (Windows and Windows for Workgroups) and later in Win32 environments (Windows 95 and Windows NT).

Needless to say, with increased use over time several opportunities for improvement were identified and, after a series of “on-list” deliberations, Version 2.0 of the WinSNMP API was released for deployment at the beginning of November, 1997. The major changes incorporated in WinSNMP v2.0 are:

- Support for operating environments other than Microsoft Windows, via the new `SnmpCreateSession()` function.
- Support for agent applications, via the new `SnmpListen()` function.

- Support for transparent SNMPv1 Trap-PDU generation, in accordance with the Informational RFC 2089.
- Required retransmission and timeout notification behavior.
- Support for SNMPv2c, via required compliance with RFCs 1901-1908.

This article will focus on these changes. You may find it helpful in reading the rest of this article to have access to the WinSNMP v2.0 Addendum (winsnmp2.txt) and the standard WinSNMP header file (winsnmp.h) which you can download from http://www.winsnmp.com, along with the base WinSNMP v1.1a specification (winsnmp.doc) and a number of other useful files, sample source code, and demo applications.

The overall WinSNMP API consists of 46 functions, divided into six groups:

1. Database Functions:
   `SnmpGetRetransmitMode`, `SnmpSetRetransmitMode`, `SnmpGetTimeout`, `SnmpSetTimeout`, `SnmpGetRetry`, `SnmpSetRetry`, `SnmpGetTranslateMode`, `SnmpSetTranslateMode`, `SnmpGetVendorInfo`
2. Communications Functions:
   `SnmpStartup`, `SnmpCleanup`, `SnmpOpen`, `SnmpClose`, `SnmpSendMsg`, `SnmpRecvMsg`, `SnmpRegister`, `SnmpCreateSession`, `SnmpListen`, `SnmpCancelMsg`
3. PDU Functions:
   `SnmpCreatePdu`, `SnmpGetPduData`, `SnmpSetPduData`, `SnmpDuplicatePdu`, `SnmpFreePdu`
4. Varbindlist Functions:
   `SnmpCreateVbl`, `SnmpDuplicateVbl`, `SnmpFreeVbl`, `SnmpCountVbl`, `SnmpGetVb`, `SnmpSetVb`, `SnmpDeleteVb`
5. Entity/Context Functions:
   `SnmpStrToEntity`, `SnmpEntityToStr`, `SnmpStrToContext`, `SnmpContextToStr`, `SnmpFreeEntity`, `SnmpFreeContext`, `SnmpSetPort`
6. Utility Functions:
   `SnmpEncodeMsg`, `SnmpDecodeMsg`, `SnmpStrToOid`, `SnmpOidToStr`, `SnmpOidCopy`, `SnmpOidCompare`, `SnmpFreeDescriptor`, `SnmpGetLastError`,

The five new functions added for WinSNMP v2.0 are `SnmpGetVendorInfo`, `SnmpCreateSession`, `SnmpListen`, `SnmpCancelMsg` and `SnmpSetPort`.

Extended Operating Systems Support

WinSNMP uses the concept of a *session*. Sessions are used for two purposes:

1. To identify a set of resources – e.g., PDUs, varbindlists (VBLs), *entities*, and *contexts* – for internal management purposes.
2. To represent a channel for communications between the SNMP *engine* (typically implemented in Windows as a dynamic link library) and a particular "code path" in the associated application which uses WinSNMP.

In this latter role, WinSNMP sessions were originally constructed in such a way – via the `SnmpOpen()` function – as to effectively limit implementations to the various Microsoft Windows environments. Since those environments were the specific targets of the original effort, that made sense at the time. With growing exposure of the API and of the various management applications being built with it, however, many people saw the need to extend the session construct to enable implementation in other environments. The result is the new `SnmpCreateSession()` function which both provides enhanced support for Microsoft Windows environments and permits support for non-Windows environments.

The `SnmpOpen()` function – which, like all WinSNMP v1.1a functions, is fully retained in WinSNMP v2.0 – has the following prototype:

```
HSNMP_SESSION
SnmpOpen(IN HWND hWnd, IN UINT wMsg);
```

In calling this function, the application passes a message identifier (`wMsg`) and a window handle (`hWnd`) to which the engine sends the message identifier when a *notification event* occurs for the session. There are four kinds of notification events:

1. Receipt of a Response-PDU in reply to an outstanding Request-PDU.
2. Receipt of a Trap-PDU or InformRequest-PDU in accordance with an `SnmpRegister()` filter.
3. Receipt of a Request-PDU in accordance with an `SnmpListen()` filter.
4. Discard of a pending Request-PDU message due to expiration of the overall timeout interval, for messages sent with `RetransmitMode` enabled.

Upon success, `SnmpOpen()` returns a non-zero session identifier. Many of the remaining WinSNMP functions take a session identifier as their initial parameter, enabling both the resource management and channel communications functions mentioned earlier.

WinSNMP is designed for maximum *asynchronicity* of operations. That is, the transmission of an SNMP request is largely dissociated from the receipt of the corresponding response in the engine. This design encourages the deployment of high-throughput, fault-tolerant, multi-threaded implementations. Clearly, for GUI applications running in Microsoft Windows environments, SnmpOpen() provides a natural means of enabling multi-session asynchronous operations. Typically, WinSNMP engines have also leveraged the parallel asychronous capabilities of the Windows Sockets (WinSock) API (now a standard component of Microsoft's Win32 platforms) to further enable such high-performance multi-session applications on relatively low-cost machine configurations.

A *notification method* is the engine's means of signalling a session when a notification event occurs for it. The single notification method offered by SnmpOpen() presented a major roadblock for WinSNMP support on other operating systems (most notably UNIX). A good portion of the heavy-duty technical collaboration on the WinSNMP mailing list in developing v2.0 went into the definition of the new SnmpCreateSession() function. The primary purpose of this new function is to add *callback function* support as an alternative notification method. In addition, SnmpCreateSession() is designed as a superset of SnmpOpen() so that, while the latter is fully retained in the API, the new function can be used in all new or modified applications as a single means to provide either or both notification methods. The SnmpCreateSession() function has the following prototype:

```
HSNMP_SESSION
SnmpCreateSession(IN HWND hWnd,
                  IN UINT wMsg,
                  IN SNMPAPI_CALLBACK fCallBack,
                  IN LPVOID lpClientData);
```

If the fCallBack parameter is NULL, then the function is evaluated as though it were simply a call to SnmpOpen() by using the hWnd and wMsg parameters. If fCallBack is non-NULL, it is taken as the address of a function to invoke as the notification method for this session, and the hWnd, wMsg, and lpClientData are all values which will be passed, along with several others, to the fCallBack function for each notification event. The callback function prototype must match the SNMPAPI_CALLBACK type:

```
typedef SNMPAPI_STATUS
    (CALLBACK *SNMPAPI_CALLBACK)
        (IN HSNMP_SESSION hSession,
         IN HWND hWnd,
         IN UINT wMsg,
         IN WPARAM wParam,
         IN LPARAM lParam,
         IN LPVOID lpClientData);
```

SNMPAPI_STATUS is a WinSNMP standard return value indicating success or failure. The hWnd, wMsg, and lpClientData parameters contain the values passed by the application when it called SnmpCreateSession() to open the session identified by the hSession parameter. In addition, the engine passes two other values:

1. wParam - Status indicator used to distinguish between a normal notification, a timeout notification or other "transport layer" errors as defined in winsnmp.h.

2. lParam - The RequestID of the corresponding Request-PDU.

As with the window/message notification method, the callback function checks the value of wParam and normally switches on it to either process an incoming SNMP message using SnmpRecvMsg(), or to perform any necessary PDU clean-up and/or user alerting steps. In either case, the lParam parameter contains the RequestID value associated with the corresponding notification event.

The RequestID value in lParam is most useful for timeout notification processing (more about this in a later section), since no further identification of the timed out message is available from the WinSNMP engine at this point. For normal "response to request" PDUs, it can be useful to know what Response-PDU the engine is signalling, but there is no guarantee that the next call to SnmpRecvMsg() will return exactly that Response-PDU. The engine may return any valid available SNMP message for the calling session on any given invocation of SnmpRecvMsg(). Synchronicity between message arrival notification (message signalling) and message delivery (in response to acting on that notification) is not mandated.

In addition, the session identifier parameter hSession can be used to further simplify notification event processing. This is often accomplished by opening multiple sessions, whether for groups of agents being managed (e.g., by subnet, vendor, or type of device) or for directing traps, informs, and requests (for agent applications) to sessions opened for those specific purposes.

In general, SnmpCreateSession() has proved to be very successful in its still brief lifetime. Not only has it been used to implement "console mode" and "service mode" applications on Win32 platforms (while maintaining the backwards-compatible `SnmpOpen()` "GUI mode"), it has also enabled ports of the WinSNMP library to multiple UNIX platforms (commercial implementations are now available for AIX, HP-UX, and Solaris). On the downside, it was hoped that the design of `SnmpCreateSession()` would enable X-Windows "GUI mode" applications on UNIX platforms too, but thus far none has emerged (to the best of this author's knowledge).

Support for Agent Applications

Originally, the WinSNMP API was qualifed as the "WinSNMP/Manager API," in recognition of the facts that:

- The industry participants felt that the management applications area had (at the time, at least) greater need of such attention and that this domain would prove to be more amenable to standardization than the agent domain.
- The IETF was haggling, off and on, about whether or not to standardize extensible agent technology.

No function was included in the WinSNMP repertoire to enable an agent to bind to a port to listen for incoming SNMP request messages. After a fairly short time in the field, however, it became apparent that a sizable need existed for agent applications to run over WinSNMP and, moreover, extending WinSNMP to yield this capability would require very little work. Several WinSNMP vendors had already added this capability to their implementations in generally straight-forward yet nonetheless non-standard (and slightly different) ways – this promised to be a source of pain for agent application developers and consumers alike.

So, WinSNMP v2.0 added the `SnmpListen()` function to address this need. Now bear in mind that we are speaking only about the "SNMP front-end" portion of an agent application. Neither the so-called "method routines" nor the "instrumentation back-end" is addressed by WinSNMP v2.0.

The ease with which this level of support for agent applications could be added is exemplified by the function prototype itself:

```
SNMPAPI_STATUS
SnmpListen(IN HSNMP_ENTITY hEntity
           IN SNMPAPI_STATUS lStatus);
```

WinSNMP uses the concept of an *entity* as an endpoint of an SNMP transaction. In most cases, this is simply another term for an agent. In the original vision, "party-based SNMPv2" (now historic) provided the model for an entity. In the interim, this has fallen into disuse. It is anticipated, however, that SNMPv3 will be cause for its resurrection (more on this in a later section). For now, suffice it to say, that a WinSNMP entity is created via the `SnmpStrToEntity()` function and has the following minimal attributes (and may have others specific to an implementation):

- Owning session
- SNMP version indicator
- Protocol family indicator
- Network address
- UDP port or IPX socket number
- Timeout interval (policy) value
- Retry count (policy) value

The owning session identifier is passed as a parameter to `SnmpStrToEntity()`. The protocol family is determined from the format of the transport address string (implementations typically support IP/UDP and IPX), which is also passed as the second and final parameter to `SnmpStrToEntity()`. The UDP port or IPX socket number assumes an initial default value, as do the timeout interval and retry count values. These default values come either from the engine (if the `SnmpStrToEntity()` call is made while the application is operating in an untranslated mode) or from the implementation-specific "local configuration database" (if the call is made from the translated mode). The SNMP version supported by the instantiated entity is also derived from the `TranslateMode` value (which may be set by the application with the `SnmpSetTranslateMode()` function). Subsequently, an entity's port/socket, timeout interval, and retry count values can be changed by the application using the `SnmpSetPort()`, `SnmpSetTimeout()`, and `SnmpSetRetry()` functions, respectively.

The `lStatus` parameter to `SnmpListen()` may take a value of `SNMPAPI_ON` or `SNMPAPI_OFF`. The former is used to initialize an agent entity in listening mode on its currently assigned port; the latter is used to terminate the agent role and free the port for other uses. Only one agent entity may listen on a given port at one time. The agent may, of course, function either as a native or as a proxy agent and may be either monolithic or extensible. In either case, `SnmpListen()` returns appropriate failure indicators if it finds the requested port already in use.

Once `SnmpListen()` returns successfully from an `SNMPAPI_ON` invocation, the agent session is notified of received SNMP management requests for it to process in the same manner that a manager session is notified of received responses. The agent application then calls `SnmpRecvMsg()` to pull the message off the queue and then `SnmpGetPduData()` to gain access to the varbindlist. After processing the varbindlist it can return the corresponding Response-PDU by simply using the `srcEntity` value from `SnmpRecvMsg()` as the `dstEntity` value to `SnmpSendMsg()` to direct the response to the appropriate local or remote manager session.

`SnmpListen()` has already led to the deployment of several WinSNMP-based agents, both monolithic and extensible. At least one commercial implementation of AgentX (RFC 2257) over WinSNMP for Win32 environments has been announced. In addition, WinSNMP applications may freely mix agent and manager operations, enabling "mid-level manager" type applications. One such application – consisting of both a mid-level manager and a pre-AgentX extensible agent environment, all running over WinSNMP for Win32 – is already being used in public information kiosks produced by a major computer manufacturer.

Support for SNMPv1 Trap-PDU Generation

From its inception, WinSNMP has sought to promote the use of SNMPv2 (now taken to mean SNMPv2c). As part of that intention, it was decided at the outset to deliver all SNMP traps to sessions waiting for them via `SnmpRegister()` as SNMPv2 Trap-PDUs. Therefore, WinSNMP engines have always converted received SNMPv1 Trap-PDUs to SNMPv2 Trap-PDUs internally (guided by RFC 1908), prior to delivering them to the registered manager sessions. Also, inasmuch as WinSNMP v1.1a was oriented almost entirely toward manager applications, with no explicit support for agent applications, no standard function or method was provided to populate the special data elements of an SNMPv1 Trap-PDU.

Naturally, as with the agent capabilities mentioned in the preceding section, customers quickly decided that, given all the SNMP operations that WinSNMP did support, the engine should also handle the generation and transmission of SNMPv1 Trap-PDUs. Therefore, several vendors implemented proprietary extensions for that purpose. These extensions resulted in application portability problems and some increased incidence of resource management errors in both implementations and applications that used these proprietary functions.

WinSNMP v2.0 decided to tackle the problem by using the method described in RFC 2089 for generating SNMPv1 Trap-PDUs when needed. Essentially, the trap sending application only constructs SNMPv2 Trap-PDUs. During the `SnmpSendMsg()` operation, the engine checks the SNMP version attribute of the `dstEntity` parameter. If it is an SNMPv1 entity, then the SNMPv2 Trap-PDU data elements are used to construct a separate SNMPv1 Trap-PDU message for that particular target entity.

Note that the submitted SNMPv2 Trap-PDU is not itself modified; a stand-alone SNMPv1 Trap-PDU message is constructed out of it for any SNMPv1 target. This permits a combination of SNMPv1 and SNMPv2 targets to be included in a trap dispatching loop on `SnmpSendMsg()`. All clean-up of the SNMPv1 Trap-PDU message resources is handled automatically by the engine, with no intervention by the application – which must, however, clean-up any resources it allocated for the SNMPv2 Trap-PDU, as usual.

This scheme has proven quite effective in the field thus far, but does come with two caveats:

1. The standard SNMPv2 Trap-PDU format does not map all of the data elements of the SNMPv1 Trap-PDU format. Therefore, some "information" is lost and some feel that the loss of the SNMPv1 Trap-PDU *agent_address* field is the most critical. The SNMPv3 Working Group is actively studying this problem at this time. This is not a WinSNMP problem; it is a generic SNMP problem.
2. The SNMPv2 Trap-PDU must be constructed in strict accordance with the relevant SNMPv2 RFCs (RFC 1905 and RFC 1907). In addition, WinSNMP mandates the presence of the (quasi-optional) `snmpTrapEnterprise` varbind, in addition to the required `sysUpTime` and `snmpTrapOID` and all positional requirements for these three varbinds must be met. WinSNMP does not make it hard to properly construct a valid SNMPv2 Trap-PDU, but it does not contain any special functions that could make it easier to get it right. (Luckily, this is a problem that typically does not recur once a programmer gets it right one time.)

Required Retransmission/Timeout Behavior

While WinSNMP v1.1a did include entity attributes (`policyTimeoutInterval` and `policyRetryCount`) designed to support a retransmission policy specific to the entity, it did not require implementations to support actual execution of that policy. The rationale was that, in the end, the application must react, in a manner appropriate to its own design and purpose, to any eventual timeout notification that would be returned by the engine and, therefore, most applications would just go ahead and handle response timeout monitoring on their own. It eventually became apparent that this decision had several flaws:

- Since implementations could elect to support retransmission policy execution, some did and some did not, thereby obstructing portability of applications.
- Since the feature was optional, no standard method had been established for returning timeout notifications to the applications for those implementations that did support it.
- As it turned out, the majority of applications developers wanted to have the implementation undertake basic retransmission policy execution on the (fairly safe) bet that this would hide most of the "in the noise" retries from their application's mainline execution stream.

As a result, several implementations opted to provide retransmission support and several different and slightly incompatible timeout notification methods were fielded. This situation presented some difficulties for applications developers. Work arounds were identified, but they were not optimal. The group targeted a fix to this problem as one of the primary goals of WinSNMP v2.0 and the API now includes a single standard method of returning timeout notifications.

Essentially, the method works identically for any session which will send SNMP request messages with the `RetransmitMode` set to `SNMPAPI_ON`. The engine's current default value for this setting is returned by the `SnmpStartup()` function and may be modified at any time (and any number of times) by the application via the `SnmpSetRetransmitMode()` function. While most applications work with a single initial value for this setting (typically `SNMPAPI_ON`), it is important to note that it is evaluated by `SnmpSendMsg()` at the *message* level. Any out-bound SNMP request message submitted when `RetransmitMode` is on will receive automatic retransmission and timeout processing by the engine, using the `timeout` and `retry` policy values the `dstEntity` held when the message was sent. The engine applies default values for these attributes when an entity is created. Subsequently, an entity's settings may be changed, as response behavior may indicate, with the `SnmpSetTimeout()` and `SnmpSetRetry()` functions.

Any SNMP request submitted when `RetransmitMode` is set to `SNMPAPI_OFF` will not be retransmitted (that is, its `retry` value is ignored) and will be silently discarded (that is, no timeout notification will be posted to the application) when its initial `timeout` expires prior to receipt of the corresponding Response-PDU. If a corresponding Response-PDU arrives after that point, it too is silently discarded by the engine. For SNMP request messages submitted when `RetransmitMode` is set to `SNMPAPI_ON`, the engine will automatically retransmit the request after the expiration of the `timeout` interval until the `retry` value has been exhausted. That is, it will make a total of `retry + 1` attempts to elicit a corresponding Response-PDU from the agent. Note that the `retry` value may be zero. If no corresponding Response-PDU has been received after the final `timeout` interval has expired, a timeout notification is returned to the application. Whether the session's notification method is of the "window/message" or "callback function" type does not matter – `wParam` will be `SNMPAPI_TL_TIMEOUT` and `lParam` will be the RequestID of the timed out PDU. The engine then frees all internal resources associated with the expired message.

Note that WinSNMP engines may record the actual `timeout` and `retry` values associated with an entity, but doing so is optional (and does not seem to be widely supported or needed to date). Applications may retrieve all current retransmission settings for an entity, both policy and actual, via the `SnmpGetTimeout()` and `SnmpGetRetry()` functions.

SNMP Version Support

Work on the original WinSNMP API specification started during the early years of the "party-based SNMPv2" effort. A valiant collective design effort attempted to provide the straightfoward support for SNMPv1 management applications that was manifestly doable, while at the same time incorporating transparent support for the still embryonic "SNMPv2" of the time. The concept of a `TranslateMode` setting was adopted to help realize this objective.

When set to an untranslated mode (either SNMPv1 or SNMPv2), the engine expects "raw" (native) values as inputs to the *entity* and *context* creation functions – namely, `SnmpStrToEntity()` and `SnmpStrToContext()`. When set to translated mode, the engine interprets the input values as "friendly names" for more complex structures and consults the implementation-specific "local configuration database" (LCD) for complete initial (default) entity or context attribute information. Beyond that point, the curent value of the `TranslateMode` setting – which may be changed at any time from its startup default via the `SnmpSetTranslateMode()` function – has no bearing on other WinSNMP operations or functions (other than the corollary functions that output the entity or context "name," `SnmpEntityToStr()` and `SnmpContextToStr()`, respectively).

Having the untranslated modes allows both direct user input of the raw values (e.g., an IP address in dotted decimal notation) and/or for an application to use its own concept of an LCD to perform whatever translations may be required. This mode is widely used. Using "translated" mode allows WinSNMP engine vendors to do a good job of also being WinSNMP application vendors and permits new or experimental addressing and/or protocol constructs to be tested fairly easily. At least one WinSNMP implementation used this mode to field early support for the "User-based Security" (USEC) model (RFC 1910). Some of this work may find its way into the eventual support for SNMPv3 (RFC 2272 and RFC 2274) in that implementation.

The format of the LCD itself is not specified, so it is implementation-specific. Furthermore, an inadequate set of "database functions" exist in WinSNMP to support a viable standard format. Both deficiencies have imposed limits on applications portability and may be addressed in a future version of the API. Such standardization may actually be a prerequisite to providing usable support for SNMPv3 as currently defined in RFC 2271 - RFC 2275.

Also, WinSNMP uses the concept of an "SNMP *level*" – a value returned by the engine via the `SnmpStartup()` function – to indicate the engine's SNMP version capabilities. Four such (cumulative) levels were originally envisioned:

- Level 0: All operations up to and including encoding and decoding actual SNMP messages, but no transport or network layer operations (i.e., no communications with agents). This is useful when the application might want to use some non-standard or otherwise unsupported transport layer interface and yet still use WinSNMP for the ASN.1/BER operations, for example.
- Level 1: Full support for transactions with SNMPv1 entities.
- Level 2: Full support for transactions with SNMPv2 (now intended only as SNMPv2c) entities.
- Level 3: Support for the "Mid-Level Manager" constructs as originally intended by the (now historic) "party-based SNMPv2" effort.

This level concept has been somewhat modified for WinSNMP v2.0. Basically, all WinSNMP v2.0 implementations must be at "Level 2," nothing lower (but recall that the levels are cumulative). Also, the original meaning of "Level 3" has been dropped, and it is anticipated that this setting will be used to indicate "Full support for SNMPv3" when work on WinSNMP v3.0 is completed (targeted for the end of this year or early in 1999). Yes, Virginia, there will be an untranslated SNMPv3 mode!

"Full support for SNMPv2" means compliance with all mandatory aspects of RFC 1901 through RFC 1908, including all of SMIv2 (e.g., Counter64). Note that when an InformRequest-PDU is received and there is at least one session (there may be multiple) which has registered to receive it (via the same `SnmpRegister()` function used to register for traps), the engine itself is responsible for generating the single Response-PDU required and for transmitting it back to the entity which sent it the InformRequest-PDU. This happens before the registered sessions are notified of the InformRequest-PDU and is totally transparent to them.

Conclusion

The WinSNMP effort is alive and well. Deployment of engines and applications for Win32 environments is now wide-spread and will become pervasive with future releases of Windows NT. A large and growing number of the major network managment platform and application vendors now utilize WinSNMP in their Win32 products. Several UNIX implementations are gaining in popularity, albeit still constitute only a small minority of SNMP engines on those platforms. The expected growth in manageable entities – especially software – that will be fostered by AgentX implementations will also further the use of WinSNMP in certain environments. WinSNMP has proven to be a stable, long-term effort with a fairly high degree of open industry participation. If you would like to play a part or just listen in and are not yet subscribed to the mailing list (which also handles HP's SNMP++ and Microsoft's SNMP software), send an electronic mail message to listserv@mailbag.intel.com with `subscribe winsnmp` in the body.

Questions Answered

David T. Perkins, SNMPinfo and Desktalk Systems

This column will continue the tradition found in *The Simple Times* by providing answers to questions about current topics. For this issue, all of the questions concern TRAPs.

What are TRAPs?

TRAPs are operations that asynchronously report the occurrence of an event. The event should be important, such as a serious error, an important state change, or a critical threshold being crossed. In the SNMPv1 framework, an event is called a *trap*. The SNMPv1 protocol uses the TRAP operation to report events. (RFC 1157 defines the protocol operations for SNMPv1.) In SNNPv2 and SNMPv3 frameworks, an event is called a *notification*. The SNMPv3 protocol contains the TRAP and INFORM operations to report events. (The SNMPv2 and SNMPv3 protocols use the second version of the protocol operations, which are defined in RFC 1905.) The difference between TRAP and INFORM operations is that TRAPs are not confirmed and INFORMs are. That is, for a TRAP operation, a single message is sent from typically an SNMP agent to an SNMP manager. For an INFORM operation, a message is sent from typically an SNMP agent to an SNMP manager, and a response is returned to the sender. If no response is received by the sender in a configured amount of time, then the sender retries the INFORM operation until a response is received or a configured number of retries is reached. (Note that the SNMPv3 architecture, as specified in RFC 2271, does not use the terms agent and manager. In it, a notification is sent by a *notification originator* and received by a *notification receiver*.) The MIB modules SNMP-TARGET-MIB and SNMP-NOTIFICATION-MIB contained in RFC 2273 define object types that specify the time-out and retry values for each *target* (a notification receiver), select the set of targets for each type of event, and specify if a TRAP or INFORM should be sent. Prior to RFC 2273 there were no general purpose MIB modules that defined object types to specify event targets (which are also called trap destinations). Thus, most device vendors have created proprietary object type definitions to specify trap destinations. Few vendors have created mechanisms to control which trap type is to be sent to each destination.

How are Event Reports Used?

The SNMP model of management is based on *smart* managers and agents with *limited intelligence*. That is, managers have a global view of the network and, thus, should be in the position to decide what management information is needed from the devices that they manage. Therefore, management applications must periodically retrieve a small set of management information from managed devices to track the status of the devices. When a manager notices that the information indicates a situation of interest, such as an abnormal increase in errors on a device, then the manager will retrieve additional information to determine the situation. After processing the information the manager may make changes to the configuration of the device, or notify a network operator of the situation and suggest human intervention, such as replacing a failed device or component. This model has a problem in that the status of devices is only updated each polling interval. The amount of time between when an event occurs and when a manager notices it is the *event detection latency*. The polling interval that a manager uses is the maximum length of time that may pass between the time an event occurs and when the manager detects the event. Agents use event reports to shorten the event detection latency. An event can be used by a manager to immediately retrieve management information from a device instead of waiting until the next scheduled poll of information from an agent. This strategy of managing an agent is called *trap directed polling*. Instead of using event reports, a manager could shorten the polling interval. This results in much higher network traffic for management, slows down the devices being managed (since they are spending more time responding to management requests), and reduces the number of devices that can be managed by a single management station.

What are the Problems with Event Reports?

The most serious problems with networks are communication failures. When such a failure occurs, an event report is least likely to be received by a management station. And, unfortunately, event reports may cause even greater problems or lengthen the period of communication failures. Failures may be caused by physical failure of network interfaces, devices, or the connecting wires (cables); misconfigured routing or forwarding tables; or network congestion (too much network traffic). When only event reports are used, a manager will never know of a problem if an event report can not reach the manager. An unanswered poll will not tell a manager what problem occurred, but will alert the manager that there is one or more problems. The manager can then start a systematic determination of the network problems.

What Types of Event Reports are Available in Each SNMP Protocol?

The SNMPv1 protocol has support for the v1 TRAP operation, which is simply named *trap*. All the variants of the SNMPv2 protocol including SNMPv2c, SNMPv2p, SNMPv2u, and SNMPv2* have support for the v2 TRAP operation, which is named *snmpV2-trap*. (See Volume 5, Number 1 of *The Simple Times* for a description of the versions of SNMP.) The SNMPv3 protocol has support for the v2 TRAP and the INFORM operations for event reports. (Note that the SNMPv2 protocols use the INFORM operation for manager-to-manager communication instead of event reporting.)

How are Event Types Defined?

The events types are defined by the TRAP-TYPE construct in MIB modules written in the SMIv1 format.

The syntax and semantics are specified in RFC 1215. MIB modules written in the SMIv2 format use the NOTIFICATION-TYPE construct, which is defined in RFC 1902. Many people now use the term *notification* instead of *event report* or *trap*. The TRAP-TYPE and NOTIFICATION-TYPE constructs are similar. Both require that a descriptor be assigned to the event type that is a name for people to refer to an event type. Also, both allow an optional list of variables to be specified, whose values will be returned in the event report message. Both have clauses so that the event type can be described and a reference given to another document that contains additional information. The NOTIFICATION-TYPE construct has a STATUS clause, not found with the TRAP-TYPE construct that specifies the status of a defined event type. Finally, both have clauses that identify an event type in a protocol message. Here are examples in SMIv1 and SMIv2:

```
ex34Event TRAP-TYPE
    ENTERPRISE exEvents
    VARIABLES  { exObj1, exObj2 }
    DESCRIPTION
        "A description goes here."
    REFERENCE
        "A reference to another document."
    ::= 34

ex34Event NOTIFICATION-TYPE
    OBJECTS  { exObj1, exObj2 }
    STATUS   current
    DESCRIPTION
        "A description goes here."
    REFERENCE
        "A reference to another document."
    ::= { exEvents 0 34 }
```

What is the Difference Between v1 and v2 TRAPs?

Events reported by the v1 TRAP message are identified by the value of three fields, which are an ASN.1 object identifier (OID) value and two nonnegative integers. Events reported by the v2 TRAP and INFORM messages are identified by a single ASN.1 OID value. The v1 TRAP messages have a format different than other SNMPv1 protocol messages. The v2 TRAP and INFORM messages have a format that is identical to other SNMPv3 protocol messages. In addition to identifying the event, all three messages contain the time (in relative time format) that the event occurred and a list of variables with values that provide information about the event. The v1 TRAP message also contains the IPv4 address of the device where the event occurred. This address is problematic, since the SNMP protocol may use transports other than UDP over IPv4.

What are Generic and Enterprise Specific Traps?

SNMPv1 has six events numbered zero through five. These are the SNMPv1 *generic* traps. They are identified by the value of the *generic-trap* field in v1 TRAP messages. No new generic traps may be defined. All other events are SNMPv1 *specific* traps. They are identified by the OID value of the *enterprise* field and the nonnegative integer value of the *specific-trap* field in v1 TRAP messages. This identification scheme is unusual and confusing. It is this way for historical reasons.

What are Reverse Mappable Traps?

The major difference between an event defined in SMIv1 and an event defined in SMIv2 is how they are identified. There is an OID value in SMIv2 for each SMIv1 generic trap. For the SMIv1 specific traps, there is a mapping algorithm to an OID value for a SMIv2 notification, which is specified in section 2.2 of RFC 1908. The algorithm creates an OID value by starting with the OID value of the enterprise field and adding two sub-identifiers. The first of the two has a value of zero and second has the value of the specific-trap field. The resulting OID value is used to identify a notification in SMIv2. Given the OID value for an SMIv2 notification, if the next to last sub-identifier has value zero, then the reverse of the mapping algorithm can be used to specify the identification of an SMIv1 trap. SMIv2 has no requirement that the OID values identifying notifications have zero as the next to last sub-identifier. If such OID values are used to identify notifications, then a mapping to a SMIv1 trap can not performed by a proxy without additional information.

How is the Community String Used in v1 TRAPs?

There has been much confusion as to how community strings are used in SNMPv1 TRAP messages. The SNMPv1 protocol document, RFC 1157, is more confusing than helpful. The true usage and meaning of community strings in all SNMP messages has always been confusing. This is because the community string in an SNMP message is simultaneously used to specify the target of the operation, specify the context of the operation, and to specify authentication information.

In SNMPv1 messages, the community string is the index into a conceptual table on an agent that has the value of these fields. There has been no standards track document that defined this conceptual table or provided a MIB module containing definitions for such a table. However, several vendors in their agent implementations have such a table. Note that there is ongoing work by the SNMPv3 working group to create standards track documents that define objects for this conceptual mapping table. A manager must know the contents of the conceptual table on an agent to know what community string values can be used in requests to an agent and how to interpret the community string value in a TRAP message. When a manager receives a TRAP message, the community string is an index into a conceptual table. The table indicates the source of the TRAP, the context of the TRAP, and authentication information. Because of the confusion as to how to use the community string value, all of the capabilities that it provides have not been fully used. First, there are few situations where a context is needed. A context is similar to an additional index for MIB objects, and is used when a MIB designer did not anticipate that another level of indexing was needed. For example, the bridge MIB, RFC 1493, assumes that an SNMP agent manages a single bridge. However, when a single agent manages more than one, the value of the context is used to specify which bridge the objects in an SNMP operation apply. A context is needed for GET and SET operations as well as the TRAP operations.

What UDP Port is Used to Send and Receive Event Reports?

UDP port 162 is the recommended port specified in the transport mapping document (RFC 1906) for notification receivers to listen for v2 TRAP and NOTIFICATION messages. A notification generator may use any available UDP port as the source, since the source port has no significance. Thus, a dynamic port, not 162, is used as the source. Likewise, UDP port 162 is specified by the SNMPv1 protocol document (RFC 1157) for managers to receive v1 TRAP messages. An SNMPv1 agent may use any available UDP port as the source.

What is this Lost Address in Event Reports?

One of the differences between v1 and v2 TRAP messages is that the v2 TRAP messages do not contain the source network address of the system where the event occurred. (And we noted that the address in the v1 TRAPs is problematic, since only an IPv4 address may be specified and the SNMPv1 protocol may run over other transports.) Not having the address is not a problem when a proxy is not used. The event receiver can obtain the source network address from the message delivery service, which will be the network address of the system where the event occurred. However, in proxy or third party event reporting situations, the network address that the event receiver obtains from the message delivery service will be for the proxy or third party event reporter and not the system where the event occurred. Since there is no standard MIB to define the objects in the conceptual table that a proxy or third party event reporter uses to know what community string value to use and which managers to forward an event report, an event receiver has no place to look to determine the original system where the event occurred! The problem exists when the event originator sends a v1 TRAP and the transport is not UDP over IPv4, when the proxy uses a v2 TRAP, or when the proxy does not use the UDP over IPv4 transport.

Note that the original community string value may also be lost when using a proxy. A solution that has been proposed, but is not yet finalized is for the first proxy to add the lost information as varBinds to the varBind list of the forwarded event report. The information would consist of three new object types, which are the value of the original community string, the transport domain of the event originator, and the transport address of the event originator.

What are Common Implementation Problems?

Some of the common implementation problems are:

- Misunderstanding how to identify v1 TRAPs:
  The names of the three fields used to identify v1 TRAPs has confused many people. Even though the field is called enterprise, this does not imply that the trap is contained in a vendor specific MIB module. The value for the enterprise field is just an OID value.

- Missing instance identifier on returned variables:
  The TRAP and INFORM messages contain a varBind list like that found on all other SNMP messages. A varBind is a pairing of a variable and a value. A variable is the identification of an instance of management information. The mistake that some vendors have made is to only specify the identification of an object type. That is, the instance information is not specified.

- Poor event definition design:
  The varBind list contained in an event report message is a list of variables and associated values that describe the event. There is no need to design an event so that any of the variables that are returned are indices of tables. A highly visible example is the linkUp event, which has object ifIndex as a required variable. Requiring ifIndex is poor event design.

Instead, the variable should have been ifOperStatus and possibly ifType. The interface is determined by the instance of the ifOperStatus variable.

- Problems with extra varBinds:
  An agent (or proxy) may add additional varBinds to an event report message. The event definition only specifies the required varBinds. Some management applications crash and burn when additional varBinds are present in an event report. Don't let this happen to you.

Has Everything About Events and Event Reporting Been Covered?

This column has covered much material in a brief space. However, only the most important and frequently asked questions about traps have been covered. The most important issue is that the infrastructure to use traps in SNMPv1 was not fully developed in the original SNMPv1 framework. However, in developing SNMPv3, much work was put into developing a complete infrastructure, and that infrastructure with a little more work (which is in progress) can also be used in SNMPv1. So you can see, work continues in the IETF SNMPv3 working group for supporting the SNMPv1 framework. The approach taken is to keep the SNMPv1 protocol unchanged and to enhance the surrounding SNMPv1 infrastructure using technology created during development of the SNMPv3 framework. The result is a better SNMPv1 infrastructure for vendors and users that want or need to continue using the SNMPv1 approach, with an easy transition to the SNMPv3 protocol, which provides security and additional protocol operation functionality.

Editor's Comment

Jürgen Schönwälder, TU Braunschweig
Aiko Pras, University of Twente

Welcome to the first issue of *The Simple Times* in 1998. There is a lot of enthusiasm back in the SNMP world and 1998 might become the "Year of SNMPv3". The SNMPv3 specifications have been published as RFCs in January and we already have several independent implementations of SNMPv3. The new year also brings some changes to *The Simple Times* newsletter and the leadership of the IETF "Operations and Management Area". More on all this below.

SNMPv3 Interoperability Demonstrations

The SNMPv3 specifications were published as Proposed Standards in January 1998 (RFC 2271 - RFC 2275). Three months after publication of the RFCs, an ad-hoc interoperability test with four independent SNMPv3 implementations was held at the 41st IETF meeting. This test is quite encouraging as the participants did not detect any serious interoperability problems. And the next big event is just around the corner: Network+Interop in Las Vegas will have an SNMPv3 Hot Spot where several vendors will demonstrate secure SNMPv3 interoperabilty including authentication, privacy and remote administration. The list of participating vendors includes Advent Network Management, Bay Networks, BMC Software, Cisco Systems, Hewlett-Packard, IBM, Liebert Corporation, SNMP Research, Tivoli and the University of Quebec in Montreal. More implementations by other vendors and universities are announced and will show up over the year.

If you want to read the latest news about SNMPv3, visit the SNMPv3 Web page at http://www.ibr.cs.tu-bs.de/projects/snmpv3/. This Web page is actively maintained and provides links to the SNMPv3 documents, short descriptions of SNMPv3 implementations, material from presentations about SNMPv3 and interoperability reports.

Editorial Board for *The Simple Times*

The editors of *The Simple Times* formed an editorial board which will help to ensure that this newsletter provides you with useful and technically accurate information. The motivation was simple: Making more people feel responsible for *The Simple Times* makes the job a bit easier for the editors. The members of the editorial board are listed on the last page of this newsletter. We would like to thank those volunteers for their support of *The Simple Times* newsletter and the interesting discussions we already had about the articles in this issue.

This issue also continues the tradition of *The Simple Times* to provide answers to frequently asked questions. In this issue, David T. Perkins addresses several questions concerning SNMP Traps in his featured column called "Questions Answered".

We also decided to make *The Simple Times*, including all the previews issues, available in Adobe's Portable Document Format (PDF). We provided a PDF version of the last issue on *The Simple Times* Web server and we got approximately the same number of hits for the PostScript and the PDF version. So we decided that providing PDF in addition to PostScript and HTML is worth the effort.

Nomination of IETF Area Directors

Finally, we would like to announce that the "Operations and Management Area" of the IETF, which hosts all the core SNMP technology working groups such as SNMPv3, AgentX or Distributed Management (DISMAN), got two new Area Directors. Harald Alvestrand (Maxware) and Bert Wijnen (IBM Research) were selected by the nominating committee to take over the positions previously held by Mike O'Dell (UUNET) and John Curran (BBN). Harald Alvestrand has previously served as one of the Area Directors in the "Applications Area". Bert Wijnen is well known to this community as one of the driving forces behind SNMPv3 and the DPI protocol, which has had major influence on the AgentX protocol.

We like to wish the Area Directors good luck for their new position and that they find a good and pragmatic way to move Internet management technology forward, for the benefits of the whole community.

Standards Summary

Please consult the latest version of *Internet Official Protocol Standards*. As of this writing, the latest version is RFC 2200.

SNMPv1 Framework

Full Standards:

- RFC 1155 - Structure of Management Information (SMI);
- RFC 1157 - Simple Network Management Protocol (SNMP); and,
- RFC 1212 - Concise MIB definitions.

Proposed Standards:

- RFC 1418 - SNMP over OSI;
- RFC 1419 - SNMP over AppleTalk; and,
- RFC 1420 - SNMP over IPX.

Informational:

- RFC 1215 - A convention for defining traps for use with the SNMP.

SNMPv2 Framework

Draft Standards:

- RFC 1902 - SMI for SNMPv2;
- RFC 1903 - Textual Conventions for SNMPv2;
- RFC 1904 - Conformance Statements for SNMPv2;
- RFC 1905 - Protocol Operations for SNMPv2;
- RFC 1906 - Transport Mappings for SNMPv2;
- RFC 1907 - MIB for SNMPv2; and,
- RFC 1908 - Coexistence between SNMPv1 and SNMPv2.

Experimental:

- RFC 1901 - Introduction to Community-based SNMPv2;
- RFC 1909 - An Administrative Infrastructure for SNMPv2; and,
- RFC 1910 - User-based Security Model for SNMPv2.

SNMPv3 Framework

Proposed Standards:

- RFC 2271 - Architecture for Describing SNMP Management Frameworks;
- RFC 2272 - Message Processing and Dispatching;
- RFC 2273 - SNMPv3 Applications;
- RFC 2274 - User-based Security Model (USM); and,
- RFC 2275 - View-based Access Control Model (VACM).

Agent Extensibility

Proposed Standards:

- RFC 2257 - AgentX Protocol Version 1.

MIB Modules

Full Standards:

- RFC 1213 - Management Information Base (MIB-II); and,
- RFC 1643 - Ether-Like Interface Type (SMIv1).

Draft Standards:

- RFC 1493 - Bridge MIB;
- RFC 1559 - DECnet phase IV MIB;
- RFC 1657 - BGP version 4 MIB;

- RFC 1658 - Character Device MIB;
- RFC 1659 - RS-232 Interface Type MIB;
- RFC 1660 - Parallel Printer Interface Type MIB;
- RFC 1694 - SMDS Interface Protocol (SIP) Interface Type MIB;
- RFC 1724 - RIP version 2 MIB;
- RFC 1748 - IEEE 802.5 Token Ring Interface Type MIB;
- RFC 1757 - Remote Network Monitoring MIB;
- RFC 1850 - OSPF version 2 MIB; and,
- RFC 2115 - Frame Relay DTE Interface Type MIB.

Proposed Standards:

- RFC 1285 - FDDI Interface Type (SMT 6.2) MIB;
- RFC 1381 - X.25 LAPB MIB;
- RFC 1382 - X.25 PLP MIB;
- RFC 1406 - DS1/E1 Interface Type MIB;
- RFC 1407 - DS3/E3 Interface Type MIB;
- RFC 1414 - Identification MIB;
- RFC 1461 - Multiprotocol Interconnect over X.25 MIB;
- RFC 1471 - PPP Link Control Protocol (LCP) MIB;
- RFC 1472 - PPP Security Protocols MIB;
- RFC 1473 - PPP IP Network Control Protocol MIB;
- RFC 1474 - PPP Bridge Network Control Protocol MIB;
- RFC 1512 - FDDI Interface Type (SMT 7.3) MIB;
- RFC 1513 - Token Ring Extensions to RMON MIB;
- RFC 1514 - Host Resources MIB;
- RFC 1525 - Source Routing Bridge MIB;
- RFC 1567 - X.500 Directory Monitoring MIB;
- RFC 1573 - Evolution of the Interfaces Group of MIB-II;
- RFC 1595 - SONET/SDH Interface Type MIB;
- RFC 1604 - Frame Relay Service MIB;
- RFC 1611 - DNS Server MIB;
- RFC 1612 - DNS Resolver MIB;
- RFC 1628 - Uninterruptible Power Supply MIB;
- RFC 1650 - Ether-Like Interface Type (SMIv2);
- RFC 1666 - SNA NAU MIB;
- RFC 1695 - ATM MIB;
- RFC 1696 - Modem MIB;
- RFC 1697 - Relational Database Management System MIB;
- RFC 1742 - AppleTalk MIB;
- RFC 1747 - SNA DLC MIB;
- RFC 1749 - 802.5 Station Source Routing MIB;
- RFC 1759 - Printer MIB;
- RFC 2006 - Mobile IP MIB;
- RFC 2011 - SNMPv2 IP MIB;
- RFC 2012 - SNMPv2 TCP MIB;
- RFC 2013 - SNMPv2 UDP MIB;
- RFC 2020 - IEEE 802.12 Interfaces MIB;
- RFC 2021 - RMON-2 MIB;
- RFC 2024 - Data Link Switching MIB;
- RFC 2037 - Entity MIB;
- RFC 2051 - APPC MIB;
- RFC 2074 - RMON Protocol Identifier;
- RFC 2096 - IP Forwarding Table MIB;
- RFC 2108 - IEEE 802.3 Repeater MIB;
- RFC 2127 - ISDN MIB;
- RFC 2128 - Dial Control MIB;
- RFC 2155 - APPN MIB;
- RFC 2206 - Resource Reservation Protocol MIB;
- RFC 2213 - Integrated Services MIB;
- RFC 2214 - Integrated Services Guaranteed Service Extensions MIB;
- RFC 2232 - DLUR MIB;

- RFC 2233 - Interfaces Group MIB;
- RFC 2238 - High Performance Routing MIB;
- RFC 2239 - IEEE 802.3 Medium Attachment Unit (MAU) MIB;
- RFC 2248 - Network Services Monitoring MIB;
- RFC 2249 - Mail Monitoring MIB;
- RFC 2266 - IEEE 802.12 Repeater MIB,
- RFC 2287 - System-Level Managed Objects for Applications; and,
- RFC 2320 - Classical IP and ARP over ATM MIB.

Related Documents

Informational:

- RFC 1270 - SNMP communication services;
- RFC 1321 - MD5 message-digest algorithm;
- RFC 1470 - A network management tool catalog;
- RFC 2039 - Applicablity of Standard MIBs to WWW Server Management; and,
- RFC 2089 - Mapping SNMPv2 onto SNMPv1 within a bi-lingual SNMP agent.

Experimental:

- RFC 1187 - Bulk table retrieval with the SNMP;
- RFC 1224 - Techniques for managing asynchronously generated alerts;
- RFC 1238 - CLNS MIB; and,
- RFC 1592 - SNMP Distributed Program Interface (SNMP-DPI);
- RFC 1792 - TCP/IPX Connection MIB Specification; and,
- RFC 2064 - Traffic Flow Measurement: Meter MIB.

Calendar and Announcements

IETF Meetings:

- 41th Meeting of the IETF
  March 30-April 3, 1998, Los Angeles, CA, USA
- 42th Meeting of the IETF
  August 23-28, 1998, Chicago, IL, USA
- 43th Meeting of the IETF
  December 7-11, 1998, Orlando, FL, USA

Conferences and Workshops:

- IEEE Workshop on Systems Management '98
  April 22-24, 1998, Newport, Rhode Island, USA
- Enterprise Management Summit '98
  August 3-7, 1998, Santa Clara, CA, USA
- Distributed Systems Operations & Management '98
  October 26-28, 1998, Delaware, USA
- Integrated Network Management '99
  May 10-14, 1999, Boston, MA, USA

Exhibitions and Trade Shows:

- NetWorld + Interop Singapore
  March 30-April 3, 1998, Singapore
- NetWorld + Interop Las Vegas
  May 4-8, 1998, Las Vegas, USA
- NetWorld + Interop Tokio
  June 1-5, 1998, Tokio, Japan
- NetWorld + Interop London
  October 13-15, 1998, London, UK
- NetWorld + Interop Atlanta
  October 19-23, 1998, Atlanta, USA
- NetWorld + Interop Sao Paulo
  November 3-5, 1998, Sao Paulo, Brazil
- NetWorld + Interop Paris
  November 5-7, 1998, Paris, France
- NetWorld + Interop Sydney
  November 24-28, 1998, Sydney, Australia

Publication Information

Contact Information	
E-mail	`st-editorial@simple-times.org`
ISSN	1060–6068

Submissions

The Simple Times solicits high-quality articles of technology and comment. Technical articles are refereed to ensure that the content is marketing-free. By definition, commentaries reflect opinion and, as such, are reviewed only to the extent required to ensure commonly-accepted publication norms.

The Simple Times also solicits terse announcements of products and services, publications, and events. These contributions are reviewed only to the extent required to ensure commonly-accepted publication norms.

Submissions are accepted only via electronic mail, and must be formatted in HTML version 1.0. Each submission must include the author's full name, title, affiliation, postal and electronic mail addresses, telephone, and fax numbers. Note that by initiating this process, the submitting party agrees to place the contribution into the public domain.

Subscriptions

The Simple Times is available in HTML, PDF and PostScript. New issues are announced via an electronic mailing list. Send electronic mail to

```
st-request@simple-times.org
```

with

```
subscribe simple-times
```

in the body if you want to subscribe to this list. Back issues are available via *The Simple Times* Web server:

```
http://www.simple-times.org/
```