ntc-models Documentation

Release 0.0.1

David Barroso

Jun 05, 2019

Contents

CHAPTER 1

Contents

1.1 Models

1.1.1 ntc-arp

Model to configure and retrieve operational state of ARP and ARP entries

Data nodes

/arp

Top-level container for ARP model

nodetype: `container`

/arp/config

ARP configuration

nodetype: `container`

/arp/config/timeout

Cache arp entries for this amount of time (seconds)

nodetype: `leaf`

Type: `uint16`

/arp/config/entries

List of ARP entries configured in the system

nodetype: `list`

/arp/config/entries/ip-address

IP address associated to the ARP entry

nodetype: `leaf` (list key)

Type: `ntc-types:ip-address`

/arp/config/entries/hw-address

Physical address associated to the ARP entry

nodetype: `leaf`

Type: `yang:mac-address`

/arp/config/entries/vrf

VRF associoated to the ARP entry

nodetype: `leaf`

Type: `leafref`

- **path reference**: `/vrf/config/vrfs/name`

/arp/state

ARP state

nodetype: `container`

/arp/state/timeout

Cache arp entries for this amount of time (seconds)

nodetype: `leaf`

Type: `uint16`

/arp/state/entries

List of ARP entries in the system

nodetype: `list`

/arp/state/entries/ip-address

IP address associated to the ARP entry

nodetype: `leaf` (list key)

Type: `ntc-types:ip-address`

/arp/state/entries/hw-address

Physical address associated to the ARP entry

nodetype: `leaf`

Type: `yang:mac-address`

/arp/state/entries/vrf

VRF associoated to the ARP entry

nodetype: `leaf`

Type: `leafref`

- **path reference**: `/vrf/state/vrfs/name`

1.1.2 ietf-yang-types

This module contains a collection of generally useful derived YANG data types.

This version of this YANG module is part of RFC 6991; see the RFC itself for full legal notices.

Types

counter32

The counter32 type represents a non-negative integer that monotonically increases until it reaches a maximum value of 2^32-1 (4294967295 decimal), when it wraps around and starts increasing again from zero.

Counters have no defined 'initial' value, and thus, a single value of a counter has (in general) no information content. Discontinuities in the monotonically increasing value normally occur at re-initialization of the management system, and at other times as specified in the description of a schema node using this type. If such other times can occur, for example, the creation of a schema node of type counter32 at times other than re-initialization, then a corresponding schema node should be defined, with an appropriate type, to indicate the last discontinuity.

The counter32 type should not be used for configuration schema nodes. A default statement SHOULD NOT be used in combination with the type counter32.

In the value set and its semantics, this type is equivalent to the Counter32 type of the SMIv2.

type: `uint32`

zero-based-counter32

The zero-based-counter32 type represents a counter32 that has the defined 'initial' value zero.

A schema node of this type will be set to zero (0) on creation and will thereafter increase monotonically until it reaches a maximum value of 2^32-1 (4294967295 decimal), when it wraps around and starts increasing again from zero.

Provided that an application discovers a new schema node of this type within the minimum time to wrap, it can use the 'initial' value as a delta. It is important for a management station to be aware of this minimum time and the actual time between polls, and to discard data if the actual time is too long or there is no defined minimum time.

In the value set and its semantics, this type is equivalent to the ZeroBasedCounter32 textual convention of the SMIv2.

type: `counter32`

counter64

The counter64 type represents a non-negative integer that monotonically increases until it reaches a maximum value of 2^64-1 (18446744073709551615 decimal), when it wraps around and starts increasing again from zero.

Counters have no defined 'initial' value, and thus, a single value of a counter has (in general) no information content. Discontinuities in the monotonically increasing value normally occur at re-initialization of the management system, and at other times as specified in the description of a schema node using this type. If such other times can occur, for example, the creation of a schema node of type counter64 at times other than re-initialization, then a corresponding schema node should be defined, with an appropriate type, to indicate the last discontinuity.

The counter64 type should not be used for configuration schema nodes. A default statement SHOULD NOT be used in combination with the type counter64.

In the value set and its semantics, this type is equivalent to the Counter64 type of the SMIv2.

type: `uint64`

zero-based-counter64

The zero-based-counter64 type represents a counter64 that has the defined 'initial' value zero.

A schema node of this type will be set to zero (0) on creation and will thereafter increase monotonically until it reaches a maximum value of 2^64-1 (18446744073709551615 decimal), when it wraps around and starts increasing again from zero.

Provided that an application discovers a new schema node of this type within the minimum time to wrap, it can use the 'initial' value as a delta. It is important for a management station to be aware of this minimum time and the actual time between polls, and to discard data if the actual time is too long or there is no defined minimum time.

In the value set and its semantics, this type is equivalent to the ZeroBasedCounter64 textual convention of the SMIv2.

type: `counter64`

gauge32

The gauge32 type represents a non-negative integer, which may increase or decrease, but shall never exceed a maximum value, nor fall below a minimum value. The maximum value cannot be greater than 2^32-1 (4294967295 decimal), and the minimum value cannot be smaller than 0. The value of a gauge32 has its maximum value whenever the information being modeled is greater than or equal to its maximum value, and has its minimum value whenever the information being modeled is smaller than or equal to its minimum value. If the information being modeled subsequently decreases below (increases above) the maximum (minimum) value, the gauge32 also decreases (increases).

In the value set and its semantics, this type is equivalent to the Gauge32 type of the SMIv2.

type: `uint32`

gauge64

The gauge64 type represents a non-negative integer, which may increase or decrease, but shall never exceed a maximum value, nor fall below a minimum value. The maximum value cannot be greater than 2^64-1 (18446744073709551615), and the minimum value cannot be smaller than 0. The value of a gauge64 has its maximum value whenever the information being modeled is greater than or equal to its maximum value, and has its minimum value whenever the information being modeled is smaller than or equal to its minimum value. If the information being modeled subsequently decreases below (increases above) the maximum (minimum) value, the gauge64 also decreases (increases).

In the value set and its semantics, this type is equivalent to the CounterBasedGauge64 SMIv2 textual convention defined in RFC 2856

type: `uint64`

object-identifier

The object-identifier type represents administratively assigned names in a registration-hierarchical-name tree.

Values of this type are denoted as a sequence of numerical non-negative sub-identifier values. Each sub-identifier value MUST NOT exceed 2^32-1 (4294967295). Sub-identifiers are separated by single dots and without any intermediate whitespace.

The ASN.1 standard restricts the value space of the first sub-identifier to 0, 1, or 2. Furthermore, the value space of the second sub-identifier is restricted to the range 0 to 39 if the first sub-identifier is 0 or 1. Finally, the ASN.1 standard requires that an object identifier has always at least two sub-identifiers. The pattern captures these restrictions.

Although the number of sub-identifiers is not limited, module designers should realize that there may be implementations that stick with the SMIv2 limit of 128 sub-identifiers.

This type is a superset of the SMIv2 OBJECT IDENTIFIER type since it is not restricted to 128 sub-identifiers. Hence, this type SHOULD NOT be used to represent the SMIv2 OBJECT IDENTIFIER type; the object-identifier-128 type SHOULD be used instead.

type: `string`

pattern: `(([0-1](\.[1-3]?[0-9]))|(2\.(0|([1-9]\d*))))(\.(0|([1-9]\d*)))*`

object-identifier-128

This type represents object-identifiers restricted to 128 sub-identifiers.

In the value set and its semantics, this type is equivalent to the OBJECT IDENTIFIER type of the SMIv2.

type: `object-identifier`

yang-identifier

A YANG identifier string as defined by the 'identifier' rule in Section 12 of RFC 6020. An identifier must start with an alphabetic character or an underscore followed by an arbitrary sequence of alphabetic or numeric characters, underscores, hyphens, or dots.

A YANG identifier MUST NOT start with any possible combination of the lowercase or uppercase character sequence 'xml'.

type: `string`

pattern: `[a-zA-Z_][a-zA-Z0-9\-_.]*`

date-and-time

The date-and-time type is a profile of the ISO 8601 standard for representation of dates and times using the Gregorian calendar. The profile is defined by the date-time production in Section 5.6 of RFC 3339.

The date-and-time type is compatible with the dateTime XML schema type with the following notable exceptions:

(a) The date-and-time type does not allow negative years.

(b) The date-and-time time-offset -00:00 indicates an unknown

time zone (see RFC 3339) while -00:00 and +00:00 and Z all represent the same time zone in dateTime.

(c) The canonical format (see below) of data-and-time values

differs from the canonical format used by the dateTime XML schema type, which requires all times to be in UTC using the time-offset 'Z'.

This type is not equivalent to the DateAndTime textual convention of the SMIv2 since RFC 3339 uses a different separator between full-date and full-time and provides higher resolution of time-secfrac.

The canonical format for date-and-time values with a known time zone uses a numeric time zone offset that is calculated using the device's configured known offset to UTC time. A change of the device's offset to UTC time will cause date-and-time values to change accordingly. Such changes might happen periodically in case a server follows automatically daylight saving time (DST) time zone offset changes. The canonical format for date-and-time values with an unknown time zone (usually referring to the notion of local time) uses the time-offset -00:00.

type: `string`

pattern: `\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d+)?(Z|[\+\-]\d{2}:\d{2})`

timeticks

The timeticks type represents a non-negative integer that represents the time, modulo 2^32 (4294967296 decimal), in hundredths of a second between two epochs. When a schema node is defined that uses this type, the description of the schema node identifies both of the reference epochs.

In the value set and its semantics, this type is equivalent to the TimeTicks type of the SMIv2.

type: `uint32`

timestamp

The timestamp type represents the value of an associated timeticks schema node at which a specific occurrence happened. The specific occurrence must be defined in the description of any schema node defined using this type. When the specific occurrence occurred prior to the last time the associated timeticks attribute was zero, then the timestamp value is zero. Note that this requires all timestamp values to be reset to zero when the value of the associated timeticks attribute reaches 497+ days and wraps around to zero.

The associated timeticks schema node must be specified in the description of any schema node using this type.

In the value set and its semantics, this type is equivalent to the TimeStamp textual convention of the SMIv2.

type: `timeticks`

phys-address

Represents media- or physical-level addresses represented as a sequence octets, each octet represented by two hexadecimal numbers. Octets are separated by colons. The canonical representation uses lowercase characters.

In the value set and its semantics, this type is equivalent to the PhysAddress textual convention of the SMIv2.

type: `string`

pattern: `([0-9a-fA-F]{2}(:[0-9a-fA-F]{2})*)?`

mac-address

The mac-address type represents an IEEE 802 MAC address. The canonical representation uses lowercase characters.

In the value set and its semantics, this type is equivalent to the MacAddress textual convention of the SMIv2.

type: `string`

pattern: `[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}`

xpath1.0

This type represents an XPATH 1.0 expression.

When a schema node is defined that uses this type, the description of the schema node MUST specify the XPath context in which the XPath expression is evaluated.

type: `string`

hex-string

A hexadecimal string with octets represented as hex digits separated by colons. The canonical representation uses lowercase characters.

type: `string`

pattern: `([0-9a-fA-F]{2}(:[0-9a-fA-F]{2})*)?`

uuid

A Universally Unique IDentifier in the string representation defined in RFC 4122. The canonical representation uses lowercase characters.

The following is an example of a UUID in string representation: f81d4fae-7dec-11d0-a765-00a0c91e6bf6

type: `string`

pattern: `[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}`

dotted-quad

An unsigned 32-bit number expressed in the dotted-quad notation, i.e., four octets written as decimal numbers and separated with the '.' (full stop) character.

type: `string`

pattern: `(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])`

1.1.3 ntc-system

Model to configure and retrieve system data

Types

snmp-version

SNMP version

type: `enumeration`

- `SNMP_VERESION_1`: SNMP version 1
- `SNMP_VERSION_2C`: SNMP version 2c

- `SNMP_VERSION_3`: SNMP version 3

Data nodes

/system

Top level container for system configuration and state

nodetype: `container`

/system/config

Top level container for system configuration

nodetype: `container`

/system/config/snmp

Top level container for SNMP configuration

nodetype: `container`

/system/config/snmp/communities

List of communities in the system

nodetype: `list`

/system/config/snmp/communities/name

Name of community

nodetype: `leaf` (list key)

Type: `string`

/system/config/snmp/communities/version

SNMP version allowed by the community

nodetype: `leaf`

Type: `snmp-version`

/system/config/snmp/communities/access-list

Access list protecting the community

nodetype: `container`

/system/config/snmp/communities/access-list/ipv4

IPv4 access-list

nodetype: `leaf`

Type: `string`

/system/config/snmp/communities/access-list/ipv6

IPv6 access-list

nodetype: `leaf`

Type: `string`

/system/config/snmp/name

Name of the commnity

nodetype: `leaf`

Type: `string`

/system/config/snmp/description

Description of the system

nodetype: `leaf`

Type: `string`

/system/config/snmp/contact

Contact information for the system

nodetype: `leaf`

Type: `string`

/system/config/snmp/location

Location information of the system

nodetype: `leaf`

Type: `string`

/system/state

Top level container for system state

nodetype: `container`

/system/state/snmp

Top level container for SNMP state

nodetype: `container`

/system/state/snmp/communities

List of communities in the system

nodetype: `list`

/system/state/snmp/communities/name

Name of community

nodetype: `leaf` (list key)

Type: `string`

/system/state/snmp/communities/version

SNMP version allowed by the community

nodetype: `leaf`

Type: `snmp-version`

/system/state/snmp/communities/access-list

Access list protecting the community

nodetype: `container`

/system/state/snmp/communities/access-list/ipv4

IPv4 access-list

nodetype: `leaf`

Type: `string`

/system/state/snmp/communities/access-list/ipv6

IPv6 access-list

nodetype: `leaf`

Type: `string`

/system/state/snmp/name

Name of the commnity

nodetype: `leaf`

Type: `string`

/system/state/snmp/description

Description of the system

nodetype: `leaf`

Type: `string`

/system/state/snmp/contact

Contact information for the system

nodetype: `leaf`

Type: `string`

/system/state/snmp/location

Location information of the system

nodetype: `leaf`

Type: `string`

1.1.4 ntc-types

Common types

Types

ip-address

A bare IPv4 or IPv6 address.

type: `union`

- **Type**: `ipv4-address`
- **Type**: `ipv6-address`

ipv4-address

An IPv4 address in dotted quad notation using the default zone (copied from openconfig)

type: `string`

pattern: `(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])`

ipv6-address

An IPv6 address represented as either a full address; shortened or mixed-shortened formats, using the default zone (copied from openconfig)

type: `string`

pattern: `(([0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,7}:|([0-9a-fA-F]{1,4}:){1,6}:[0-9a-fA-F]{1,4}|([0-9a-fA-F]{1,4}:){1,5}(:[0-9a-fA-F]{1,4}){1,2}|([0-9a-fA-F]{1,4}:){1,4}(:[0-9a-fA-F]{1,4}){1,3}|([0-9a-fA-F]{1,4}:){1,3}(:[0-9a-fA-F]{1,4}){1,4}|([0-9a-fA-F]{1,4}:){1,2}(:[0-9a-fA-F]{1,4}){1,5}|[0-9a-fA-F]{1,4}:((:[0-9a-fA-F]{1,4}){1,6})|:((:[0-9a-fA-F]{1,4}){1,7}|:))`

1.1.5 ntc-vlan

Model to configure and retrieve operational state of VLANs

Data nodes

/vlan

Top-level container for VLAN configuration and state

nodetype: `container`

/vlan/config

Top-level container for VLAN configuration

nodetype: `container`

/vlan/config/vlans

List of VLANs

nodetype: `list`

/vlan/config/vlans/vlan-id

VLAN identifier

nodetype: `leaf` (list key)

Type: `uint16`

- **range**: `1..4094`

/vlan/config/vlans/name

VLAN name

nodetype: `leaf`

Type: `string`

/vlan/config/vlans/active

Whether the VLAN is enabled and bridging traffic or not

nodetype: `leaf`

Type: `boolean`

/vlan/state

Top-level container for VLAN state

nodetype: `container`

/vlan/state/vlans

List of VLANs

nodetype: `list`

/vlan/state/vlans/vlan-id

VLAN identifier

nodetype: `leaf` (list key)

Type: `uint16`

- **range**: `1..4094`

/vlan/state/vlans/name

VLAN name

nodetype: `leaf`

Type: `string`

/vlan/state/vlans/active

Whether the VLAN is enabled and bridging traffic or not

nodetype: `leaf`

Type: `boolean`

/vlan/state/vlans/members

Interfaces in this VLAN

nodetype: `leaf-list`

Type: `string`

1.1.6 ntc-vrf

A simplfied model to manage VRFs

Data nodes

/vrf

Top container for VRF configuration and state

nodetype: `container`

/vrf/config

VRF configuration

nodetype: `container`

/vrf/config/vrfs

List of VRFs

nodetype: `list`

/vrf/config/vrfs/name

Name of VRF

nodetype: `leaf` (list key)

Type: `string`

/vrf/state

VRF state

nodetype: `container`

/vrf/state/vrfs

List of VRFs

nodetype: `list`

/vrf/state/vrfs/name

Name of VRF

nodetype: `leaf` (list key)

Type: `string`

Indices and tables

- genindex
- modindex
- search