Java™ Management Extensions SNMP Manager API

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

August 1999, Draft 2.0

Contents

Preface

This specification defines the SNMP manager API that is part of the Java™ Management extensions. It is not intended to be a programming guide nor a tutorial, but rather the specification of the programming interfaces for the SNMP manager API components.

Who Should Use This Book

The primary focus of this specification is to define the SNMP manager API of the Java programming language for all actors in the software and network management field. Also, programmers who wish to build applications or implementations which use this API and conform to JMX will find this book useful as a reference guide.

Before You Read This Book

This specification assumes a working knowledge of SNMP, the Java programming language and of the development environment for the Java programming language. A working knowledge of system and network management is also helpful.

How This Book Is Organized

Chapter 1 specifies the components of the SNMP Manager API.

Related Information

The following documents are also part of the JMX specification; they should be available from the same source as the present document:

- *Java™ Management Extensions*
- *Java™ Management Extensions CIM/WBEM APIs*

The definitive specification for all Java objects and interfaces of JMX is the Javadoc™ generated for these classes. The following compressed archive files should also be available from the same source as the present document:

- `jmx_javadoc.tar.Z` (this includes the Javadoc for the SNMP manager API)
- `jmx_wbem_javadoc.tar.Z`

Additional information can be found on the JMX Internet site at:

```
http://java.sun.com/products/JavaManagement
```

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

SNMP Manager API

This API is a component of the Java Management extensions (JMX), a specification for application and network management. See “Related Information” on page vi for information about the other documents which make up the JMX specification.

The SNMP manager API is a set of Java classes that simplify the development of applications for managing SNMP agents. This chapter describes the main components of the SNMP manager API, provides usage examples and also explains how to implement authentication and encryption.

Java Packages

The JMX SNMP manager API is bundled in two packages furnished as part of the Java Management extensions:

- `javax.management.snmp.manager`
- `javax.management.snmp`

The `javax.management.snmp.manager` package contains the classes and interfaces which make up the main components and functions of the API. These are the objects which need to be implemented or instantiated for manipulating agents, parameters, sessions, variable lists, and traps within the manager.

The `javax.management.snmp` package contains all of the helper classes and interfaces. These classes let the manager handle individual variables, strings, addresses, messages, PDU packets, and other information that is often used as a parameter when calling the methods of the main components.

Main Components

The JMX SNMP manager API allows you to develop applications that manage SNMP agents. There are four main components of the SNMP manager API:

- SNMP Peer
- SNMP Parameters
- SNMP Session
- SNMP Request

A peer and its parameters describe an agent to which the manager wishes to connect. The session represents the context for opening connections between the manager and one or more agents. Each operation the manager initiates on an agent through an open session is represented by a request object.

SNMP Peer

The SNMP manager uses instances of the `SnmpPeer` class to represent remote agents. Each remote agent is represented by a single `SnmpPeer` object. An `SnmpPeer` can be instantiated with the IP address, the hostname and port of the remote agent, or with just its hostname or IP address. When no port is specified, the manager uses the default SNMP port 161. The `SnmpPeer` object holds the following information:

- IP address of the remote agent
- Port number of the remote agent
- Maximum size (in bytes) of a request packet
- Maximum number of SNMP object identifiers (OIDs) that can be encoded in a single request packet
- Number of retries allowed
- Time-out delay (in milliseconds)
- List of IP addresses of alternate hosts
- The `SnmpParameter` object associated with this peer

SNMP Parameters

The `SnmpParameter` class contains information on the SNMP read and write communities, and the SNMP version. This information is used by an `SnmpSession` object while exchanging packets with an `SnmpPeer`.

By default, the manager do not have write access and belongs to the *public* read community when accessing a peer. Each peer that requires a different read or write access must be explicitly configured to use a specific SnmpParameter object that represents its communities.

Multiple SnmpPeer objects can share the same parameter object. Changing values for an SnmpParameters object affects all SnmpPeer objects that use that object. When you change these parameters, the changes are applied to all active messages for the peer or peers which use that parameter object.

CODE EXAMPLE 1-1 shows the instantiation and configuration of an SnmpPeer object representing a remote agent using port 8085 on host summer. The manager will access this peer as part of the *public* read community and the *private* write community.

CODE EXAMPLE 1-1 Instantiating and Configuring an SnmpPeer Object

```
String hostname = String ("summer");
int port = 8085;

// Create an SnmpPeer object for representing the remote agent.
SnmpPeer agent = new SnmpPeer( hostname, port );

// Create parameters to associate with the remote agent.
// When creating the parameter object, you specify the read and write
// community to be used when querying the agent.
SnmpParameters params = new SnmpParameters("public", "private");

// The newly created parameter object must be associated with the agent.
agent.setSnmpParam( params );
```

SNMP Session

The SNMP session between the manager and its peers is controlled by an instance of the SnmpSession class. An SnmpSession object creates and manages SNMP requests to multiple peers. The SnmpSession object can be instantiated with a default peer so that all requests created without specifying a peer use the default. The default peer can also be set while the SnmpSession object is running.

The SnmpSession class provides methods to initiate all the different types of SNMP requests. Each individual request performs a specific SNMP operation and is represented by an instance of the SnmpRequest object (see “SNMP Request” on page 13).

Each SNMP manager session uses its own thread of execution and its own `Datagram Socket` object.

When sending synchronous requests (see “Synchronous Mode” on page 15), the session can send only one request at time. The manager must wait for the reply to each request before issuing another. This mode is easier to implement but can only handle peers and requests at a limited rate. You may specify the wait time for each reply in order to optimize the synchronous requests.

When using asynchronous requests (see “Asynchronous Mode” on page 16), the session does not need to wait for each reply. In this case, the session maintains the list of all active requests and responses. This lets the manager handle a greater number of requests and peers in the same session. Requests are retried if the peer does not respond within a specified time. The maximum retry limit is specified by the `SnmpPeer` objects. You may cancel an active request through its `SnmpRequest` object at any time during an asynchronous session.

Session Options

Each `SnmpSession` object contains a public field called `snmpOptions` which configures the behavior of that session. This field is an instance of the `SnmpOptions` class whose methods are used to modify its settings.

Setting these options will affect all subsequent requests; existing requests are also affected, depending on the nature of the option. The SNMP session options are:

- **Allow multiplexing**. The variable binding lists of different requests in the same session are merged when sending requests. The responses are de-multiplexed.
- **Fix erroneous PDUs.** If a response contains an error, the erroneous variable is removed from the variable binding list and the request is resent to the agent.
- **Handle oversized PDUs.** If a request is too big to be held in one SNMP PDU, it is split and sent in parts.

By default, all options are enabled in a newly created session: multiplexing is allowed, erroneous PDUs are fixed and oversized PDUs are handled.

CODE EXAMPLE 1-2 shows the instantiation and configuration of an `SnmpSession` using the `SnmpPeer` instantiated in CODE EXAMPLE 1-1 as the default peer.

CODE EXAMPLE 1-2 Instantiating and Configuring an `SnmpSession`

```
// Instantiate a session with the given name.
SnmpSession session = new SnmpSession("Manager session");

// Use the "agent" peer as the default peer.
session.setDefaultPeer( agent );

// Set the desired session options.
session.snmpOptions.setMultiplexAllowed( false );
session.snmpOptions.setPduFixedOnError( false );
```

SNMP Request

Instances of the `SnmpRequest` class enable you to manage requests sent during a session. The methods of a request object let you handle retries, time-outs, and process responses from an agent. An `SnmpRequest` is created and returned by the methods which issue requests in an `SnmpSession`.

You may specify a specific peer or use the session default peer as the destination of the request. In either case, the chosen `SnmpPeer` object determines the agent which will receive the request and controls the characteristics of the messaging between manager and agent.

An `SnmpRequest` object is created and returned when requesting one of the following SNMP operations in a session:

- `snmpGet`
- `snmpGetNext`
- `snmpSet`
- `snmpGetBulk` on a specified list of variables

The SNMP manager API also provides advanced requests based on SNMP operations. These are also represented by an `SnmpRequest` object:

- `snmpGetPoll`
- `snmpGetNextPoll`
- `snmpWalkUntil`

Request States

A request becomes active when the session successfully returns the request object for the SNMP operation. The manager can then find the status of the request by calling the `getRequestStatus()` method of the request object. The possible states of a request are defined as static final variables of the `SnmpRequest` class:

- Request aborted
- In progress
- Internal error occurred
- Request never used
- Reply received
- Result available for the request
- Time-out
- Waiting for reply
- Waiting to be sent

At any time, one or more active requests in a session can be cancelled. An active request is cancelled by calling the `cancelRequest()` method of its corresponding `SnmpRequest` object.

Variable Bindings

All requests issued during a session are called with a list of variables on which to operate. The `SnmpVarbindList` class contains and manipulates a list of `SnmpVar` objects. Each instance of the `SnmpVar` class holds information for a MIB variable and consists of:

- The corresponding OID object for the MIB variable
- A value associated with that OID instance
- The status of the `SnmpVar` which indicates whether there is an exception, as stated in the SNMP v2 specification

Operation of an SNMP Manager

The SNMP manager session can be used in synchronous mode (blocking) for simple applications or in asynchronous mode (non-blocking) in higher load situations. The operating mode is determined by parameters to methods of the session that implement requested operations. The JMX SNMP manager can also handle SNMP v1 and v2 traps, do polling of a peer, and allow you to implement security or encryption. These functions are described in the following sections.

Synchronous Mode

In synchronous mode all operations are blocking, that is, an SNMP request is sent and the request object waits for a specified time, blocking the user thread. If no response is received during the wait time, the request is considered to have failed.

All methods for issuing requests from a session object contain a parameter of type `SnmpHandler` which is a callback object used in asynchronous mode. Setting this parameter to `null` when issuing a request will cause the session to operate in synchronous mode. In this mode, the manager must then specify how long to block while waiting for the response.

The request object provides the `waitForCompletion` method for specifying this waiting period. This method is used only in synchronous mode and its only parameter is the interval, in milliseconds, to wait for the response. This method sends the request to the agent and then blocks the user's thread for the required time interval.

The user thread is notified whenever a request reaches completion, whether it has succeeded or failed. The manager must then check the status of the request to know whether or not a response was received during the wait. Invoking the `waitForCompletion` method with an interval of zero blocks the request object until a response is received. CODE EXAMPLE 1-3 shows the instantiation of a synchronous request.

CODE EXAMPLE 1-3 Instantiating a Synchronous Request

```
// Build the list of variables you want to query.
// For debug purposes, you can associate a name to your list.
SnmpVarbindList list = new SnmpVarbindList("MyManager varbind list");

// We want to read the "sysDescr" variable.
list.addVariable("sysDescr.0");

// Issue the SNMP get request in synchronous mode
SnmpRequest request = session.snmpGet( null, list );
System.out.println("SimpleManager::main: Send get request to SNMP Agent on " +
    host + " at port " + port );

// Send the request and wait (block) for ten seconds
boolean completed = request.waitForCompletion( 10000 );
```

Asynchronous Mode

In asynchronous mode, management applications are not blocked while waiting for individual responses and are thus able to issue more than one active request at a time. Asynchronous mode also permits polling, as described in “Polling” on page 18.

You send an asynchronous request by instantiating an `SnmpRequest` object that specifies a response handler. The response handler is a class that implements the `SnmpHandler` interface in order to act on the result of a request. You need to implement this class according to the behavior you wish your manager to have when responding to the request. The response handler has three callback methods, and the session invokes the appropriate one according to the result of the request:

- `processSnmpPollData` is called when the agent has responded to the requested SNMP operation; the manager should check the status of the request to find out the agent’s response
- `processSnmpPollTimeout` is called when the agent did not respond within the specified time and the allowed retries have been exhausted
- `processSnmpInternalError` is called when any internal error has occurred while processing the request.

When a request is submitted, it joins a queue. When the request is sent to an agent, it is placed in a PDU packet and a timer is started when the packet is sent. If specified in the `SnmpOptions` object, requests are multiplexed and sent in a PDU packet containing other requests. Responses are de-multiplexed automatically. All of the multiplexing operations are transparent.

Requests are automatically retried if a response does not arrive within a specified interval. If the agent responds with an error, the `SnmpRequest` object uses the options defined in the `SnmpOptions` object to determine the subsequent behavior. CODE EXAMPLE 1-4 shows the instantiation of an asynchronous request.

CODE EXAMPLE 1-4 Sending an Asynchronous Request

```
// Build the list of variables you want to query.
SnmpVarbindList list = new SnmpVarbindList();
list.addVariable("sysDescr.0");

// Instantiate a custom SNMP response handler.
MyRspHandler handler = new MyRspHandler( Thread.currentThread() );

// Make the SNMP walk request with this handler:
// read all MIB variables from "sysDescr" to "sysServices".

SnmpRequest request = session.snmpWalkUntil( handler, list, new
             SnmpOid("sysServices"));
System.out.println("MyManager::main: Start snmpWalkUntil for SNMP Agent on “ +
             host + " at port " + port);

// The responses will be received through the specified handler.
```

SNMP Traps

The SNMP manager API provides classes that enable you to receive SNMP v1 and SNMP v2 trap PDUs.

The JMX SNMP manager receives all trap PDUs through an instance of the `SnmpEventReportDispatcher` class. This class runs as a thread and receives traps for the manager and dispatches them to listener objects. Listeners are instances of a class that implements the `SnmpEventReportListener` interface. The dispatcher’s `addTrapListener` method registers a listener for receiving all traps. The listener interface defines two callback methods, one for SNMP v1 trap PDUs and the other for SNMP v2 trap PDUs.

When instantiating the `SnmpEventReportDispatcher` object, you may specify the port number on which it will receive traps. Otherwise, the dispatcher will receive traps on port 162 by default. When the dispatcher receives a trap, it determines its type of PDU, SNMP v1 or v2, and calls the appropriate callback method for each of its listener objects.

CODE EXAMPLE 1-5 shows the code for a class which implements the `SnmpEventReportListener` interface.

CODE EXAMPLE 1-5 Implementing SNMP Trap Callbacks

```
// processSnmpEventReport is called when a valid SNMP v1 trap PDU is received.
// processSnmpV2EventReport is called when a valid SNMP v2 trap PDU is received.

public class MyTrapListener implements SnmpEventReportListener {
   public void processSnmpEventReport( SnmpPduTrap trap ) {
      java.lang.System.out.println("NOTE: MyTrapListener received Trap V1:");
      java.lang.System.out.println("\tGeneric "+trap.genericTrap);
      java.lang.System.out.println("\tSpecific "+trap.specificTrap);
      java.lang.System.out.println("\tTimeStamp "+trap.timeStamp);
      java.lang.System.out.println("\tAgent address "
          +trap.agentAddr.stringValue());
   }
 public void processSnmpV2EventReport(SnmpPduRequest trapV2) {
      java.lang.System.out.println("NOTE: MyTrapListener received Trap V2:");
     ...
   }
}
```

To receive traps, the manager must first instantiate a dispatcher and then add a trap listener to it. It is the callback method of the listener which is called when a valid SNMP v1 or SNMP v2 trap PDU is processed. CODE EXAMPLE 1-6 shows the instantiation of a trap dispatcher and registration of a listener.

CODE EXAMPLE 1-6 Instantiating a Trap Listener

```
// Create a dispatcher to receive SNMP Traps on UDP port 8086
SnmpEventReportDispatcher trapDispatcher = new SnmpEventReportDispatcher(8086);

// Add the trap listener whose callbacks will process all trap PDUs
trapDispatcher.addEventReportListener( new MyTrapListener() );

new Thread( trapDispatcher ).start();
```

Polling

Using the polling feature, an SNMP manager can continually issue periodic requests to its agents in order to implement advanced management capabilities.

The `SnmpSession` class provides the polling capability through its `snmpGetPoll(..)` and `snmpGetNextPoll(..)` methods. All of these methods take a parameter specifying the polling interval in milliseconds and then perform the given SNMP operation periodically with this interval. At every polling cycle, the `SnmpHandler` callback implementation will be called with the result of each request.

The polling operation will continue indefinitely until there is an error or until it is cancelled. The polling is stopped if an error of any type is encountered during any of the individual requests. Otherwise, you must explicitly cancel the `SnmpRequest` object to terminate the polling operation (see "SNMP Request" on page 13).

For `snmpGetPoll` and `snmpGetNextPoll` operations, each polling cycle is started with the original `SnmpVarbindList` specified. This means that each poll request returns the value of the same variable or list of variables. This function is different from the `SnmpWalk` operation which advances through the variables and returns the value of a different variable each time, until it meets a specified condition.

Implementing Security

The JMX SNMP manager API provides a hook which enables authentication and encryption to be added to the SNMP manager. This hook involves the following classes in the javax.management.snmp package:

- `SnmpPduFactory` interface
- `SnmpPduPacket` class
- `SnmpMessage` class

This section describes how these classes are used so that you can write an implementation of the `SnmpPduFactory` interface which provides security to your SNMP manager.

Decoding and Encoding SNMP Packets

A manager analyzes an SNMP packet in the following steps:

- The received bytes are translated into an `SnmpMessage` object.
- The SnmpMessage object is decoded into an `SnmpPduPacket` object.
- The `SnmpPduPacket` is analyzed and the corresponding operation is performed.

The manager creates an SNMP packet as follows before sending it:

- An `SnmpPduPacket` object is initialized according to the requested operation.
- The `SnmpPduPacket` object is encoded into an `SnmpMessage`.
- The `SnmpMessage` is translated into bytes.

The SnmpPduPacket represents the fully decoded description of an SNMP request. In particular, it includes the operation type (get, set...), the list of variables to be operated upon, the request identifier, and the protocol version.

CODE EXAMPLE 1-7 The SnmpPduPacket class

```
public class SnmpPduPacket {
             public int version;
             public byte[] community;
             public int type;
             public int requestId;
             public SnmpVarBind[] varBindList;
}
```

The SnmpMessage is a partially decoded representation of the SNMP request. Only the protocol version and the community strings are decoded. All the other parameters remain encoded as a byte array. The SnmpMessage class is derived from the Message syntax in RFC 1157 and RFC 1902.

CODE EXAMPLE 1-8 The SnmpMessage class

```
public class SnmpMessage {
             public int version;
             public byte[] community;
             public byte[] data;
}
```

The SnmpPduFactory Interface

The task of translating an SnmpMessage object into an SnmpPduPacket object is delegated to an object which implements the SnmpPduFactory interface. This interface defines two methods, one for decoding packets from messages and one for encoding packets into messages.

CODE EXAMPLE 1-9 The `SnmpPduFactory` interface

```
public interface SnmpPduFactory {

            // Decode an SnmpMessage into an SnmpPduPacket
            public SnmpPduPacket decodePdu( SnmpMessage msg )
                throws SnmpStatusException;

            // Encode an SnmpMessage from an SnmpPduPacket
            public SnmpMessage encodePdu( SnmpPduPacket pdu, int maxPktSize )
                throws SnmpStatusException, SnmpTooBigException;
}
```

Implementing the `SnmpPduFactory` Interface

The methods of the `SnmpPduFactory` implementation control every incoming or outgoing SNMP packet in the manager:

- `decodePdu()` decodes the `SnmpMessage` and returns a fully initialized `SnmpPduPacket` object. If it returns `null`, the `SnmpMessage` is assumed to be unsafe and will be dropped. It throws an `SmpStatusException` if decoding failed or if the PDU contains out-of-bounds values. The `SnmpMessage` will also be dropped in this case.
- `encodePdu()` encodes the `SnmpPduPacket` and returns a fully initialized `SnmpMessage` object. If it returns null, the `SnmpPduPacket` is assumed not safe to send and the current request will be aborted. It throws an `SnmpStatusException` if the `SnmpPduPacket` contains out-of-bounds values. It throws an `SnmpTooBigException` if the `SnmpPduPacket` does not fit into the internal buffer used by the JMX implementation.

The default implementation of this interface implements the encoding and decoding by calling the corresponding methods in the `SnmpMessage` class. These methods perform the conversions according to the SNMP protocol.

Using a Different PDU Factory

By creating your own implementation of the `SnmpPduFactory` interface, you can perform all security checks your management solution requires. Since the message bodies are transferred as bytes, you can implement encryption in the encoding and decoding of messages. These methods can also verify the authenticity of packets or of the agent through digital signatures or any other resource available to the manager. The return value of each method can then be set according to the results of all security checks. This indicates to the manager whether or not a particular packet, its information, or its source can be trusted.

Note: in order to use such an encoding system, matching security mechanisms need to be implemented on all agents participating in the secure management solution. These implementations and their deployment are beyond the scope of this specification.

PDU factories are associated with `SnmpPeer` objects so that security may be set according to each agent. The customized `SnmpPduFactory` implementation should be attached to all `SnmpPeer` objects that represent agents requiring a secure connection. A peer's PDU factory can be changed using its `setPduFactory` method. CODE EXAMPLE 1-10 shows how to change the `SnmpPduFactory` of an SnmpPeer object and send a secure request.

CODE EXAMPLE 1-10 Sending a secure request

```
String hostname = String ("summer");
SnmpPeer myPeer = new SnmpPeer( hostname );

// replace the default PDU factory with one customized for my firewall
myPeer.setPduFactory( new MyFireWallPduFactory() );
SnmpRequest request = session.snmpGet( myPeer, this, myVarBindList );
```

The PDU factory of the `SnmpEventReportDispatcher` class can also be replaced in order to insure security when receiving traps. This implies that the customized implementation of `SnmpPduFactory` also handles encryption of traps and identification of trap senders. Again, this is beyond the scope of this specification.