NSO Resource Manager 4.2.6

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

CONTENTS

CHAPTER 1

NSO Resource Manager Guide

Introduction

NSO Resource Manager package contains both an API for generic resource pool handling called `resource allocator`, and two applications utilizing the API. The applications are the `id-allocator` and the `ipaddress-allocator`, explained in separate chapters. This version of NSO Resource Manager is 4.2.6 and was released together with NSO version 6.3.

Background

NSO is often used to provision services in the networking layer. It is not unusual that these services require network-level information that are not (or cannot be) part of the instance data provided by the northbound system, so it needs to be fetched from, and eventually released back to a separate system. A common example of this is IP-addresses used for layer 3 VPN services. The orchestrator tool is not aware of the blocks of IP-addresses assigned to the network but relies on lower layers to fulfill this need.

Some customers have software systems to manage these types of temporary assets. E.g. for IP-addresses they are usually known as IP Address Management (IPAM) systems. There is a whole industry of solutions for such systems, ranging from simple open source solutions to entire suites integrated with DNS management. See https://en.wikipedia.org/wiki/IP_address_management for more on this.

There are customers that either don't have an IPAM system for services that are planned for NSO and are not planning to get one for this single purpose. They usually don't want the operational overhead of another system and/or don't see the need of a separate investment.

These customers are looking for NSO to provide basic resource allocation and lifecycle management for the assets required for services managed by NSO. They appreciate the fact that NSO is not an appropriate platform to provide more advanced features from the IPAM world like capacity planning nor to integrate

with DNS and DHCP platforms. This means that the `NSO Resource Manager` does not compete with full-blown systems, but rather is a complementary feature.

Overview

The `NSO Resource Manager` interface, the `resource allocator`, provides a generic resource allocation mechanism that works well with `services` and in a high availability (HA) configuration. Expected is implementations of specific resource allocators implemented as separate NSO packages. A `service` will then have the possibility to use allocator implementations dedicated for different resources.

The YANG model of the `resource allocator` (`resource-allocator.yang`) can be augmented with different resource pools, as is the case for the two applications `id-allocator` and `ipaddress-allocator`. Each pool has an `allocation` list where services are expected to create instances to signal that they request an allocation. Request parameters are stored in the `request` container and the allocation response is written in the `response` container.

Since the allocation request may fail the response container contains a choice where one case is for error and one for success.

Each allocation list entry also contains an `allocating-service` leaf-list. These are instance-identifiers that point to the services that requested the resource. These are the services that will be re-deployed when the resource has been allocated.

Resource allocation packages should subscribe to several points in this `resource-pool` tree. First, they must detect when a new resource pool is created or deleted, secondly they must detect when an allocation request is created or deleted. A package may also augment the pool definition with additional parameters, for example, an ip address allocator may wish to add configuration parameters for defining the available subnets to allocate from, in which case it must also subscribe to changes to these settings.

Installation

Installation of this package is done as with any other package, as described in the NSO Packages chapter in NSO 6.3 Administration Guide.

Resource Allocator Data Model

The API of the resource allocator is defined in this YANG data model:

```
grouping resource-pool-grouping {
  leaf name {
    tailf:info "Unique name for the pool";
    type string;
  }

  list allocation {
    key id;

    leaf id {
      type string;
    }

    leaf username {
      description
        "Authenticated user for invoking the service";
      type string;
      mandatory true;
    }
```

```
leaf-list allocating-service {
  tailf:info "Instance identifiers of service that own resource";
  type instance-identifier;
}

container request {
  description
    "When creating a request for a resource the
     implementing package augments here.";
}

container response {
  config false;
  tailf:cdb-oper {
    tailf:persistent true;
  }
  choice response-choice {
    case error {
      leaf error {
        type string;
      }
    }
    case ok {
      // The implementing package augments here
    }
  }
}
}
```

HA Considerations

Looking at `High Availability` there are two things we need to consider - the allocator state needs to be replicated, and the allocation needs only to be performed on one node.

The easiest way to replicate the state is to write it into CDB-oper and let CDB perform the replication. This is what we do in the `ipaddress-allocator`.

We only want the allocator to allocate addresses on the primary node. Since the allocations are written into CDB they will be visible on both primary and secondary nodes, and the CDB subscriber will be notified on both nodes. In this case we only want the allocator on the primary node to perform the allocation.

We therefore read the HA mode leaf from CDB to determine which HA mode the current subscriber is running in; if HA mode is not configured and enabled in ncs.conf, or if HA is configured and enabled in ncs.conf and the current node is primary we proceed with the allocation.

Synchronous Allocation

This synchronized allocation API request uses a reactive fast map, so the user can allocate resources and still keep a synchronous interface. It allocates resources in the create callback, at that moment everything we modify in the database is part of the service intent and fast map. We need to guarantee that we have used a stable resource and communicate to other services, which resources we have used. So, during the create callback, we store what we have allocated. Other services that are evaluated within the same transaction which runs subsequent to ours will see allocations, when our service is redeployed, it will not have to create the allocations again.

When an allocation raises an exception in case the pool is exhausted, or if the referenced pool does not exist in the CDB, `commit` will get aborted. Synchronous allocation doesn't require service `re-`

`deploy` to read allocation. The same transaction can read allocation, `commit dry-run` or `get-modification` should show up the allocation details as output.

`Synchronous allocation is only supported through Java and Python APIs provided by resource-manager.`

CHAPTER 2

NSO ID Allocator Deployment Guide

Introduction

This document contains deployment information and procedures for the `NSO ID Allocator`.

The `NSO ID Allocator` is an extension of the generic resource allocation mechanism named `NSO Resource Manager`. It can allocate integers which can serve for instance as VLAN identifiers.

Overview

The `ID Allocator` can host any number of ID pools. Each pool contains a certain number of IDs that can be allocated. They are specified by a range, and potentially broken into several ranges by a list of excluded ranges.

The `ID allocator` YANG models are divided into a configuration data specific model (`id-allocator.yang`), and an operational data specific model (`id-allocator-oper.yang`). Users of this package will request allocations in the configuration tree. The operational tree serves as an internal data structure of the package.

An ID request can allocate either the lowest possible ID in a pool, or a specified (by the user) value, such as `5` or `1000`.

Allocation requests can be synchronized between pools. This synchronization is based on the id of the allocation request itself (such as for instance `allocation1`), the result is that the allocations will have the same allocated value across pools.

Examples

This section presents some simple use cases of the `NSO ID Allocator`. The examples below are presented using Cisco style CLI.

Create an ID pool

The CLI interaction below depicts how it is possible to create a new ID pool, and assign it a range of values from `100` to `1000`.

```
admin@ncs# resource-pools id-pool pool1 range start 100 end 1000
admin@ncs# commit
```

Create an allocation request

When a pool has been created, it is possible to create allocation requests on the values handled by a pool. The CLI interaction below shows how to allocate a value in the pool defined above.

```
admin@ncs# resource-pools id-pool pool1 allocation a1 user myuser
admin@ncs# commit
```

At this point, we have a pool with range 100 to 1000 and one allocation (100). This is shown in Table 1, “Pool range 100-1000”

Table 1. Pool range 100-1000

NAME	START	END	START	END	START	END	ID
pool1	-	-			101	1000	100

Create an allocation request shared by multiple services

Allocations can be shared by multiple services by requesting the same allocation ID from all the services. All instance services in the `allocating-service` leaf-list will be re-deployed when the resource has been allocated. The CLI interaction below shows how to allocate an ID shared by two services.

```
admin@ncs# resource-pools id-pool pool1 allocation a1 allocating-service \
           /services/vl:loop[name='myservice1'] user myuser
admin@ncs# resource-pools id-pool pool1 allocation a1 allocating-service \
           /services/vl:loop[name='myservice2'] user myuser
admin@ncs# commit
```

The allocation resource gets freed once all allocating services in the `allocating-service` leaf-list delete the allocation request.

Create a synchronized allocation request

Allocations can be synchronized between pools by setting `request sync` to `true` when creating each allocation request. The allocation id, which is `b` in this CLI interaction, determines which allocations will be synchronized across pools.

```
admin@ncs# resource-pools id-pool pool2 range start 100 end 1000
admin@ncs# resource-pools id-pool pool1 allocation b user myuser request sync true
```

```
admin@ncs# resource-pools id-pool pool2 allocation b user myuser request sync true
admin@ncs# commit
```

As can be seen in Table 2, “Synchronized pools”, allocations b (in `pool1` and in `pool2`) are synchronized across pools pool1 and pool2 and receive the ID value of 1000 in both pools.

Table 2. Synchronized pools

NAME	START	END	START	END	START	END	ID
pool1	-	-			101	999	100
	-	-					1000
pool2	-	-			101	999	1000

Update Request Id

The element allocation/request/id can be created, changed, then previously allocated id will be released and new id will be allocated depending on the new value of allocation/request/id. In the case of a delete request/id, the previously allocated id will be retained.

```
admin@ncs# set resource-pools id-pool testPool allocation testAlloc username admin
admin@ncs# commit
admin@ncs# set resource-pools id-pool testPool allocation testAlloc request id 150
admin@ncs# commit
admin@ncs# set resource-pools id-pool testPool allocation testAlloc request id 180
admin@ncs# commit
```

Request an id using round robin method

Default behavior for requesting a new id is to request the first free id in increasing order.

This method is selectable using the 'method' container. For example the 'first free' method can be explicitly set:

```
admin@ncs# set resource-pools id-pool methodRangeFirst allocation a username \
             admin request method firstfree
```

If we remove allocation a, and do a new allocation, using the default method we allocate the first free id, in this case 1 again. Using the round robin scheme, we instead allocate the next in order, i.e. 2.

```
admin@ncs# set resource-pools id-pool methodRoundRobin allocation a username \
             admin request method roundrobin
```

Note Note that the request method is set on a per-request basis. Two different requests may request ids from the same pool using different request methods.

Create a synchronous allocation API request for a ID

Synchrous allocation can be requested through various Java APIs provided in `resource-manager/src/java/src/com/tailf/pkg/idallocator/IDAllocator.java` and python API provided in `resource-manager/python/resource_manager/id_allocator.py`.

- Request:`Java:` void idRequest(ServiceContext context, NavuNode service, RedeployType redeployType, String poolName, String username, String id, boolean sync, long requestedId, boolean sync_alloc)`Python:` id_request(service, svc_xpath, username, pool_name, allocation_name, sync, requested_id=-1, redeploy_type="default", alloc_sync=False, root=None)
- Check Response Ready:`Java:` boolean responseReady(NavuContext context, String poolName, String id)
- Read Response:`Java:` ConfUInt32 idRead(NavuContext context, String poolName, String id)`Python:` id_read(username, root, pool_name, allocation_name)

Security

The `NSO ID Allocator` requires a username to be configured by the service application when creating an allocation request. This username will be used to re-deploy the service application once a resource has been allocated. Default NACM rules denies all standard users access to the `/ralloc:resource-pools` list. These default settings are provided in the (`initial_data/aaa_init.xml`) file of the `resource-manager` package.

It's up to the administrator to add a rule that allows the user to perform the service re-deploy.

How the administrator should write these rules are detailed in the The AAA Infrastructure chapter in NSO 6.3 Administration Guide.

Alarms

There are two alarms associated with the ID Allocator:

Empty alarm

This alarm is raised when the pool is empty, there are no availabe IDs for further allocation.

Low threshold reached alarm

This alarm is raised when the pool is nearing empty, e.g. there is only ten percent or less left in the pool.

CDB upgrade from package version below 4.0.0

Since resource-manager version 4.0.0 the operational data model is not compapatible with previous version. In version 4.0.0 yang model, there is a new element "allocationId" added for /id-allocator/pool/allocation to support sync Id allocation. The system will run the upgrade script automatically when resource-manager of new version is loaded if there is yang model change in the new version. User can also run the script manually for resource-manager from 3.5.6(or any version below 4.0.0) to version 4.0.0 or above , the script will add missing "allocationId" element in the CDB operational data path /id-allocator/pool/allocation. The upgrade python script is located in the resource-manager package: python/resource_manager/rm_upgrade_nso.py

Important: After run the script manually to update CDB , user must request package reload or restart ncs to reload new CBD data into ID Pool java object in memory. For example, in nso-cli console, admin@ncs> request packages reload force

id-allocator-tool Action

A set of Debug and Data tools contained in rm-action/id-allocator-tool Action is available to help admin or support to operate on RM data . Two parameters in the id-allocator-tool Action can be provided:

`operation`, `pool`. All the process info and result will be logged in ncs-java-vm.log, and the action itself just return result `Done!` Here is list of the valid operation value for the id-allocator-tool Action

- check_missing_report: scan the current resource pool and idPool in the system, and identify and report the missing element for each id-allocator entry without fixing
- fix_missing_allocation_id: add missing allocation Id for each id-allocator entry.
- fix_missing_owner: add missing owners info for each id-allocator entry.
- fix_missing_allocation: create missing allocation entry in id-allocator for each idPool allocation response/id
- fix_response_id: scan the idPool, and check if the allocation contains invalid allocation request id, release the allocation from idpool if found. It happens for sync allocation when device configuration fail after a successful id allocation and then cause a service transaction fail. This leaves the idpool contains success allocated id while allocation request response doesn't exist
- persistAll: manually sync from idPool in memory to id-allocator in CDB
- printIdPool: print current idPool data in the ncs-java-vm.log for debug purpose.

Action usage example

Note that when pool parameter is provided , the operation will be on this specific idPool, and if no pool provided, the operation will be running on all idPool in the system .

```
admin@ncs>  unhide debug

admin@ncs>  request rm-action id-allocator-tool operation fix_missing_allocation

admin@ncs>  request rm-action id-allocator-tool operation printIdPool pool multiService
```

CHAPTER 3

NSO IP Address Allocator Deployment Guide

Introduction

This document contains deployment information and procedures for the Tail-f `NSO IP Address Allocator` application.

Overview

The `NSO IP Address Allocator` application contains an IP address allocator that use the `Resource Manager` API to provide IP address allocation. It uses a RAM based allocation algorithm that stores its state in CDB as oper data.

The file `resource-manager/src/java/src/com/tailf/pkg/ipaddressallocator/IPAddressAllocator.java` contains the part that deals with the resource manager APIs whereas the RAM based IP address allocator resides under `resource-manager/src/java/src/com/tailf/pkg/ipam`

The `IPAddressAllocator` class subscribes to five points in the DB:

`/ralloc:resource-pools/ip-address-pool`
: To be notified when new pools are created/deleted. It needs to create/delete instances of the `IPAddressPool` class. Each instance of the `IPAddressPool` handles one pool.

`/ralloc:resource-pools/ip-address-pool/subnet`
: To be notified when subnets are added/removed from an existing address pool. When a new subnet is added it needs to invoke the `addToAvailable` method of the right `IPAddressPool` instance. When a pool is removed it needs to reset all existing allocations from the pool, create new allocations, and re-deploy the services that had the allocations.

`/ralloc:resource-pols/ip-address-pool/exclude`
: To detect when new exlcusions are added, and when old exlusions are removed.

`/ralloc:resource-pols/ip-address-pool/range`

To be notified when ranges are added to or removed from an address pool.

`/ralloc:resource-pols/ip-address-pool/allocation`

To detect when new allocation requests are added, and when old allocations are released. When a new request is added the right size of subnet is allocated from the `IPAddressPool` instance, and the result is written to the `response/subnet` leaf, and finally the service is re-deployed.

Examples

This section presents some simple use cases of the `NSO IP Address Allocator`. It uses the C-style CLI.

Create an IP pool

Creating an IP pool requires the user to specify a list of subnets (identified by a network address and a CIDR mask), a list of IP ranges (identified by its first and last IP address), or a combination of the two to be handled by the pool. The following CLI interaction shows an allocation where a pool `pool1` is created, and the subnet `10.0.0.0/24` and the range `192.168.0.0 - 192.168.255.255` is added to it.

```
admin@ncs# resource-pools ip-address-pool pool1 subnet 10.0.0.0 24

admin@ncs# resource-pools ip-address-pool pool1 range 192.168.0.0 192.168.255.255
```

Create an allocation request for a subnet

Since we have already populated one of our pools, we can now start creating allocation requests. In the CLI interaction below, we request to allocate a subnet with a CIDR mask of 30, in the pool `pool1`.

```
admin@ncs# resource-pools ip-address-pool pool1 allocation a1 username \
              myuser request subnet-size 30
```

Create an allocation request for a subnet shared by multiple services

Allocations can be shared by multiple services by requesting the same subnet and using the same allocation ID. All instance services in the `allocating-service` leaf-list will be re-deployed when the resource has been allocated. The CLI interaction below shows how to allocate a subnect shared by two services.

```
admin@ncs# resource-pools ip-address-pool pool1 allocation a1 allocating-service \
            /services/vl:loop[name='myservice1'] user myuser request subnet-size 30

admin@ncs# resource-pools ip-address-pool pool1 allocation a1 allocating-service \
            /services/vl:loop[name='myservice2'] user myuser request subnet-size 30

admin@ncs# commit
```

The allocation resource gets freed once all allocating services in the `allocating-service` leaf-list delete the allocation request.

Create a static allocation request for a subnet

If you need a specific IP or range of IPs for an allocation, now you can use the optional `subnet-start-ip` leaf, together with the `subnet-size`. The allocator will go through the available subnets in the requested pool and will look for a subnet containing the `subnet-start-ip` and which can also fit the `subnet-size`.

```
admin@ncs# resource-pools ip-address-pool pool1 allocation a2 username \
              myuser request subnet-start-ip 10.0.0.36 subnet-size 32
```

The `subnet-start-ip` has to be the first IP address out of a subnet with size `subnet-size`:

- Valid: `subnet-start-ip 10.0.0.36 subnet-size 30`, IP range 10.0.0.36 to 10.0.0.39
- Invalid: `subnet-start-ip 10.0.0.36 subnet-size 29`, IP range 10.0.0.32 to 10.0.0.39

If the subnet-start-ip/subnet-size pair does not give a subnet range starting with subnet-start-ip, the allocation will fail.

Create a synchronous allocation request for a subnet

Synchrous allocation can be requested through various Java APIs provided in `resource-manager/src/java/src/com/tailf/pkg/ipaddressallocator/IPAddressAllocator.java` and python API provided in `resource-manager/python/resource_manager/ipadress_allocator.py`.

- Request:`Java:` void subnetRequest(ServiceContext context, NavuNode service, RedeployType redeployType, String poolName, String username, String startIp, int cidrmask, String id, boolean invertCidr, boolean sync)`Python:` def net_request(service, svc_xpath, username, pool_name, allocation_name, cidrmask, invert_cidr=False, redeploy_type="default", sync=False, root=None)
- Check Response Ready:`Java:` boolean responseReady(NavuContext context, String poolName, String id)
- Read Response:`Java:` ConfIPPrefix subnetRead(NavuContext context, String poolName, String id)`Python:` def net_read(username, root, pool_name, allocation_name)

Read the response to an allocation request

The response to an allocation request comes in the form of operational data written to the path `/resource-pools/ip-address-pool/allocation/response`. The response container contains a choice with two cases, `ok` and `error`. If the allocation failed, the `error` case will be set and an error message can be found in the leaf `error`. If the allocation succeeded, the `ok` case will be set and the allocated subnet will be written to the leaf `subnet` and the subnet from which the allocation was made will be written to the leaf `from`. The following CLI interaction shows how to view the status of the current allocation requests.

```
admin@ncs# show resouce-pools
```

Table Table 3, “Subnet allocation” shows that a subnet with a CIDR of 30 has been allocated from the subnet `10.0.0.0/24` in `pool1`.

Table 3. Subnet allocation

NAME	ID	ERROR	SUBNET	FROM
pool1	a1	-	10.0.0.0/30	10.0.0.0/24

Automatic redeployment of service

An allocation request may contain references to services that are to be redeployed whenever the status of the allocation changes. The following status changes trigger redeployment.

- Allocation response goes from no case to some case (`ok` or `error`)
- Allocation response goes from one case to the other
- Allocation response case stays the same but the leaves within the case change. Typically because a reallocation was triggered by configuration changes in the IP pool.

The service references are set in the `allocating-service` leaf-list, for example

```
admin@ncs# resource-pools ip-address-pool pool1 allocation a1 allocating-service \
             /services/vl:loop[name='myservice'] username myuser request subnet-size 30
```

Security

The `NSO IP Address Allocator` requires a username to be configured by the service applications when creating an allocation request. This username will be used to re-deploy the service applications once a resource has been allocated. Default NACM rules denies all standard users access to the `/ralloc:resource-pools` list. These default settings are provided in the (`initial_data/aaa_init.xml`) file of the `resource-manager` package.

Alarms

There are two alarms associated with the IP Address Allocator:

Empty alarm

This alarm is raised when the pool is empty, there are no availabe IPs that can be allocated.

Low threshold reached alarm

This alarm is raised when the pool is nearing empty, e.g. there is only ten percent or less separate IPs left in the pool.

ip-allocator-tool Action

A set of Debug and Data tools contained in rm-action/ip-allocator-tool Action is available to help admin or support to operate on RM data . Two parameters in the ip-allocator-tool Action can be provided: `operation`, `pool`. All the process info and result will be logged in ncs-java-vm.log, and the action itself just return result `Done!` Here is list of the valid operation value for the ip-allocator-tool Action

- fix_response_ip: scan the ipPool, and check if the allocation contains invalid allocation request id, release the allocation from ipPool if found. It happens for sync allocation when device configuration fail after a successful ip allocation and then cause a service transaction fail. This leaves the ipPool contains success allocated ip while allocation request response doesn't exist
- printIpPool: print current ipPool data in the ncs-java-vm.log for debug purpose.

Action usage example

Note that when pool parameter is provided , the operation will be on this specific ipPool.

```
admin@ncs>  unhide debug

admin@ncs>  request rm-action ip-allocator-tool operation fix_response_ip pool multiService

admin@ncs>  request rm-action ip-allocator-tool operation printIpPool pool multiService
```

CHAPTER 4

The NSO Resource Manager Data Models

Resource allocator model

Example 4. Resource allocator YANG Model

```
module resource-allocator {
  namespace "http://tail-f.com/pkg/resource-allocator";
  prefix "ralloc";

  import tailf-common {
    prefix tailf;
  }

  import ietf-inet-types {
    prefix inet;
  }

  organization "Tail-f Systems";
  description
   "This is an API for resource allocators.

    An allocation request is signaled by creating an entry in the
    allocation list.

    The response is signaled by writing a value in the response
    leave(s). The responder is responsible for re-deploying the
    allocating owners after writing the result in the response
    leaf.

    We expect a specific allocator package to do the following:

      1. Subscribe to changes in the allocation list and look for
         create operations.
      2. Perform the allocation and respond by writing the result
         into the response leaf, and then invoke the re-deploy
         action of the services pointed to by the owners leaf-list.

    Most allocator packages will want to annotate this model with
    additional pool definition data.";
```

```
revision 2022-03-11 {
    description
     "support multi-service and synchronous allocation request.";
 }

revision 2020-07-29 {
  description
    "1.1
    Enhancements:
     - Add 'redeploy-type' option for service redeploy action.
        If not provided, the 'default' is assumed, where the
        redeploy type is chosen based on NSO version.
    ";
}

 revision 2015-10-20 {
   description
    "Initial revision.";
}

grouping resource-pool-grouping {
  leaf name {
    type string;
    description
      "The name of the pool";
    tailf:info "Unique name for the pool";
  }
  leaf sync-dryrun {
    tailf:hidden debug;

    config false;
    type empty;
  }
  list allocation {
    tailf:info "contains all the details of a resource request made from user";
    key id;

    leaf id {
      type string;
    }

    leaf username {
      description
        "Authenticated user for invoking the service";
      type string;
      mandatory true;
    }

    leaf-list allocating-service {
      type instance-identifier {
        require-instance false;
      }
      description
        "Points to the services that own the resource.";
      tailf:info "Instance identifiers of services that own resource";
    }

    leaf sync-alloc {
      tailf:hidden debug;
      tailf:info "process allocation in synchronous flow";
      type empty;
```

```
    }

    leaf redeploy-type {
      description "Service redeploy type:
                   default, touch, reactive-re-deploy, re-deploy.";
      type enumeration {
        enum "default";
        enum "touch";
        enum "reactive-re-deploy";
        enum "re-deploy";
      }
      default "default";
    }

    container request {
      description
        "When creating a request for a resource the
         implementing package augments here.";
    }

    container response {
      config false;
      tailf:cdb-oper {
        tailf:persistent true;
      }
      choice response-choice {
        case error {
          leaf error {
            type string;
            description
              "Text describing why the allocation request failed";
          }
        }
        case ok {
        }
      }
      description
        "The response to the allocation request.";
    }
  }
}

container resource-pools {
}

container rm-action {
  tailf:action sync-alloc {
    tailf:hidden debug;
    tailf:actionpoint sync-alloc-action;
    input {
        leaf pool {
          type string;
        }
        leaf allocid {
          type string;
        }
        leaf user {
          type string;
        }
        leaf cidrmask {
          type int8;
        }
```

```
        leaf invertcidr {
          type boolean;
        }
        leaf owner {
          type string;
        }
        leaf subnetstartip {
          type string;
        }
        leaf dryrun {
          type boolean;
          default false;
        }
      }
    output {
      leaf allocated {
        type string;
        mandatory true;
      }
      leaf subnet {
        type string;
      }
    }
  }

tailf:action sync-alloc-id {
    tailf:actionpoint sync-alloc-id-action;
    input {
      leaf pool {
        type string;
      }
      leaf allocid {
        type string;
      }
      leaf user {
        type string;
      }
      leaf owner {
        type string;
      }
      leaf requestedId {
        type int32;
      }
   leaf method{
    type string;
    default "firstfree";
   }
   leaf sync {
      type boolean;
      default false;
   }
      leaf dryrun {
        type boolean;
        default false;
      }
    }
    output {
      leaf allocatedId {
        type string;
        mandatory true;
      }
    }
```

```
    }

  }
}
```

Id allocator model

Example 5. Id allocator YANG Model

```
module id-allocator {

  namespace "http://tail-f.com/pkg/id-allocator";
  prefix idalloc;

  import tailf-common {
    prefix tailf;
  }

  import resource-allocator {
    prefix ralloc;
  }

  include id-allocator-alarms {
    revision-date "2017-02-09";
  }

  organization "Tail-f Systems";
  description
    "This module contains a description of an id allocator for defining pools
     of id:s. This can for instance be used when allocating VLAN ids.
     This module contains configuration schema of the id allocator. For the
     operational schema, please see the id-allocator-oper module.";

  revision 2023-11-16 {
      description
       "Add action id-allocator-tool.";
   }
  revision 2022-03-11 {
      description
       "support multi-service and synchronous allocation request.";
   }

  revision 2017-08-14 {
    description
      "2.2
      Enhancements:
       Removed 'disable', add 'enable' for alarms.
       This means that if you want alarms you need to enable this explicitly
       now.
      ";
  }

  revision 2017-02-09 {
    description
      "2.1
      Enhancements:
       Added support for alarms
      ";
  }

  revision 2015-12-28 {
```

```
   description "2nd revision. Added support for allocation methods.";
}

revision 2015-10-20 {
   description "Initial revision.";
}

grouping range-grouping {
  leaf start {
    type uint32;
    mandatory true;
  }
  leaf end {
    type uint32;
    mandatory true;
    must ". >= ../start" {
      error-message "range end must be greater or equal to range start";
      tailf:dependency "../start";
    }
  }
}

// This is the interface
augment "/ralloc:resource-pools" {
  list id-pool {
    key "name";
    container range {
      description "The range the resource-pool should contain";
      uses range-grouping;
    }
    list exclude {
      tailf:info "list of id resource not available for allocation ";
      key "start end";
      leaf stop-allocation {
        type boolean;
        default "false";
      }
      uses range-grouping;
      tailf:cli-suppress-mode;
    }
    uses ralloc:resource-pool-grouping {
      augment "allocation/response/response-choice/ok" {
        leaf id {
          type uint32;
        }
      }
    }
    container alarms {
      leaf enabled {
        type empty;
        description "Set this leaf to enable alarms";
      }
      leaf low-threshold-alarm {
        type uint8 {
          range "0 .. 100";
        }
        default 10;
        description "Change the value for when the low threshold alarm is
                     raised. The value describes the percentage IDs left in
                     the pool. The default is to raise the alarm when there
                     are ten (10) percent IDs left in the pool.";
```

```
      }
    }
    description "The state of the id-pool.";
    tailf:info "Id pool";
  }
}

//augmenting the request/responses form resource-manager
augment "/ralloc:resource-pools/id-pool/allocation/request" {
  leaf sync {
    type boolean;
    default "false";
    description "Synchronize allocation with all other allocation
                 with same allocation id in other pools";
    tailf:info "Synchronize allocation id with other pools";
  }
  leaf id {
    type uint32;
    description "The specific id to sync with";
    tailf:info "Request a specific id";
  }
  container method {
    choice method {
      default firstfree;
      case firstfree {
        leaf firstfree {
          type empty;
          description "The default method to allocating a new id
                       is using the first free method. Using this
                       allocation method might mean that an id is reused
                       quickly which might not be what one wants nor is
                       supported in lower layers.";
          tailf:info "Default method used to request a new id.";
        }
      }
      case roundrobin {
        leaf roundrobin {
          type empty;
          description "Pick the next available id using a round
                         robin approach. Earlier used id:s will not be
                         reused until the range is exhausted and allocation
                         restarts from the start of the range again.

                      Note that sync will override round robin.";
          tailf:info "Round robin method used to request a new id.";
        }
      }
    }
  }
}

augment "/ralloc:rm-action" {
    tailf:action id-allocator-tool {
    tailf:hidden debug;
    tailf:actionpoint id-allocator-tool-action;
    input {
      leaf pool {
        type leafref {
            path "/ralloc:resource-pools/idalloc:id-pool/name";
        }
      }
      leaf operation{
```

```
            type enumeration {
                enum printIdPool;
                enum check_missing_report;
                enum fix_missing_allocation_id;
                enum fix_missing_owner;
                enum fix_missing_allocation;
                enum fix_response_id;
                enum persistAll;
            }

            mandatory true;
          }
        }
        output {
          leaf result {
            type string;
          }
        }
      }
    }
}
```

IP address allocator model

Example 6. IP address allocator YANG Model

```
module ipaddress-allocator {

  namespace "http://tail-f.com/pkg/ipaddress-allocator";

  prefix ipalloc;

  import tailf-common {
    prefix tailf;
  }

  import ietf-inet-types {
    prefix inet;
  }

  import resource-allocator {
    prefix ralloc;
  }

  include ipaddress-allocator-alarms {
    revision-date "2017-02-09";
  }

  organization "Tail-f Systems";
  description
    "This module contains a description of an IP address allocator for defining
     pools of IPs and allocating addresses from these.
     This module contains configuration schema of the ip allocator. For the
     operational schema, please see the ip-allocator-oper module.";

  revision 2022-03-11 {
      description
       "support multi-service and synchronous allocation request.";
   }

  revision 2018-02-27 {
```

```
  description
    "Introduce the 'invert' field in the request container that enables
     one to allocate the same size network regardless of the network
     type (IPv4/IPv6) in a pool by using the inverted cidr.";
}

revision 2017-08-14 {
  description
    "2.2
    Enhancements:
     Removed 'disable', add 'enable' for alarms.
     This means that if you want alarms you need to enable this explicitly
     now.
    ";
}

revision 2017-02-09 {
  description
    "1.2
    Enhancements:
     Added support for alarms
    ";
}

revision 2016-01-29 {
  description
    "1.1
    Enhancements:
     Added support for defining pools using IP address ranges.
    ";
}

revision 2015-10-20 {
  description "Initial revision.";
}

// This is the interface
augment "/ralloc:resource-pools" {
  list ip-address-pool {
    tailf:info "IP Address pools";
    key name;

    uses ralloc:resource-pool-grouping {
      augment "allocation/request" {
        leaf subnet-size {
          tailf:info "Size of the subnet to be allocated.";
          type uint8 {
            range "1..128";
          }
          mandatory true;
        }

        leaf subnet-start-ip {
          description
            "Optional parameter used to request for a particular IP for
            the allocation (instead of the first available subnet matching
            the subnet-size). The subnet-start-ip has to be the first IP
            in the range defined by subnet-size, otherwise the allocation will
            fail. Ex: subnet-start-ip 10.0.0.36 subnet-size 30 gives the
            equivalent range 10.0.0.36-10.0.0.39, and it's a valid value.
            subnet-start-ip 10.0.0.36 subnet-size 29 gives the range
            10.0.0.32-10.0.0.39 and it's NOT a valid option.";
```

```
      type inet:ip-address;
    }

    leaf invert-subnet-size {
      description
        "By default subnet-size is considered equal to the cidr, but by
         setting this leaf the subnet-size will be the \"inverted\" cidr.
         I.e: If one sets subnet-size to 8 with this leaf unset 2^24
         addresses will be allocated for a IPv4 pool and in a IPv6 pool
         2^120 addresses will be allocated. By setting this leaf only
         2^8 addresses will be allocated in either a  IPv4 or a IPv6
         pool.";
      type empty;
    }
  }
  augment "allocation/response/response-choice/ok" {
    leaf subnet {
      type inet:ip-prefix;
    }
    leaf from {
      type inet:ip-prefix;
    }
  }
}

leaf auto-redeploy {
  tailf:info "Automatically re-deploy services when an IP address is "
    +"re-allocated";
  type boolean;
  default "true";
}

list subnet {
  key "address cidrmask";
  tailf:cli-suppress-mode;
  description
    "List of subnets belonging to this pool. Subnets may not overlap.";

  must "(contains(address, '.') and cidrmask <= 32) or
        (contains(address, ':') and cidrmask <= 128)" {
    error-message "cidrmask is too long";
  }

  tailf:validate ipa_validate {
    tailf:dependency ".";
  }

  leaf address {
    type inet:ip-address;
  }

  leaf cidrmask {
    type uint8 {
      range "1..128";
    }
  }
}

list exclude {
  key "address cidrmask";
  tailf:cli-suppress-mode;
  description "List of subnets to exclude from this pool. May only "
```

```
      +"contains elements that are subsets of elements in the list of "
      +"subnets.";
    tailf:info "List of subnets to exclude from this pool. May only "
      +"contains elements that are subsets of elements in the list of "
      +"subnets.";

    must "(contains(address, '.') and cidrmask <= 32) or
          (contains(address, ':') and cidrmask <= 128)" {
      error-message "cidrmask is too long";
    }

    tailf:validate ipa_validate {
      tailf:dependency ".";
    }

    leaf address {
      type inet:ip-address;
    }

    leaf cidrmask {
      type uint8 {
        range "1..128";
      }
    }
  }

  list range {
    key "from to";
    tailf:cli-suppress-mode;
    description
      "List of IP ranges belonging to this pool, inclusive. If your "
      +"pool of IP addresses does not conform to a convenient set of "
      +"subnets it may be easier to describe it as a range. "
      +"Note that the exclude list does not apply to ranges, but of "
      +"course a range may not overlap a subnet entry.";

    tailf:validate ipa_validate {
      tailf:dependency ".";
    }

    leaf from {
      type inet:ip-address-no-zone;
    }

    leaf to {
      type inet:ip-address-no-zone;
    }

    must "(contains(from, '.') and contains(to, '.')) or
          (contains(from, ':') and contains(to, ':'))" {
      error-message
        "IP addresses defining a range must agree on IP version.";
    }
  }
  container alarms {
    leaf enabled {
      type empty;
      description "Set this leaf to enable alarms";
    }

    leaf low-threshold-alarm {
      type uint8 {
```

```
            range "0 .. 100";
          }
          default 10;
          description  "Change the value for when the low threshold alarm is
                       raised. The value describes the percentage IPs left in
                       the pool. The default is to raise the alarm when there
                       are ten (10) percent IPs left in the pool.";
        }
      }
    }
  }

  augment "/ralloc:rm-action" {
      tailf:action ip-allocator-tool {
      tailf:hidden debug;
      tailf:actionpoint ip-allocator-tool-action;
      input {
        leaf pool {
          type leafref {
              path "/ralloc:resource-pools/ipalloc:ip-address-pool/name";
          }
        }
        leaf operation{
            type enumeration {
                enum printIpPool;
                enum fix_response_ip;
            }

            mandatory true;
        }
      }
      output {
        leaf result {
          type string;
        }
      }
    }
  }
}
```

CHAPTER 5

Resources

References for further reading

NSO Packages chapter in NSO 6.3 Administration Guide.

The AAA Infrastructure chapter in NSO 6.3 Administration Guide.