Network Working Group
Request for Comments: 3179
Obsoletes: 2593
Category: Experimental

J. Schoenwaelder
TU Braunschweig
J. Quittek
NEC Europe Ltd.

October 2001

Script MIB Extensibility Protocol Version 1.1

Status of this Memo

Copyright Notice

Abstract

The Script MIB extensibility protocol (SMX) defined in this memo separates language specific runtime systems from language independent Script MIB implementations. The IETF Script MIB defines an interface for the delegation of management functions based on the Internet management framework. A management script is a set of instructions that are executed by a language specific runtime system.

Table of Contents

1. Introduction

The Script MIB [1] defines a standard interface for the delegation of management functions based on the Internet management framework. In particular, it provides the following capabilities:

1. Transfer of management scripts to a distributed manager.

2. Initiating, suspending, resuming and terminating management scripts.

3. Transfer of arguments for management scripts.

4. Monitoring and control of running management scripts.

5. Transfer of results produced by management scripts.

A management script is a set of instructions executed by a language specific runtime system. The Script MIB does not prescribe a specific language. Instead, it allows to control scripts written in different languages that are executing concurrently.

The Script MIB Extensibility protocol (SMX) defined in this memo can be used to separate language specific runtime systems from the runtime system independent Script MIB implementations. The lightweight SMX protocol can be used to support different runtime systems without any changes to the language neutral part of a Script MIB implementation.

Examples of languages and runtime systems considered during the design of the SMX protocol are the Java virtual machine [2] and the Tool Command Language (Tcl) [3]. Other languages with comparable features should be easy to integrate as well.

2. Process Model and Communication Model

Figure 1 shows the process and communication model underlying the SMX protocol. The language and runtime system independent SNMP agent implementing the Script MIB communicates with one ore more runtime systems via the SMX protocol. A runtime system may be able to execute one or multiple scripts simultaneously (multi-threading). The SMX protocol supports multi-threading, but it does not require multi-threaded runtime systems.

The SMX protocol uses a local storage device (usually implemented on top of the local file system) to transfer scripts from the SNMP agent to the runtime systems. The SNMP agent has read and write access to the script storage device while the runtime systems only need read access. The SMX protocol passes the location of a script in the local storage device to the runtime engines. It is then the responsibility of the runtime engines to load the script from the specified location.

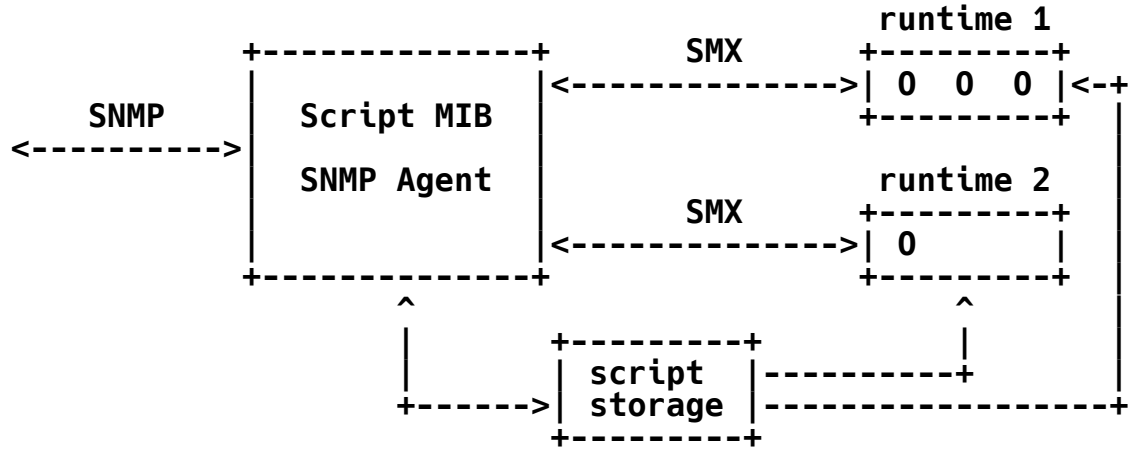

Figure 1: SMX process and communication model

3. Security Profiles

Security profiles control what a running script is allowed to do. It is useful to distinguish two different classes of security profiles:

- The operating system security profile specifies the set of operating system services that can be used by the operating system level process which executes a script. Under UNIX, this maps to the effective user and group identity for the running process. In addition, many UNIX versions allow to set other resource limits, such as the number of open files or the maximum stack sizes. Another mechanism in UNIX is the chroot() system call which changes the file system root for a process. The chroot() mechanism can be used to prevent runtime systems from accessing any system files. It is suggested to make use of all applicable operating system security mechanism in order to protect the operating system from malicious scripts or runtime systems.

- Secure runtime systems provide fine grained control over the set of services that can be used by a running script at a particular point during script execution. A runtime security profile specifying fine grained access control is runtime system dependent. For a Java virtual machine, the runtime security profile is interpreted by the SecurityManager and ClassLoader classes[4]. For Tcl, the runtime security profile maps to the interpreter's security profile [5].

The SMX protocol allows to execute scripts under different operating system profiles and runtime system profiles. Multiple operating system security profiles are realized by using multiple runtime systems which execute in operating system processes with different security profiles. Multiple runtime security profiles are supported by passing a security profile name to a runtime system during script invocation.

The Script MIB does not define how operating system or runtime system security profiles are identified. This memo suggests that the smLaunchOwner is mapped to an operating system security profile and a runtime system security profile when a script is started.

4. Start of Runtime Systems and Connection Establishment

The SNMP agent starts runtime systems based on the static properties of the runtime system (multi-threaded or single-threaded) and the operating system security profiles. Starting a new runtime system requires to create a process environment which matches the operating system security profile.

In order to prevent SMX communication from untrusted peers the SNMP agent has to choose a secure SMX transport. This memo defines two transports in Section 8: (a) a bi-directional pipe using standard input/output streams on the runtime engine side, and (b) a TCP connection where the SNMP agent acts as a listening server that accepts only connections from local runtime engines that authenticate themselves with a secret shared between the agent and the runtime engine.

5. SMX Messages

The message formats described below are defined using the Augmented BNF (ABNF) defined in RFC 2234 [6]. The definitions for `ALPHA', `DIGIT', `HEXDIG', `WSP', `CRLF', `CR', `LF', `HTAB', `VCHAR' and `DQUOTE' are imported from appendix A of RFC 2234 and not repeated here.

5.1. Common Definitions

The following ABNF definitions are used in subsequent sections to define the SMX protocol messages.

```
Zero          = %x30          ; the ASCII character '0'

ProfileChars  = DIGIT / ALPHA / %x2D-2F / %x3A / %x5F
                              ; digits, alphas, and the characters
                              ; '-', '.', '/', ':', '_'

QuotedString  = DQUOTE *(VCHAR / WSP) DQUOTE

HexString     = 1*(HEXDIG HEXDIG)

Id            = 1*DIGIT       ; identifier for an SMX transaction

Script        = QuotedString  ; script file name

RunId         = 1*DIGIT       ; globally unique identifier for a
                              ; running script (note, smRunIndex
                              ; is only unique for a smLaunchOwner,
                              ; smLaunchName pair)

Profile       = 1*ProfileChars ; security profile name

RunState      =  "1"          ; smRunState `initializing'
RunState      =/ "2"          ; smRunState `executing'
RunState      =/ "3"          ; smRunState `suspending'
RunState      =/ "4"          ; smRunState `suspended'
RunState      =/ "5"          ; smRunState `resuming'
```

```
RunState      =/ "6"          ; smRunState `aborting'
RunState      =/ "7"          ; smRunState `terminated'

ExitCode      =  "1"          ; smRunExitCode `noError'
ExitCode      =/ "2"          ; smRunExitCode `halted'
ExitCode      =/ "3"          ; smRunExitCode `lifeTimeExceeded'
ExitCode      =/ "4"          ; smRunExitCode `noResourcesLeft'
ExitCode      =/ "5"          ; smRunExitCode `languageError'
ExitCode      =/ "6"          ; smRunExitCode `runtimeError'
ExitCode      =/ "7"          ; smRunExitCode `invalidArgument'
ExitCode      =/ "8"          ; smRunExitCode `securityViolation'
ExitCode      =/ "9"          ; smRunExitCode `genericError'

Authenticator = HexString     ; authentication cookie

Version       = "SMX/1.1"     ; current version of the SMX protocol

Argument      = HexString / QuotedString      ; see smRunArgument

Result        = HexString / QuotedString      ; see smRunResult

ErrorMsg      = HexString / QuotedString      ; see smRunError
```

The definition of QuotedString requires further explanation. A quoted string may contain special character sequences, all starting with the backslash character (%x5C). The interpretation of these sequences is as follows:

`\\'	backslash character	(`%x5C')
`\t'	tab character	(`HTAB')
`\n'	newline character	(`LF')
`\r'	carriage-return character	(`CR')
`\"'	quote character	(`DQUOTE')

In all other cases not listed above, the backslash is dropped and the following character is treated as an ordinary character.

`Argument' and `Result' is either a QuotedString or a HexString. The Script MIB defines script arguments and results as arbitrary octet strings. The SMX protocol supports a binary and a human readable representation since it is likely that printable argument and result strings will be used frequently. However, an implementation must be able to handle both formats in order to be compliant with the Script MIB.

The `Authenticator' is a HexString which does not carry any semantics other than being a random sequence of bytes. It is therefore not necessary to have a human readable representation.

5.2. Commands

The following ABNF definitions define the set of SMX commands which can be sent from the SNMP agent to a runtime system.

```
Command =  "hello"   WSP Id CRLF

Command =/ "start"   WSP Id WSP RunId WSP Script WSP Profile
                     WSP Argument CRLF

Command =/ "suspend" WSP Id WSP RunId CRLF

Command =/ "resume"  WSP Id WSP RunId CRLF

Command =/ "abort"   WSP Id WSP RunId CRLF

Command =/ "status"  WSP Id WSP RunId CRLF
```

The `hello' command is always the first command sent over a SMX connection. It is used to identify and authenticate the runtime system. The `start' command starts the execution of a script. The `suspend', `resume' and `abort' commands can be used to change the status of a running script. The `status' command is used to retrieve status information for a running script.

There is no compile command. It is the responsibility of the SNMP agent to perform any compilation steps as needed before using the SMX `start' command. There is no SMX command to shutdown a runtime system. Closing the connection must be interpreted as a request to terminate all running scripts in that runtime system and to shutdown the runtime system.

5.3. Replies

Every reply message starts with a three digit reply code and ends with `CRLF'. The three digits in a reply code have a special meaning. The first digit identifies the class of a reply message. The following classes exist:

```
1yz   transient positive response
2yz   permanent positive response
3yz   transient negative response
4yz   permanent negative response
5yz   asynchronous notification
```

The classes 1yz and 3yz are currently not used by SMX version 1.1. They are defined only for future SMX extensions.

The second digit encodes the specific category. The following categories exist:

```
x0z   syntax errors that don't fit any other category
x1z   replies for commands targeted at the whole runtime system
x2z   replies for commands targeted at scripts
x3z   replies for commands targeted at running instances of scripts
```

The third digit gives a finer gradation of meaning in each category specified by the second digit. Below is the ABNF definition of all reply messages and codes:

```
   Reply =  "211" WSP Id WSP Version *1(WSP Authenticator) CRLF
                                    ; identification of the
                                    ; runtime system

   Reply =/ "231" WSP Id WSP RunState CRLF
                                    ; status of a running script

   Reply =/ "232" WSP Id CRLF    ; abort of a running script

   Reply =/ "401" WSP Id CRLF    ; syntax error in command

   Reply =/ "402" WSP Id CRLF    ; unknown command

   Reply =/ "421" WSP Id CRLF    ; unknown or illegal Script

   Reply =/ "431" WSP Id CRLF    ; unknown or illegal RunId

   Reply =/ "432" WSP Id CRLF    ; unknown or illegal Profile

   Reply =/ "433" WSP Id CRLF    ; illegal Argument

   Reply =/ "434" WSP Id CRLF    ; unable to change the status of
                                    ; a running script

   Reply =/ "511" WSP Zero WSP QuotedString CRLF
                                    ; an arbitrary message send from
                                    ; the runtime system

   Reply =/ "531" WSP Zero WSP RunId WSP RunState CRLF
                                    ; asynchronous running script
                                    ; status change

   Reply =/ "532" WSP Zero WSP RunId WSP RunState WSP Result CRLF
                                    ; intermediate script result
```

```
      Reply =/ "533" WSP Zero WSP RunId WSP RunState WSP Result CRLF
                                    ; intermediate script result that
                                    ; triggers an event report

      Reply =/ "534" WSP Zero WSP RunId WSP Result CRLF
                                    ; normal script termination,
                                    ; deprecated

      Reply =/ "535" WSP Zero WSP RunId WSP ExitCode WSP ErrorMsg CRLF
                                    ; abnormal script termination,
                                    ; deprecated

      Reply =/ "536" WSP Zero WSP RunId WSP RunState WSP ErrorMsg CRLF
                                    ; script error

      Reply =/ "537" WSP Zero WSP RunId WSP RunState WSP ErrorMsg CRLF
                                    ; script error that
                                    ; triggers an event report

      Reply =/ "538" WSP Zero WSP RunId WSP ExitCode CRLF
                                    ; script termination
```

6. Elements of Procedure

This section describes in detail the processing steps performed by the SNMP agent and the runtime system with regard to the SMX protocol.

6.1. SMX Message Processing on the Runtime Systems

This section describes the processing of SMX command messages by a runtime engine and the conditions under which asynchronous notifications are generated.

When the runtime system receives a message, it first tries to recognize a command consisting of the command string and the transaction identifier. If the runtime system is not able to extract both the command string and the transaction identifier, then the message is discarded. An asynchronous `511' reply may be generated in this case. Otherwise, the command string is checked to be valid, i.e. to be one of the strings `hello', `start', `suspend', `resume', `abort', or `status'. If the string is invalid, a `402' reply is sent and processing of the message stops. If a valid command has been detected, further processing of the message depends on the command as described below.

The command specific processing describes several possible syntax errors for which specific reply messages are generated. If the runtime engine detects any syntax error which is not explicitly mentioned or which cannot be identified uniquely, a generic `401' reply is sent indicating that the command cannot be executed.

6.1.1. Processing the `hello' Command

When the runtime system receives a `hello' command, it processes it as follows:

1. The runtime system sends a `211' reply. If the runtime system has access to a shared secret, then the reply must contain the optional `Authenticator', which is a function of the shared secret.

6.1.2. Processing the `start' Command

When the runtime system receives a `start' command, it processes it as follows:

1. The syntax of the arguments of the `start' command is checked. The following four checks must be made:

   (a) The syntax of the `RunId' parameter is checked and a `431' reply is sent if any syntax error is detected.

   (b) The syntax of the `Script' parameter is checked and a `421' reply is sent if any syntax error is detected.

   (c) The syntax of the `Profile' parameter is checked and a `432' reply is sent if any syntax error is detected.

   (d) If syntax of the `Argument' parameter is checked and a `433' reply is sent if any syntax error is detected.

2. The runtime system checks whether the new `RunId' is already in use. If yes, a `431' reply is sent and processing stops.

3. The runtime system checks whether the `Script' parameter is the name of a file on the local storage device, that can be read. A `421' reply is sent and processing stops if the file does not exist or is not readable.

4. The runtime system checks whether the security profile is known and sends a `432' reply and stops processing if not.

5. The runtime engine starts the script given by the script name.

When the script has been started, a `231' reply is sent including the current run state.

Processing of the `start' command stops, when the script reaches the state `running'. For each asynchronous state change of the running script, a `531' reply is sent. Processing of the `start' command is also stopped if an error occurs before the state `running' is reached. In this case, the run is aborted and a `538' reply is generated. An optional `536' reply can be send before the `538' reply to report an error message.

If an `abort' command or a `suspend' command for the running script is received before processing of the `start' command is complete, then the processing of the `start' command may be stopped before the state `running' is reached. In this case, the resulting status of the running script is given by the respective reply to the `abort' or `suspend' command, and no reply with the transaction identifier of the `start' command is generated.

6.1.3. Processing the `suspend' Command

When the runtime system receives a `suspend' command, it processes it as follows:

1. If there is a syntax error in the running script identifier or if there is no running script matching the identifier, a `431' reply is sent and processing of the command is stopped.

2. If the running script is already in the state `suspended', a `231' reply is sent and processing of the command is stopped.

3. If the running script is in the state `running', it is suspended and a `231' reply is sent after suspending. If suspending fails, a `434' reply is sent and processing of the command is stopped.

4. If the running script has not yet reached the state `running' (the `start' command still being processed), it may reach the state `suspended' without having been in the state `running'. After reaching the state `suspended', a `231' reply is sent.

5. If the running script is in any other state, a `434' reply is sent.

6.1.4. Processing the `resume' Command

When the runtime system receives a `resume' command, it processes it as follows:

1. If there is a syntax error in the running script identifier or if there is no running script matching the identifier, a `431' reply is sent and processing of the command is stopped.

2. If the running script is already in the state `running', a `231' reply is sent and processing of the command is stopped.

3. If the running script is in the state `suspended', it is resumed and a `231' reply is sent after resuming. If resuming fails, a `434' reply is sent and processing of the command is stopped.

4. If the `start' command is still being processed for the script, a `231' reply is sent when the state `running' has been reached.

5. If the running script is in any other state, a `434' reply is sent.

6.1.5. Processing the `abort' Command

When the runtime system receives an `abort' command, it processes it as follows:

1. If there is a syntax error in the running script identifier or if there is no running script matching the identifier, a `431' reply is sent and processing of the command is stopped.

2. If the running script is already aborted, a `232' reply is sent and processing of the command is stopped.

3. The running script is aborted and a `232' reply is sent after aborting. If aborting fails, a `434' reply is sent and processing is stopped.

6.1.6. Processing the `status' Command

When the runtime system receives a `status' command, it processes it as follows:

1. If there is a syntax error in the running script identifier or if there is no running script matching the identifier, a `431' reply is sent and processing of the command is stopped.

2. The status of the script is obtained and a `231' reply is sent.

6.1.7. Generation of Asynchronous Notifications

The runtime system generates or may generate the following notifications:

1. If a change of the status of a running script is observed by the runtime system, a `531' reply is sent.

2. A `534' reply is sent if a running script terminates normally. This reply is deprecated. You can emulate this reply with a combination of a `532' reply and a `538' reply.

3. A `535' reply is sent if a running script terminates abnormally. This reply is deprecated. You can emulate this reply with a combination of a `536' reply and a `538' reply.

4. A `532' reply is sent if a script generates an intermediate result.

5. A `533' reply is sent if a script generates an intermediate result which causes the generation of a `smScriptResult' notification.

6. A `536' reply is sent if a running script produces an error. If the error is fatal, the script execution will be terminated and a 538 reply will follow. Otherwise, if the error is non-fatal, the script continues execution.

7. A `537' reply is sent if a running script produces an error which should cause the generation of a `smScriptException' notification. If the error is fatal, the script execution will be terminated and a 538 reply will follow. Otherwise, if the error is non-fatal, the script continues execution.

8. A `538' reply is sent if a running script terminates. The ExitCode is used to distinguish between normal termination (`noError') or abnormal termination.

9. Besides the notifications mentioned above, the runtime system may generate arbitrary `511' replies, which are logged or displayed by the SNMP agent.

6.2. SMX Message Processing on the SNMP Agent

This section describes the conditions under which an SNMP agent implementing the Script MIB generates SMX commands. It also describes how the SNMP agent processes replies to SMX commands.

6.2.1. Creating a Runtime System

New runtime systems are started by the SNMP agent while processing set requests for a `smLaunchStart' variable. The SNMP agent first searches for an already running runtime systems which matches the security profiles associated with the `smLaunchStart' variable. If no suitable runtime system is available, a new runtime system is started by either

(a) starting the executable for the runtime system in a new process which conforms to the operating system security profile, and establishing a bi-directional pipe to the runtime systems standard input/output streams to be used for SMX transport, or

(b) preparing the environment for the new runtime system and starting the executable for the runtime system in a new process which conforms to the operating system security profile. The SNMP agent prepares to accept a connection from the new runtime system.

The `smRunState' of all scripts that should be executed in the new runtime system is set to `initializing'.

6.2.2. Generating the `hello' Command

The `hello' command is generated once an SMX connection is established. The SNMP agent sends the `hello' command as defined in section 5.2. The SNMP agent then expects a reply from the runtime system within a reasonable timeout interval.

1. If the timeout expires before the SNMP agent received a reply, then the connection is closed and all data associated with it is deleted. Any scripts that should be running in this runtime system are aborted, the `smRunExitCode' is set to `genericError' and `smRunError' is modified to describe the error situation.

2. If the received message can not be analyzed because it does not have the required format, then the connection is closed and all data associated with it is deleted. Any scripts that should be running in this runtime system are aborted, the `smRunExitCode' is set to `genericError' and `smRunError' is modified to describe the error situation.

3. If the received message is a `211' reply, then the `Id' is checked whether it matches the `Id' used in the `hello' command. If the `Id' matches, then the `Version' is checked. If the `Version' matches a supported SMX protocol version, then, if present, the `Authenticator' is checked. If any of the tests fails or if the

SNMP agent requires an authenticator and it did not receive a matching `Authenticator' with the `211' reply, then the connection is closed and all data associated with this runtime system is deleted. Any scripts that should be running in this runtime system are aborted, the `smRunExitCode' is set to `genericError' and `smRunError' is modified to describe the error situation.

4. Received messages are discarded if none of the previous rules applies.

6.2.3. Generating the `start' Command

The `start' command is generated while processing set-requests for a `smLaunchStart' variable. The `start' command assumes that the SNMP agent already determined a runtime system suitable to execute the script associated with the `smLaunchStart' variable. The SNMP agent sends the `start' command as defined in section 5.2 to the selected runtime system. The SNMP agent then expects a reply from the runtime system within a reasonable timeout interval.

1. If the timeout expires before the SNMP agent received a reply, then the SNMP agent sends an `abort' command to abort the running script and sets the `RunState' of the running script to `terminated', the `smRunExitCode' to `genericError' and `smRunError' is modified to describe the timeout situation.

2. If the received message can not be analyzed because it does not have the required format, then the message is ignored. The SNMP agent continues to wait for a valid reply message until the timeout expires.

3. If the received message is a `4yz' reply and the `Id' matches the `Id' of the `start' command, then the SNMP agent assumes that the script can not be started. The `smRunState' of the running script is set to `terminated', the `smRunExitCode' to `genericError' and the `smRunError' is modified to contain a message describing the error situation.

4. If the received message is a `231' reply and the `Id' matches the `Id' of the `start' command, then the `smRunState' variable of the running script is updated.

5. Received messages are discarded if none of the previous rules applies.

6.2.4. Generating the `suspend' Command

The `suspend' command is generated while processing set-requests for the `smLaunchControl' and `smRunControl' variables which change the value to `suspend'. The SNMP agent sets the `smRunState' variable to `suspending' and sends the `suspend' command as defined in section 5.2. The SNMP agent then expects a reply from the runtime system within a reasonable timeout interval.

1. If the timeout expires before the SNMP agent received a reply, then the SNMP agent sends an `abort' command to abort the running script and sets the `smRunState' of the running script to `terminated', the `smRunExitCode' to `genericError' and `smRunError' is modified to describe the timeout situation.

2. If the received message can not be analyzed because it does not have the required format, then the message is ignored. The SNMP agent continues to wait for a valid reply message until the timeout expires.

3. If the received message is a `401', `402' or a `431' reply and the `Id' matches the `Id' of the `suspend' command, then the runtime systems is assumed to not provide the suspend/resume capability and processing of the `suspend' command stops.

4. If the received message is a `231' reply and the `Id' matches the `Id' of the `suspend' command, then the `smRunState' variable of the running script is updated.

5. Received messages are discarded if none of the previous rules applies.

6.2.5. Generating the `resume' Command

The `resume' command is generated while processing set-requests for the `smLaunchControl' and `smRunControl' variables which change the value to `resume'. The SNMP agent sets the `smRunState' variable to `resuming' and sends the `resume' command as defined in section 5.2. The SNMP agent then expects a reply from the runtime system within a reasonable timeout interval.

1. If the timeout expires before the SNMP agent received a reply, then the SNMP agent sends an `abort' command to abort the running script and sets the `smRunState' of the running script to `terminated', the `smRunExitCode' to `genericError' and `smRunError' is modified to describe the timeout situation.

2. If the received message can not be analyzed because it does not have the required format, then the message is ignored. The SNMP agent continues to wait for a valid reply message until the timeout expires.

3. If the received message is a `401', `402' or a `431' reply and the `Id' matches the `Id' of the `resume' command, then the runtime systems is assumed to not provide the suspend/resume capability and processing of the `resume' command stops.

4. If the received message is a `231' reply and the `Id' matches the `Id' of the `resume' command, then the `smRunState' variable of the running script is updated.

5. Received messages are discarded if none of the previous rules applies.

6.2.6. Generating the `abort' Command

The `abort' command is generated while processing set-requests for the `smLaunchControl' and `smRunControl' variables which change the value to `abort'. In addition, the `abort' command is also generated if the `smRunLifeTime' variable reaches the value 0. The SNMP agent sends the `abort' command as defined in section 5.2. The SNMP agent then expects a reply from the runtime system within a reasonable timeout interval.

1. If the timeout expires before the SNMP agent received a reply, then the SNMP agent sets the `smRunState' of the running script to `terminated', the `smRunExitCode' to `genericError' and `smRunError' is modified to describe the timeout situation.

2. If the received message can not be analyzed because it does not have the required format, then the message is ignored. The SNMP agent continues to wait for a valid reply message until the timeout expires.

3. If the received message is a `4yz' reply and the `Id' matches the `Id' of the `abort' command, then the SNMP agent assumes that the script can not be aborted. The `smRunState' of the running script is set to `terminated', the `smRunExitCode' to `genericError' and the `smRunResult' is modified to describe the error situation.

4. If the received message is a `232' reply and the `Id' matches the `Id' of the `abort' command, then the `smRunExitCode' variable of the terminated script is changed to either `halted' (when processing a set-request for the `smLaunchControl' and `smRunControl' variables) or `lifeTimeExceeded' (if the `abort'

command was generated because the `smRunLifeTime' variable reached the value 0). The `smRunState' variable is changed to the value `terminated'.

5. Received messages are discarded if none of the previous rules applies.

6.2.7. Generating the `status' Command

The `status' command is generated either periodically or on demand by the SNMP agent in order to retrieve status information from running scripts. The SNMP agent sends the `status' command as defined in 5.2. The SNMP agent then expects a reply from the runtime system within a reasonable timeout interval.

1. If the timeout expires before the SNMP agent received a reply, then the SNMP agent sends an `abort' command to abort the running script and sets the `smRunState' of the running script to `terminated', the `smRunExitCode' to `genericError' and `smRunError' is modified to describe the timeout situation.

2. If the received message can not be analyzed because it does not have the required format, then the message is ignored. The SNMP agent continues to wait for a valid reply message until the timeout expires.

3. If the received message is a `4yz' reply and the `Id' matches the `Id' of the `status' command, then the SNMP agent assumes that the script status can not be read, which is a fatal error condition. The SNMP agent sends an `abort' command to abort the running script. The `smRunState' of the running script is set to `terminated', the `smRunExitCode' to `genericError' and the `smRunError' is modified to describe the error situation.

4. If the received message is a `231' reply and the `Id' matches the `Id' of the `status' command, then the `smRunState' variable of the running script is updated.

5. Received messages are discarded if none of the previous rules applies.

6.2.8. Processing Asynchronous Notifications

The runtime system can send asynchronous status change notifications. These `5yz' replies are processed as described below.

1. If the received message is a `511' reply, then the message is displayed or logged appropriately and processing stops.

2. If the received message is a `531' reply, then the SNMP agent checks whether a running script with the given `RunId' exists in the runtime system. Processing of the notification stops if there is no running script with the `RunId'. Otherwise, the `smRunState' is updated.

3. If the received message is a `532' reply, then the SNMP agent checks whether a running script with the given `RunId' exists in the runtime system. Processing of the notification stops if there is no running script with the `RunId'. Otherwise, `smRunState' and `smRunResult' are updated.

4. If the received message is a `533' reply, then the SNMP agent checks whether a running script with the given `RunId' exists in the runtime system. Processing of the notification stops if there is no running script with the `RunId'. Otherwise, `smRunState' and `smRunResult' are updated and the `smScriptResult' notification is generated.

5. If the received message is a `534' reply, then the SNMP agent checks whether a running script with the given `RunId' exists in the runtime system. Processing stops if there is no running script with the `RunId'. Otherwise, `smExitCode' is set to `noError', `smRunState' is set to `terminated' and `smRunResult' is updated.

6. If the received message is a `535' reply, then the SNMP agent checks whether a running script with the given `RunId' exists in the runtime system. Processing stops if there is no running script with the `RunId'. Otherwise, `smRunState' is set to `terminated' and `smExitCode' and `smRunError' are updated.

7. If the received message is a `536' reply, then the SNMP agent checks whether a running script with the given `RunId' exists in the runtime system. Processing of the notification stops if there is no running script with the `RunId'. Otherwise, `smRunState' and `smRunError' are updated.

8. If the received message is a `537' reply, then the SNMP agent checks whether a running script with the given `RunId' exists in the runtime system. Processing of the notification stops if there is no running script with the `RunId'. Otherwise, `smRunState' and `smRunError' are updated and the `smScriptException' notification is generated.

9. If the received message is a `538' reply, then the SNMP agent checks whether a running script with the given `RunId' exists in the runtime system. Processing of the notification stops if there is no running script with the `RunId'. Otherwise, `smRunState' is set to `terminated' and the `smExitCode' is updated.

7. Example SMX Message Flow

Below is an example SMX message exchange. Messages sent from the SNMP agent are marked with `>' while replies sent from the runtime system are marked with `<'. Line terminators (`CRLF') are not shown in order to make the example more readable.

```
> hello 1
< 211 1 SMX/1.1 0AF0BAED6F877FBC
> start 2 42 "/var/snmp/scripts/foo.jar" untrusted ""
> start 5 44 "/var/snmp/scripts/bar.jar" trusted "www.ietf.org"
< 231 2 2
> start 12 48 "/var/snmp/scripts/foo.jar" funny ""
< 231 5 2
< 532 0 44 2 "waiting for response"
> status 18 42
> status 19 44
< 432 12
< 231 19 2
< 231 18 2
> hello 578
< 211 578 SMX/1.1 0AF0BAED6F877FBC
> suspend 581 42
< 231 581 4
< 532 0 44 7 "test completed"
< 538 0 44 1
> abort 611 42
< 232 611
```

8. Transport Mappings

In order to prevent SMX communication from untrusted peers the SNMP agent has to choose a secure SMX transport. This memo defines two transports in Section 8: (a) a bi-directional pipe using standard input/output streams on the runtime engine side, and (b) a TCP

connection where the SNMP agent acts as a listening server that accepts only connections from local runtime engines that authenticate themselves with a secret shared between the agent and the runtime engine.

For simplicity and security reasons the transport over bi-directional pipes is the preferred transport.

Further transports (e.g., UNIX domain sockets) are possible but not defined at this point in time. The reason for choosing pipes and TCP connections as the transport for SMX was that these IPC mechanisms are supported by most potential runtime systems, while other transports are not universally available.

8.1. SMX over Bi-directional Pipes

The SNMP agent first creates a bi-directional pipe. Then the agent creates the runtime system process with its standard input and standard output streams connected to the pipe. Further authentication mechanisms are not required.

8.2. SMX over TCP

The SNMP agent first creates a listening TCP socket which accepts connections from runtime systems. Then the agent creates the runtime system process. It is then the responsibility of the runtime system to establish a connection to the agent's TCP socket once it has been started. The SNMP agent must ensure that only authorized runtime systems establish a connection to the listening TCP socket. The following rules are used for this purpose:

- The TCP connection must originate from the local host.

- The SNMP agent must check the `Authenticator' in the `211' reply if authentication is required and it must close the TCP connection if no valid response is received within a given time interval.

9. Security Considerations

The SMX protocol as specified in this memo runs over a bi-directional pipe or over a local TCP connection between the agent and the runtime system. Protocol messages never leave the local system. It is therefore not possible to attack the message exchanges if the underlying operating system protects bi-directional pipes and local TCP connections from other users on the same machine.

The transport over a bi-directional pipe specifies that the pipe is created and connected to the standard input/output stream of the runtime engine by the agent before the runtime engine is started. It is therefore not possible that an unauthorized process can exchange SMX messages over the bi-directional pipe.

In case of the TCP transport, the only critical situation is the connection establishment phase. The rules defined in section 8 ensure that only local connections are accepted and that a runtime system has to authenticate itself with an authenticator if the agent requires authentication. It is strongly suggested that agents require authentication, especially on multiuser systems.

The SMX 1.0 specification in RFC 2593 suggested a scheme where the authenticator was passed to the runtime engines as part of the process environment. This scheme relies on the protection of process environments by the operating system against unauthorized access. Some operating systems allow users to read the process environment of arbitrary processes. Hence the scheme proposed in RFC 2593 is considered unsecure on these operating systems. This memo does not dictate the mechanism by which the runtime obtains the shares secret. It is the responsibility of implementors or administrators to select a mechanism which is secure on the target platforms.

The SMX protocol assumes a local script storage area which is used to pass script code from the SNMP agent to the runtime systems. The SMX protocol passes file names from the agent to the runtime engines. It is necessary that the script files in the local script storage area are properly protected so that only the SNMP agent has write access. Failure to properly protect write access to the local script storage area can allow attackers to execute arbitrary code in runtime systems that might have special privileges.

The SMX protocol allows to execute script under different operating system and runtime system security profiles. The memo suggests to map the smLaunchOwner value to an operating system and a runtime system security profile. The operating system security profile is enforced by the operating system by setting up a proper process environment. The runtime security profile is enforced by a secure runtime system (e.g., the Java virtual machine or a safe Tcl interpreter) [7].

10. Changes from RFC 2593

The following non-editorial changes have been made:

1. Added the `536' and `537' replies which may be generated asynchronously by runtime engines to report error conditions.

2. Added the `538' reply which can be used to signal the (normal or abnormal) termination of a running script. This new reply replaces the `534' and `535' replies, which are now deprecated.

3. Relaxed the rules for ProfileChars to also include the characters ':' and '_', which are frequently used in namespaces and identifiers.

4. Changed the SMX protocol version number from 1.0 to 1.1.

5. Added a second (and preferred) transport over a bi-directional pipe due to security risks when a shared secret is passed through an operating system's environment variable.

6. Made the `Authenticator' in the `211' reply optional.

11. Acknowledgments

The protocol described in this memo is the result of a joint project between the Technical University of Braunschweig and C&C Research Laboratories of NEC Europe Ltd. in Heidelberg. The authors like to thank Matthias Bolz, Cornelia Kappler, Andreas Kind, Sven Mertens, Jan Nicklisch, and Frank Strauss for their contributions to the design and the implementation of the protocol described in this memo. The authors also like to thank David Wallis for pointing out a security risk in SMX 1.0 with passing a cookie via an operating system environment variable.

12. References

[1] Levi, D. and J. Schoenwaelder, "Definitions of Managed Objects for the Delegation of Management Scripts", RFC 3165, September 2001.

[2] Lindholm, T., and F. Yellin, "The Java Virtual Machine Specification", Addison Wesley, 1997.

[3] J.K. Ousterhout, "Tcl and the Tk Toolkit", Addison Wesley, 1994.

[4] Fritzinger, J.S., and M. Mueller, "Java Security", White Paper, Sun Microsystems, Inc., 1996.

[5] Levy, J.Y., Demailly, L., Ousterhout, J.K., and B. Welch, "The Safe-Tcl Security Model", Proc. USENIX Annual Technical Conference, June 1998.

[6] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[7] Schoenwaelder, J., and J. Quittek, "Secure Internet Management by Delegation", Computer Networks 35(1), January 2001.

13. Authors' Addresses

Juergen Schoenwaelder
TU Braunschweig
Bueltenweg 74/75
38106 Braunschweig
Germany

Phone: +49 531 391-3283
EMail: schoenw@ibr.cs.tu-bs.de

Juergen Quittek
NEC Europe Ltd.
C&C Research Laboratories
Adenauerplatz 6
69115 Heidelberg
Germany

Phone: +49 6221 90511-15
EMail: quittek@ccrle.nec.de

14. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.