NETCONF Interoperability Testing

Iyad Tumar, Ha Manh Tran, Jürgen Schönwälder
Computer Science, Jacobs University Bremen, Germany
{i.tumar, h.tran, j.schoenwaelder}@jacobs-university.de

Abstract

The IETF has developed a network configuration management protocol called NETCONF which was published as proposed standard in 2006. The NETCONF protocol provides mechanisms to install, manipulate, and delete the configuration of network devices by using an Extensible Markup Language (XML) based data encoding on top of a simple Remote Procedure Call (RPC) layer.

This report describes a NETCONF interoperability testing plan that is used to test whether NETCONF protocol implementations meet the NETCONF protocol specification. The test of three independent NETCONF implementations reveals bugs in several NETCONF implementations. While constructing test cases, a few shortcomings of the specifications were identified as well.

1 Introduction

The NETCONF protocol specified in RFC 4741 [1] defines a mechanism to configure and manage network devices. It allows clients to retrieve configuration from network devices or to add new configuration to these devices. The NETCONF protocol uses a remote procedure call (RPC) paradigm. A client encodes an RPC request in XML [2] and sends it to a server using a secure, connection-oriented session. The server returns with an RPC-REPLY response encoded in XML.

The NETCONF protocol supports many features required for configuration management that were lacking in other network management protocols, like for example SNMP [3]. NETCONF operates on so called datastores and represents the configuration of a device as a structured document. The protocol distinguishes between running configurations, startup configurations and candidate configurations. In addition, it provides primitives to assist with the coordination of concurrent configuration change requests and to support distributed configuration change transactions over several devices. Finally, NETCONF provides filtering mechanisms, validation capabilities, and event notification support [4].

The aim of this report is twofold. First, we describe a NETCONF interoperability testing plan that is used to test whether the NETCONF protocol implementations meet the NETCONF protocol specification in RFC 4741. Second, we will discuss the observations and results that show how the test plan found some NETCONF implementation bugs, and how it revealed a few shortcomings where the specification (RFC 4741 and RFC 4742) is either somewhat ambiguous or totally silent.

In order to make the paper concise and precise, we use the word request when we refer to an `rpc` request message and the word response when we refer to an `rpc-reply` response message. We refer to NETCONF operations such as `get-config` by typesetting the operation name in teletype font. The names of test suites are typeset in small caps, e.g., VACM.

The rest of the paper is structured as follows: An overview of the NETCONF protocol is presented in Section 2. Section 3 provides information about the systems under test before the test plan is introduced in Section 4. The NETCONF interoperability tool (NOT) is described in Section 5. Preliminary observations are reported in Section 6 before the paper concludes in Section 7.

2 NETCONF Overview

The NETCONF protocol [1] uses a simple remote procedure call (RPC) layer running over secure transports to facilitate communication between a client and a server. The Secure Shell (SSH) [5] is the mandatory secure transport that all NETCONF clients and servers are required to implement as a means of promoting interoperability [6].

The NETCONF protocol can be partitioned into four layers as shown in Figure 1. The transport protocol layer provides a secure communication path between the client and server. The RPC layer provides a mechanism for encoding RPCs. The operations layer residing on top of the RPC layer defines a set of base operations invoked as RPC methods with XML-encoded parameters to manipulate configuration state. The configuration data itself forms the content

layer residing above the operations layer.

The NETCONF protocol supports multiple configuration datastores. A configuration datastore is defined as the set of configuration objects required to get a device from its initial default state into a desired operational state. The `running` datastore is present in the base model and provides the currently active configuration. In addition, NETCONF supports a `candidate` datastore, which is a buffer that can be manipulated and later committed to the `running` datastore, and a `startup` configuration datastore, which is loaded by the device as part of initialization when it reboots or reloads [4].

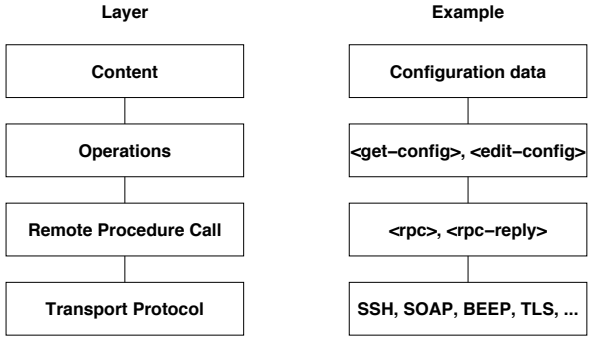

Figure 1. NETCONF protocol layers [1].

Figure 2 shows the protocol operations that have been defined so far by the NETCONF working group of the IETF. The first two operations `get-config` and `edit-config` can be used to read and manipulate the content of a datastore. The `get-config` operation can be used to read all or parts of a specified configuration. The `edit-config` operation modifies all or part of a specified configuration datastore. Special attributes embedded in the config parameter control which parts of the configuration is created, deleted, replaced or merged. The test-option and the error-option parameters control the validation and the handling of errors. The `copy-config` operation creates or replaces an entire configuration datastore with the contents of another complete configuration datastore and the `delete-config` operation deletes a configuration datastore (the `running` configuration datastore cannot be deleted).

The `lock` and `unlock` operations do coarse grain locking of a complete datastore and locks are intended to be short lived. More fine grained locking mechanisms are currently being defined in the IETF [4]. The `get` operation can be used to retrieve the running configuration and the current operational state of a device.

NETCONF sessions can be terminated using the `close-session` and `kill-session` operations. The

Operation	Arguments
get-config	source [filter]
edit-config	target [default-operation] [test-option] [error-option] config
copy-config	target source
delete-config	target
lock	target
unlock	target
get	[filter]
close-session	
kill-session	session-id
discard-changes	
validate	source
commit	[confirmed confirm-timeout]
create-subscription	[stream] [filter] [start] [stop]

Figure 2. NETCONF protocol operations (arguments in brackets are optional) [4]

`close-session` operation initiates a graceful close of the current session while the `kill-session` operation forces the termination of another session.

The optional `discard-changes` operation clears the candidate configuration datastore by copying the running configuration into the candidate buffer while the optional `validate` operation runs validation checks on a datastore. The optional `commit` operation is used to commit the configuration in the candidate datastore to the running datastore.

A separate specification published as RFC 5277 [7] extends the base NETCONF operations defined in RFC 4741 for notification handling. This is done by adding the `create-subscription` operation and introducing new `notification` messages carrying notification content. By using a notification stream abstraction, it is possible to receive live notifications as well as replay recorded notifications.

NETCONF protocol introduces the notion of capabilities. A capability is some functionality that supplements the base NETCONF specification. A capability is identified by a uniform resource identifier (URI). The base capabilities are defined using URNs following the method described in RFC 3553 [8]. NETCONF peers exchange device capabilities when the session is initiated: When the NETCONF session is opened, each peer (both client and server) must send a `hello` message containing a list of that peer's capabilities. This list must include the NETCONF `:base` capability[1]. Following RFC 4741, we denote capabilities by the capability name prefixed with a colon, omitting the rest of the URI.

[1] `urn:ietf:params:netconf:base:1.0`

3 Systems Under Test

The systems used for the NETCONF interoperability testing comprise Cisco 1802 integrated services routers, Juniper J6300 routers, and the Tail-f ConfD software for configuration management. The ConfD software is an extensible development toolkit that allows users to add new components by writing a configuration specification for a data model and loading the generated object and schema files for the components. For the sake of consistency, we refer to the ConfD software as the Tail-f system. Table 1 briefly describes the three platform and the SSH support of the three systems. The ConfD is installed and configured to run on a Linux XEN virtual machine [9].

System	Platform	SSH Support
Juniper	JUNOS ver. 9.0	ver. 1.5/2.0
Tail-f	ConfD ver. 2.5.2	ver. 2.0
Cisco	IOS ver. 12.4	ver. 2.0

Table 1. Systems under test

Table 2 presents the NETCONF capabilities announced by the systems under test. The Tail-f system supports all capabilities except the `:startup` capability. The Cisco and Juniper systems support fewer capabilities and apparently the Cisco implementation favours a distinct `startup` datastore while the Juniper implementation favours a `candidate` datastore with commit and rollback support. In addition to the capabilities listed in Table 2, each system announces several proprietary capabilities.

Capability	Juniper	Tail-f	Cisco
`:base`	√	√	√
`:writable-running`		√	√
`:candidate`	√	√	
`:confirmed-commit`	√	√	
`:rollback-on-error`		√	
`:validate`	√	√	
`:startup`			√
`:url`	√	√	√
`:xpath`		√	

Table 2. NETCONF capabilities supported by the systems under test

The Tail-f and Juniper implementations use an event driven parser. They do not wait for the framing character sequence to respond to a request. The Cisco system does not seem to have the event driven parser or at least it does not start processing requests until the framing character sequence has been received.

The Juniper implementation is very lenient. For example, it continues processing requests even if the client does not send a `hello` message or the client does not provide suitable XML namespace and message-id attributes. The Juniper implementation supports a large number of vendor-specific operations. In addition, it renders the returned XML content in a tree-structure that is relatively easy to read and it generates XML comments in cases of fatal errors before closing the connection. As a consequence, the Juniper implementation is very easy to get use for users who like to learn how things work without using tools other than a scratch pad and a cut and paste device.

The Tail-f and Cisco implementations are much less tolerant when processing input not closely following RFC 4741. They also return XML data in a compact encoding, minimizing the embedded white-space and thus reducing message sizes. Without proper tools, it is pretty difficult for humans to read the responses. In some cases, these two implementations close the connection when the client sends illegal input without an indication of the reason for closing the connection. In such cases, it can take some effort to investigate the wrongdoings.

Finally, we like to point out that the Cisco implementation does not support structured content; i.e., its configuration content is a block of proprietary IOS commands wrapped in an XML element. As a consequence, several of the advanced NETCONF features for retrieving and modifying structured configuration data cannot be applied.

4 Test Plan

In this section we describe our NETCONF test plan. To make the execution of the tests efficient and to keep the collection of tests organized, we divided our test plan into five test suites. A test suite is a collection of test cases that are intended to be used to test and verify whether the systems under test meet the NETCONF protocol specification contained in RFC 4741 [1] and RFC 4742 [6].

Table 3 lists the test suites and the number of test cases in each suite. The total number of test cases is currently 87. Our organization of test cases into test suites is not directly following the vertical layering model show in Figure 1 and the horizontal organization of operations and capabilities in the operations layer as one might expect. The reason is essentially our attempt to reduce the overhead during the execution of the test suite on the systems under test. This led to a more tightly integrated organization of the test cases.

The most basic test suite is the GENERAL test suite. It includes test cases for general operations such as `lock`, `unlock`, `close-session`, `kill-session`, `discard-changes`, `validate`, and `commit`. To test the behavior of the system under test, a client sends `lock` requests and checks the reaction of the server. For exam-

Test Suite	Number of Test Cases
GENERAL	19
GET	11
GET-CONFIG	16
EDIT-CONFIG	15
VACM	26

Table 3. Test Suites

ple, a test case might send a `lock` requests to an already locked datastore and then verify that the server reacts with a proper error message. The GENERAL test suite also tests the general format of requests and responses. For example, test cases check whether response messages contain the `message-id` attribute and that it matches the value contained in the request message.

The second test suite is the GET suite. It contains a collection of test cases that are intended to be used to test the filter mechanism of the `get` operation. For example, a test case checks whether the systems under test returns the entire content of the contents of the entire running configuration data plus the operational state when no filter is used. The third suite is the GET-CONFIG suite. It contains test cases related to the filter mechanism of the `get-config` operation.

The fourth suite is the EDIT-CONFIG suite. It includes test cases for the `edit-config`, `copy-config`, and the `delete-config` operations. Several of the test cases contained in the EDIT-CONFIG suite are data model specific and we had to implement several tests in different ways due to a lack of common data models. This extra work can be reduced if implementers volunteer to implement a common data model. A proposal for such a data model, a YANG version of the `SNMP-VIEW-BASED-ACM-MIB`, is contained in the appendix of this paper.

The last test suite is the `vacm` suite. It includes a collection of test cases to test the NETCONF protocol operations against the VACM data model (see appendix).

5 Test Tool (NOT)

We have implemented a tool called NOT (NETCONF interOperability Testing tool) to automatically execute the test suites against a system under test. Our NOT tool basically performs the following operations:

- connecting to a system under test using the SSH
- verifying the initial `hello` message
- executing test cases by
  - sending a test request and receiving a response
  - verifying both the request and the response following the criteria defined by RFC 4741 [1].
- reporting the failure or the success of each test

The tool is equipped with an XML parser to analyze the responses for verification; i.e., the parser, upon receiving a response, provides a list of elements with quantity, a list of attributes with quantity, a list of attribute values and a list of text parts. With this information, the tool can detect possible flaws from the responses, such as whether any element is missing or any error is returned. The following example shows the information of a response without errors or warnings:

```
---ELEMENT TYPES
  rpc-reply 1
         ok 1
---ATTRIBUTE TYPES
 message-id 1
      xmlns 1
---ATTRIBUTE VALUES
 message-id [u'1007']
      xmlns [u'urn:ietf:params:xml:\
               ns:netconf:base:1.0']
---TEXT PARTS
            []
```

We have used the Python unit testing framework [10]. The framework features test automation, shared configuration of setup and shutdown methods, arrangement of tests into collections, and independent reporting of the tests. The tool takes advantage of these features to maintain a single connection for all tests and to group related tests into a collection; e.g., tests concerned with creation, modification and deletion operations are grouped together to re-use and clean the testing environment easily. The tool organizes test cases into several collections of test cases, namely test suites, that have been discussed in Section 4.

While the tool has been used successfully to test some specific devices (see next section), it possesses several limitations. Firstly, it lacks a resumption mechanism to continue the test run when it encounters connection loss due to the misbehavior of systems under test. Secondly, while the test cases comply with RFC 4741, the test scripts, i.e., the piece of code that implements test cases, depends on the specification and configuration of components of the tested systems to produce the requests and to verify the responses. Finally, the framework requires some extra work for complicated test cases; e.g., testing the `lock` operation requires an extra session to lock the database.

6 Preliminary Observations

We have used the NOT tool to test the systems described in Section 3. Since the result of the tests are specific to the different NETCONF implementations, we present the results by refering to system X and we leave out the mapping of X to the systems described in Section 3. Note that we did manually re-check the failed test cases in order to erase bugs in the test scripts.

System	Success	Failure	Irrelevant
A	47.2%	14.9%	37.9%
B	82.8%	9.2%	8.0%
C	17.3%	10.3%	72.4%

Table 4. Test result summary organized by the systems under test

Table 4 presents the result of the NOT tool for the systems under test. The "success" and "failure" columns indicate the percent of passed and failed test cases respectively, while the "irrelevant" column indicates the percent of test cases that cannot be applied to a specific system due to either system configuration or implementation problems (e.g., the `vacm` data model is not implemented).

We learned that the systems A and B comply reasonably well with the RFCs. The system A fails 14.9% of the test cases and most of them are related to the basic format of request and response messages or the filter mechanism of the `get` operation. The system B performs better with very few failed test cases and most of them are concerned with the validation of XML elements in request messages. The two systems A and B have very few problems with the filter mechanism of the `get-config` operation or the usage of the `edit-config` operation for creating, modifying and deleting configuration elements. The system C performs poorer with 57.1% irrelevant test cases and 17.9% failed test cases. The failed test cases are related to the format of requests and responses or the filter mechanism of the `get` operation.

Test Suite	Success	Failure	Irrelevant
GENERAL	75.4%	10.5%	14.1%
GET	39.4%	51.5%	9.1%
GET-CONFIG	66.7%	0%	33.3%
EDIT-CONFIG	44.5%	2.2%	53.3%
VACM	25.6%	7.7%	66.7%

Table 5. Test result summary organized by the test suites

Table 5 reports the passed and failed test cases organized by the test suites over the total number of running test cases for the systems under test. There are two remarks: (i) the GET suite obtains a high percentage of failed test cases, and (ii) the GET-CONFIG and EDIT-CONFIG suites obtain low percents of failed test cases 6.1%. We found that the majority of failed test cases from the GET suite is related to the filter mechanism of the `get` operation.

With the failed test cases in mind, we have looked back into the RFCs. There are several things where the RFC is either somewhat ambiguous or totally silent. In general, the RFC should provide more detailed descriptions for error situations and it might be necessary to better constrain the currently open ended format of request and response messages since they for example allow arbitrary values for attributes. Furthermore, the RFC should be updated with clearer examples. Some particular issues are listed below:

- The RFC ignores the XML declaration

  ```
  <?xml version$="1.0" encoding="UTF-8"?>
  ```

  for requests and responses. Some systems do not execute a request without this declaration while other systems do. It seems that the IETF working group favours to have a mandatory XML declaration.

- The examples in RFC 4741 often omit namespace declarations for request and response messages. Only few systems execute a request without a proper namespace declaration and it would help interoperability if the examples would contain namespace declarations where necessary.

- RFC 4741 requires that additional attributes present in the `<rpc>` element of a request message must be returned in the `<rpc-reply>` element of the response message without any change (see section 4.1 of the RFC 4741). This requirement leads to the problems when such an attribute conflicts with attributes generated by the implementation. One implementation generated duplicated attributes (and thus invalid XML) while another implementation removes a duplicated attribute resulting in violation of RFC 4741.

- RFC 4741 allows arbitrary strings for the `message-id` attribute. From the tests, we found that implementations terminate the session often without an error indication or return strange results when the `message-id` attribute in a request message contains unexpected content such as the literal string `]]>]]>` or the literal string `</rpc>`. Of course, a proper NETCONF client would not generate such messages since they also invalid XML. But on the other hand, one can question whether arbitrary content in request and response attributes is a feature worth to support.

Some of the items listed above are meanwhile actively discussed on the NETCONF working group mailing list and work is planned to revise RFC 4741 in order to fix bugs and to clarify the processing of NETCONF messages.

7 Conclusion

We have carried out some work on NETCONF interoperability testing. This work aims at observing the compliance of NETCONF implementations with RFC 4741. It also aims at identifying inconsistencies in the RFC. We have proposed a test plan consisting of five test suites. Each test suite contains a number of test cases that involve a single operation or a group of related operations. The test cases exploit several aspects of RFC 4741 including the format of request and response messages, the filter mechanism supported by some operations, NETCONF capabilities, and so on. The test cases have been coded into the NOT tool, which automates the execution of test runs.

We have used the NOT tool to test three different NETCONF implementations. Our preliminary observations indicate that the number of failed test cases is relatively high for some systems, thus raising the question of the compliance of these systems with RFC 4741. We have also noted some inconsistencies in RFC 4741 that should be addressed in a future revision of this document.

While some interesting initial results have been obtained, this work still requires several improvements. First, the coverage of RFC 4741 by the test cases needs to be evaluated and increased by adding additional test cases as needed. Furthermore, it would be nice to reduce the dependency of the test cases on different data models. Third, the NOT tool should be improved to better support more complicated test cases that involve multiple NETCONF sessions. Fourth, it would be nice to have a tool able to generate test suites out of YANG data models. And finally, it would be valuable to repeat the tests with a larger number of different NETCONF implementations and to evaluate how test results impact future software revisions and lead to more interoperability.

Acknowledgment

The work reported in this paper is supported by the EC IST-EMANICS Network of Excellence (#26854).

References

[1] R. Enns. NETCONF Configuration Protocol. RFC 4741, Juniper Networks, December 2006.

[2] C. Sperberg-McQueen, J. Paoli, E. Maler, and T. Bray. *Extensible Markup Language (XML) 1.0*. CERT, Second edition, October 2000.

[3] J. Case, R. Mundy, D. Partain, and B. Stewart. Introduction and Applicability Statements for Internet Standard Management Framework. RFC 3410, SNMP Research, Network Associates Laboratories, Ericsson, December 2002.

[4] J. Schönwälder, M. Björklund, and P. Shafer. Network Configuration Management using NETCONF and YANG. *(under review)*, 2008.

[5] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251, SSH Communications Security Corp, Cisco Systems, December 2006.

[6] M. Wasserman and T. Goddard. Using the NETCONF Configuration Protocol over Secure Shell (SSH). RFC 4742, ICEsoft Technologies, Inc., December 2006.

[7] S. Chisholm and H. Trevino. NETCONF Event Notifications. RFC 5277, Nortel, Cisco, July 2008.

[8] M. Mealling. An IETF URN Sub-namespace for Registered Protocol Parameters. RFC 3553, SSH Communications Security Corp, Cisco Systems, June 2003.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[10] Python unit testing framework. http://pyunit.sourceforge.net/. Last access in November 2008.

A SNMP Yang Module

```
module snmp {

  /* $Id: snmp.yang 3001 2008-10-14 14:56:23Z schoenw $ */

  /*
   * Q1: What to do about permanent or readonly table entries?
   */

  namespace "urn:ietf:params:xml:ns:yang:snmp";
  prefix "snmp";

  include "snmp-common";
  include "snmp-vacm";

  organization
   "IETF NETMOD (NETCONF Data Modeling Language) Working Group";

  contact
   "Editor:   Juergen Schoenwaelder
              <mailto:j.schoenwaelder@jacobs-university.de>";

  description
   "This module contains a collection of YANG definitions for
    configuring SNMP engines via NETCONF.

    Copyright (C) The IETF Trust (2008).  This version of this
    YANG module is part of RFC XXXX; see the RFC itself for full
    legal notices.";
  // RFC Ed.: replace XXXX with actual RFC number and remove this note

  revision 2008-10-11 {
    description
      "Initial revision, published as RFC XXXX.";
  }
  // RFC Ed.: replace XXXX with actual RFC number and remove this note

}
```

B SNMP Common Yang Submodule

```
submodule snmp-common {

  /* $Id: snmp-common.yang 3016 2008-11-03 08:56:59Z mbj@tail-f.com $ */

  belongs-to snmp {
    prefix snmp;
  }

  organization
   "IETF NETMOD (NETCONF Data Modeling Language) Working Group";

  contact
   "Editor:   Juergen Schoenwaelder
              <mailto:j.schoenwaelder@jacobs-university.de>";
```

```
description
 "This submodule contains a collection of common YANG definitions
  for configuring SNMP engines via NETCONF.

  Copyright (C) The IETF Trust (2008).  This version of this
  YANG module is part of RFC XXXX; see the RFC itself for full
  legal notices.";
// RFC Ed.: replace XXXX with actual RFC number and remove this note

revision 2008-10-14 {
  description
    "Initial revision, published as RFC XXXX.";
}
// RFC Ed.: replace XXXX with actual RFC number and remove this note

/*** collection of SNMP specific data types ***/

typedef admin-string {
  type string {
    length "0..255";
  }
  description
   "Represents and SnmpAdminString as defined in RFC 3411.";
  reference
   "RFC 3411: An Architecture for Describing SNMP Management Frameworks";
}

typedef identifier {
  type admin-string {
    length "1..32";
  }
  description
   "Identifiers are used to name items in the SNMP configuration
    data store.";
}

typedef context {
  type admin-string {
    length "0..32";
  }
  description
   "The context type represents an SNMP context name.";
}

typedef sec-name {
  type admin-string;
  description
    "The sec-name type represents an SNMP security name.";
}

typedef mp-model {
  type union {
    type enumeration {
      enum any     { value 0; }
      enum SNMPv1  { value 1; }
      enum SNMPv2c { value 2; }
      enum SNMPv3  { value 3; }
    }
```

```
      type int32 {
        range "0..2147483647";
      }
    }
    reference
     "RFC3411: An Architecture for Describing SNMP Management Frameworks";
  }

  typedef sec-model {
    type union {
      type enumeration {
        enum any     { value 0; }
        enum SNMPv1  { value 1; }
        enum SNMPv2c { value 2; }
        enum USM     { value 3; }
      }
      type int32 {
        range "0..2147483647";
      }
    }
    reference
     "RFC3411: An Architecture for Describing SNMP Management Frameworks";
  }

  typedef sec-level {
    type enumeration {
      enum no-auth-no-priv { value 1; }
      enum auth-no-priv    { value 2; }
      enum auth-priv       { value 3; }
    }
    reference
     "RFC3411: An Architecture for Describing SNMP Management Frameworks";
  }

  typedef engineid {
    type binary {
      length "5..32";
    }
    reference
     "RFC3411: An Architecture for Describing SNMP Management Frameworks";
  }

  container snmp {
    description
      "Top-level container for SNMP related configuration and
       status objects.";
  }

}
```

C SNMP VACM Yang Submodule

```
submodule snmp-vacm {

  /* $Id: snmp-vacm.yang 3016 2008-11-03 08:56:59Z mbj@tail-f.com $ */

  belongs-to snmp {
```

```
  prefix snmp;
}

include "snmp-common";

organization
 "IETF NETMOD (NETCONF Data Modeling Language) Working Group";

contact
 "Editor:   Juergen Schoenwaelder
            <mailto:j.schoenwaelder@jacobs-university.de>";

description
 "This submodule contains a collection of YANG definitions for
  configuring the View-based Access Control Model (VACM) of
  SNMP via NETCONF.

  Copyright (C) The IETF Trust (2008).  This version of this
  YANG module is part of RFC XXXX; see the RFC itself for full
  legal notices.";
// RFC Ed.: replace XXXX with actual RFC number and remove this note

revision 2008-10-11 {
  description
    "Initial revision, published as RFC XXXX.";
}
// RFC Ed.: replace XXXX with actual RFC number and remove this note

/*** collection of VACM specific data types ***/

typedef view-name {
  type snmp:identifier;
  description
   "The view-name type represents an SNMP VACM view name.";
  reference
   "RFC3415: View-based Access Control Model (VACM) for the
             Simple Network Management Protocol (SNMP)";
}

typedef group-name {
  type snmp:identifier;
  description
   "The view-name type represents an SNMP VACM group name.";
  reference
   "RFC3415: View-based Access Control Model (VACM) for the
             Simple Network Management Protocol (SNMP)";
}

typedef wildcard-object-identifier {
  type string {
    pattern '((([0-1]|\*)(\.(([1-3]?[0-9])|\*)))'
          + '|((2|\*)\.((0|([1-9]\d*))|\*)))'
          + '(\.((0|([1-9]\d*))|\*))*';
  }
  description
   "The wildcard-object-identifier type represents an SNMP
    object identifier where subidentifiers can be a wildcard,
    represented by a *.";
```

```
}

augment /snmp:snmp {

  container vacm {
    config true;
    description
     "Configuration of the View-based Access Control Model (VACM).";

    /*** group definition (vacmSecurityToGroupTable) ***/

    list group {
      key name;
      description
       "Mapping of securityName and securityModel pairs into
        groups according to the vacmSecurityToGroupTable of
        the SNMP-VIEW-BASED-ACM-MIB.";

      leaf name {
        type group-name;
        description
         "The name of this VACM group.";
      }

      list member {
        key "sec-name";
        min-elements 1;
        description
         "A member of this VACM group. According to VACM, every
          group must have at least one member.";

        leaf sec-name {
          type snmp:sec-name;
          description
            "The securityName of a group member.";
        }

        leaf-list sec-model {
          min-elements 1;
          type snmp:sec-model;
          description
           "The securityModels under which this securityName
            is a member of this group.";
        }
      }
    }

    /*** access definition (vacmAccessTable) ***/

    list access {
      key "group context sec-model sec-level";
      description
       "Definition of access right for groups according to the
        vacmAccessTable of the SNMP-VIEW-BASED-ACM-MIB.";

      leaf group {
        type keyref {
          path "../../group/name";
```

```
    }
    description
     "The group to which the access rights apply.";
  }

  leaf context {
    type snmp:context;
    description
     "The context (prefix) under which the access rights apply.";
  }

  leaf sec-model {
    type snmp:sec-model;
    description
     "The security model under which the access rights apply.";
  }

  leaf sec-level {
    type snmp:sec-level;
    description
     "The minimum security level under which the access rights
      apply.";
  }

  leaf prefix-match {
    type empty;
    description
     "If present, the context must only match the prefix of
      a request. If absent, an exact match is required.";
  }

  leaf read-view {
    type view-name;
    description
     "The name of the MIB view of the SNMP context authorizing
      read access.";
  }

  leaf write-view {
    type view-name;
    description
     "The name of the MIB view of the SNMP context authorizing
      write access.";
  }

  leaf notify-view {
    type view-name;
    description
     "The name of the MIB view of the SNMP context authorizing
      notify access.";
  }
}

/*** view definition (vacmViewTreeFamilyTable) ***/

list view {
  key name;
  description
```

```
        "Definition of MIB views according to the
         vacmViewTreeFamilyTable of the SNMP-VIEW-BASED-ACM-MIB.";

       leaf name {
         type view-name;
         description
          "The name of this VACM MIB view.";
       }

       list subtree {
         key "oids";

         leaf oids {
           type wildcard-object-identifier;
           description
           "A family of subtrees included in this MIB view.";
         }

         choice type {
           mandatory true;
           leaf included {
             type empty;
             description
              "The family of subtrees is included in the MIB view";
           }
           leaf excluded {
             type empty;
             description
              "The family of subtrees is excluded from the MIB view";
           }
         }
       }
     }
   }
 }
}
```