NSO Layered Service Architecture

Release: NSO 5.7.1

Published: October 9, 2016

Last Modified: January 26, 2022

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

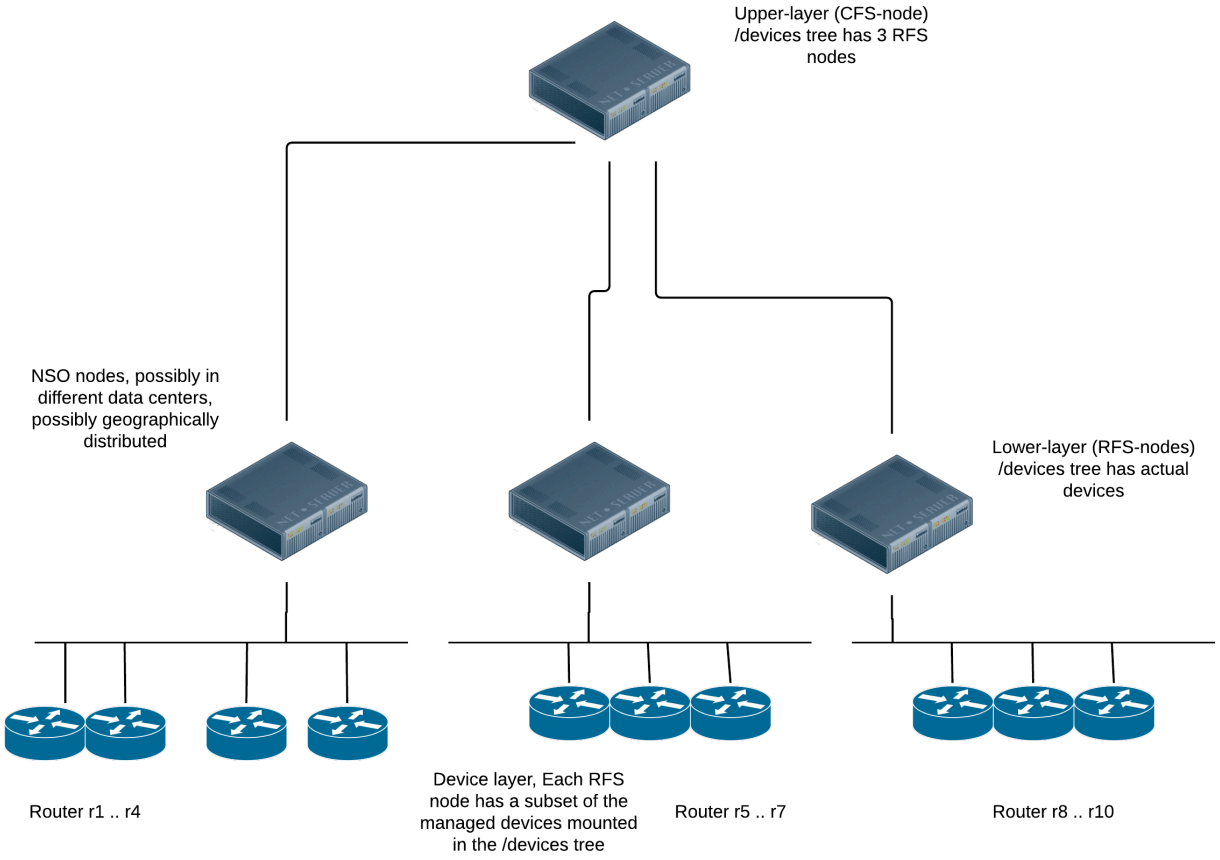
Going big - Layered service architecture

Introduction to the layered service architecture

This section describes how to design massively large and scalable NSO applications. Large service providers/enterprises want to use NSO to manage services for millions of subscribers/users, ranging over several hundred thousand managed devices. To achieve this, you can design your services in the layered fashion described here. We call this the Layered Service Architecture, or LSA for short.

The basic idea is to split a service into an upper layer and one or more lower layer parts. This can be viewed as splitting the service into a customer-facing (CFS) and a resource-facing (RFS) part. The CFS code (upper-layer) runs in one (or several) NSO cfs-nodes, and the RFS code (lower-layer) runs in one of many NSO rfs-nodes. The rfs-nodes have each a portion of the managed devices mounted in their `/devices` tree and the cfs-node(s) have the NSO rfs-nodes mounted in their `/devices` tree.

Figure 1. Layered CFS/RFS architecture



The main advantage of this architecture is that we can add arbitrarily many device nodes. The major problems with monolithic NSO applications are memory and provisioning throughput constraints. Especially the device configurations can sometimes become very large. This architecture attempts to address both of these problems as will be shown in the sections below.

Separating the service into CFS and RFS

Depending on the situation, the separation can be done in different ways. We have at least the following three different scenarios: new green field design, existing stacked services and existing monolithic design.

New design - green field

If you are starting the service design from scratch, it's obviously ideal. In this case you can choose the partitioning at leisure. The CFS must obviously contain YANG definitions for everything the customer, or an order capture system north of the service node can enter and order. However, the RFS YANG models can be designed differently. They are now system internal data models, and you as a designer are free to design them in such a way so that it makes the provisioning code as easy as possible.

We have two variants on this design, one where one CFS node is used to provision different kinds of services, and one where we also split up the CFS node into multiple nodes, for example one CFS node per service type. This latter design caters for even larger systems that can grow horizontally also at the top level of the system.

A VPN application for example could be designed to have two CFS YANG models. One for the infra-structure and then one for a CPE site. The customer buys the infra-structure, and then buys additional CPE sites (legs on the VPN). We can then design the RFS models at will, maybe divide the results of an instantiated cpe-site into 3 separate RFS models, maybe depending on input parameters to the CFS, or some other configuration the CFS code can choose to instantiate a physical CPE or a virtual CPE, i.e., the CFS code chooses where to instantiate which RFSs.

Existing monolithic application

A common use case is when a single node NSO installation grows and we're faced with performance problems due to growth and size. It is possible to split up a monolithic application into an upper-layer and lower-layer service, and we show how to do that in the following chapters. However, the decision to do that should always be accompanied by a thorough analysis determining what makes the system too slow. Sometimes, it's just something trivial, like a bad "must" expression in service YANG code or something similar. Fixing that is clearly easier than re-architecting the application.

Existing monolithic application with stacked services

Existing monolithic application that are written using the stacked services design can sometimes be easy to rewrite in the LSA fashion. The division of a service into an upper and lower layer is already done, where the stacked services make ideal candidates for lower layer (RFS) services.

Dispatching

Regardless of whether we have a green field design, or if we are extending an existing monolithic application, we always face the same problem of dispatching the RFS instantiation to the correct lower-layer NSO node.

Imagine a VPN application where the customer orders yet another leg in the VPN. This will (at least) result in one additional managed device, the CPE. That CPE resides in the `/devices/device` tree on one, and only one of the RFS-nodes. The CFS-node must thus:

- Figure out which RFS-node is responsible for that CPE.
- Dispatch the RFS instantiation to that particular RFS-node.

Techniques to facilitate this dispatch are:

- This first and most straightforward solution to the dispatch problem is to maintain a mapping list at the CFS-node(s). That list will contain 2-tuples that map device name to RFS-node. One downside to this is clearly the fact that the list must be maintained. Whenever the `/devices/device` is manipulated at one RFS-node, the mapping list at the CFS-node must also updated.

  It is straightforward to automate the maintenance of the list though, either through NETCONF notifications whenever `/devices/device` is manipulated, or, alternatively, by explicitly asking the CFS-node to query the RFS-nodes for its list of devices.
- Another scenario is when the RFS-nodes are geographically separated, different countries, different data centers. If the CFS service instance contains a parameter indicating which country/data center is to be used, the dispatching of the RFS can be inferred or calculated at the CFS layer. This way, no mapping needs to be maintained at all.

Scalability and performance aspects

This architecture scales horizontally at the RFS-node layer. Each RFS-node needs to host the RFSs that touch the devices it has in its `/devices/device` tree.

If one RFS node starts to become overloaded, it's easy to start an additional RFS node for e.g., that geographical region, or that data center, thus catering for horizontal scalability at the level of number of managed devices, and number of RFS instances.

As stated above, the main disadvantages to a monolithic application design are memory consumption and overall throughput. Memory is cheap, so unless the throughput is the major concern, we generally recommend staying with the monolithic application design. However, an LSA design can greatly improve overall throughput, especially in combination with the commit queue. In general we recommend to enable the commit queue in LSA applications. If the commit queue is not enabled, overall throughput is still limited by the slowest device on the network, at least for applications that do not use reactive FastMap.

Let's go though the execution scenario for a set of LSA configurations. The first is the default, no commit queue, and also a standard FastMap application with no Reactive FastMap at all. This is common for many VPN type of applications with NSO. A prerequisite for the discussion below is for the reader to understand the operational behavior of the commit queues as well as the difference between a regular FastMap application and a Reactive FastMap application.

1. A transaction arrives at the CFS node from the northbound, maybe from a customer order portal.
2. The CFS code determines which RFSs need to be instantiated where, NSO will send the appropriate NETCONF edit-config RPCs to the RFS nodes. These are synchronous.
3. The chosen RFS nodes get the edit-config RPCs. The RFS FastMap code runs at the RFS nodes, manipulating the `/devices/device` tree, i.e., updating the managed devices.
4. The configuration change is accepted by the choosen devices, and the RFS-node transactions return, thereby replying to the CFS-node, synchronously.
5. The reply to the initial northbound request from the customer portal is sent by the CFS node.

The entire sequence is serialized though the system as a whole. This means that if another northbound request arrives at the CFS node while the first request is being processed, the second request is synchronously queued at the CFS node, waiting for the currently running transaction to either succeed or fail.

If we enable the commit queue between the CFS node and the RFS nodes, we may achieve some increased parallelism in the system, and thus higher overall system throughput. Not much though; here is the execution sequence with commit queue enabled between CFS-node and RFS-nodes

1. A transaction arrives at the CFS node from the northbound, maybe from a customer order portal.
2. The CFS code determines which RFSs need to be instantiated where, NSO will send the appropriate NETCONF edit-config RPCs to the RFS nodes, the transaction is finished. Depending on commit flags (as requested by the northbound caller), a reply is either sent now, or the reply is delayed until all items in the commit queue generated by this transaction are finished.
3. A second request arrives from the northbound, while the first one is being processed by the commit queue. The CFS code determines which RFSs need to be instantiated where. The "where" in the sentence above, means "which RFS nodes" need to be touched. If the set of chosen RFS nodes is completely disjoint from the set of RFS nodes chosen for the first transaction, the second one gets to execute in parallel, if not, it gets queued. Thus just enabling the commit queue between the CFS node and the RFS nodes doesn't buy us a lot of parallelism.

What is required to achieve really high transaction throughput, is to not allow the slowest device to lock up the system. This can be achieved in two radically different ways. Either enable the commit queue between the RFS-nodes and their managed devices, or alternatively, if the RFS FastMap code is reactive, we almost get the same behavior, but not quite. We'll walk through both scenarios. First, enable the commit queue everywhere, we get:

1. A transaction arrives at the CFS node from the northbound, maybe from a customer order portal.
2. The CFS FastMap code determines which RFSs need to be instantiated where, NSO will send the appropriate NETCONF edit-config RPCs to the RFS nodes, the transaction is finished, no need to wait for the RFS nodes to reply.
3. The RFS nodes receive the `edit-config` RPC, and the FastMap code is invoked. The RFS FastMap code determines which devices need to be configured with what data, the proposed router configs are sent to the commit queue and the transaction is finished.

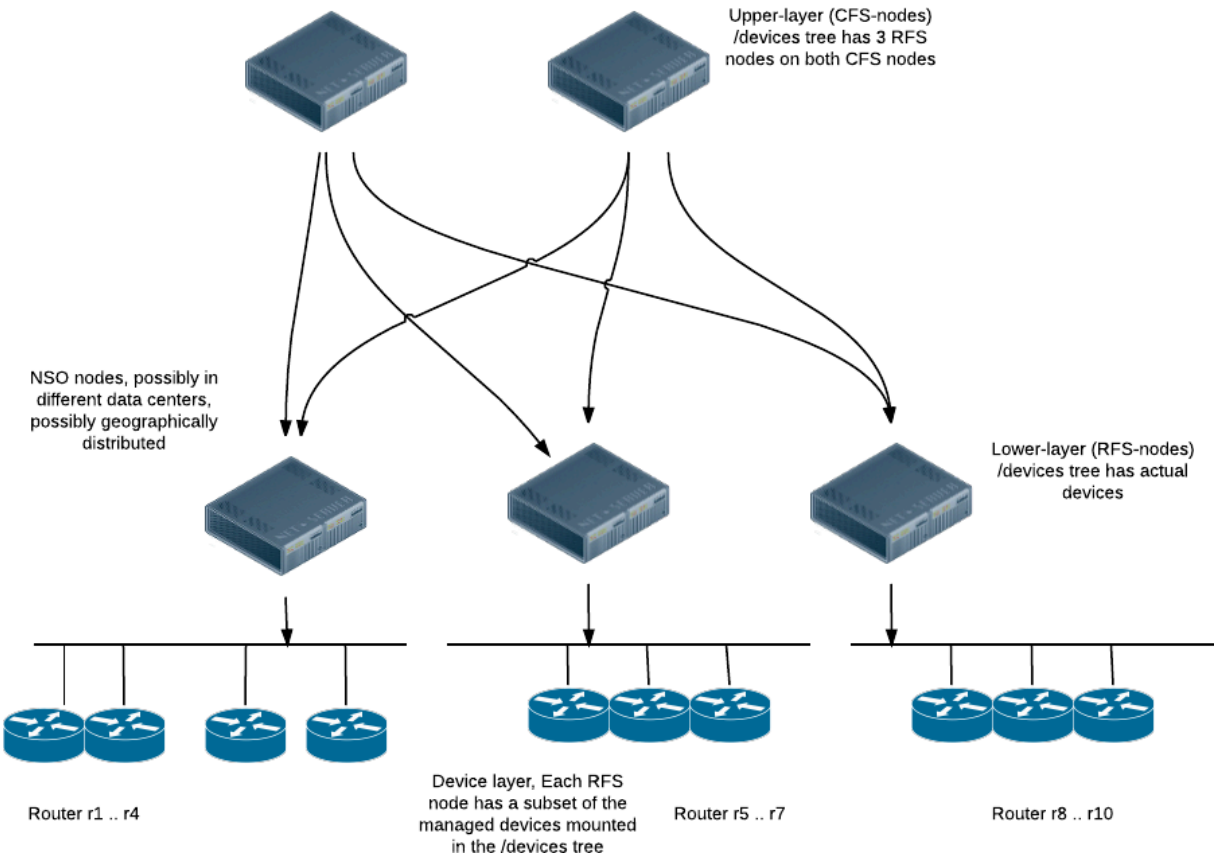
Regardless of when additional data is received from the northbound of the CFS node, data is processed and the level of concurrency in a system configured this way is very high.

If the RFS nodes FastMap code is reactive, we get similar behavior since usually Reactive FastMap applications are very fast during its first round of execution. Usually something must be allocated, an IP address or something, thus when the RFS node receives the `edit-config` from the CFS node and the RFS node FastMap code is invoked, that first round of Reactive FastMap execution returns fast, without waiting for any provisioning of any actual devices, thus the CFS node `edit-config` executes fast. We will not achieve equally high provisioning numbers as with the commit queue though, since eventually when devices actually has to modified, execution is serialized.

Multiple CFS nodes

It's possible to scale the CFS layer horizontally as well.

Figure 2. Layered CFS/RFS architecture - multiple CFS nodes



Typically we would assign a separate CFS node per application, or group of applications. Maybe a provider sells L2 and L3 VPNs. One CFS node could be running the CFS provisioning code for L2 VPNs and another CFS node the L3 VPNs.

The RFS nodes though would all have to run the RFS code for all services. Typically the set of service YANG modules related to L2 VPNs would be completely disjoint from the set of services YANG modules related to L3 VPNs. Thus we do not have any issues with multiple managers (CFS nodes). The CFS node for L2 service would see and manipulate the RFSs for L2 RFS code and vice versa for the L3 code. Since there is no overlap between what the two (or more) CFS nodes touch on the RFS nodes, we do not have a sync problem. We must however turn off the sync check between the CFS nodes and the RFS nodes since the sync check is based on CDB transaction ids. The following execution scenario explains this:

1. CFS node for L2 VPNs execute a transaction towards RFS node R1.
2. CFS node for L3 VPNs execute a transaction towards RFS node R1.
3. CFS node for L2 VPNs is now out of sync, the other CFS node touched entirely different data, but that still doesn't matter since the CDB transaction counter has been changed.

If there is overlapping configuration between two CFS applications, the code has to run on the same CFS node and it's not allowed to have multiple northbound configuration system manipulate exactly the same data. In practice this not a problem though.

Pros and cons of layered service architecture

Pros

Clearly the major advantages of this architecture are related to scalability. The solution scales horizontally, both at the upper and the lower layer, thus catering for truly massive deployments.

Another advantage not previously mentioned in this chapter is upgradability. It's possible to upgrade the RFS nodes one at a time. Also, and more importantly is that if the YANG upgrade rules are followed, the CFS node and the RFS nodes can have different release cycles. If a bug is found or a feature is missing in the RFS nodes, that can be fixed independent of the CFS node. Furthermore, if the architecture with multiple CFS nodes is used, the different CFS nodes are probably programmed by separate teams, and the CFS nodes can be upgraded independently of each other.

Cons

Compared to a provisioning system where we run everything on a single monolithic NSO system, we get increased complexity. This is expected.

Dividing a provisioning application into upper and lower layer services also increase the complexity of the application itself. Also, in order to follow the execution of a reactive FastMap RFS, typically additional NETCONF notification code has to be written. These notifications have to be sent from the RFS nodes, and received and processed by the CFS code. If something goes wrong at the device layer, this information has to be conveyed all the way to the top level of the system.

We do not get a "single pane of glass" view of all managed interfaces on any one node. Managed devices are spread out over independent RFS nodes.

CHAPTER 2

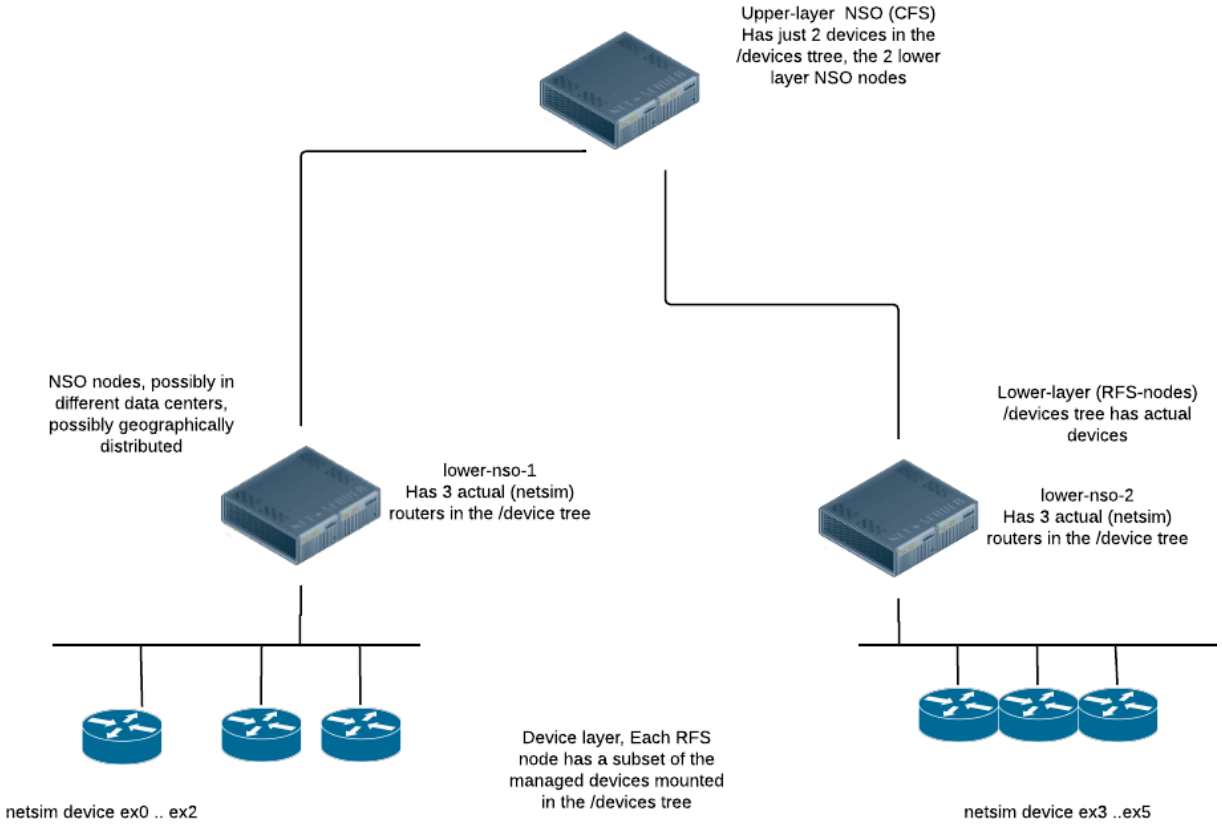
LSA examples

Greenfield LSA application

In this section we'll describe a very small LSA application. The application we describe exists as a running example under: `examples.ncs/getting-started/developing-with-ncs/22-layered-service-architecture`

This example application is a slight variation on the `examples.ncs/getting-started/developing-with-ncs/4-rfs-service` where the YANG code has been split up into an upper-layer and a lower-layer implementation. The example topology (based on netsim for the managed devices, and NSO for the upper/lower layer NSO instances) looks like:

Figure 3. Example LSA architecture



The upper layer of the YANG service data for this example looks as:

```
module cfs-vlan {
  ...
  list cfs-vlan {
    key name;
    leaf name {
      type string;
    }

    uses ncs:service-data;
    ncs:servicepoint cfs-vlan;

    leaf a-router {
      type leafref {
        path "/dispatch-map/router";
      }
      mandatory true;
    }
    leaf z-router {
      type leafref {
        path "/dispatch-map/router";
      }
      mandatory true;
    }
    leaf iface {
      type string;
      mandatory true;
    }
```

```
    leaf unit {
      type int32;
      mandatory true;
    }
    leaf vid {
      type uint16;
      mandatory true;
    }
  }
}
```

Instantiating one CFS we have:

```
admin@upper-nso% show cfs-vlan
cfs-vlan v1 {
    a-router ex0;
    z-router ex5;
    iface    eth3;
    unit     3;
    vid      77;
}
```

The provisioning code for this CFS has to make a decision on where to instantiate what. In this example the "what" is trivial, it's the accompanying RFS, whereas the "where" is more involved. The two underlying RFS nodes, each manage 3 netsim routers, thus the given the input, the CFS code must be able to determine which RFS node to choose. In this example we have chosen to have an explicit map, thus on the `upper-nso` we also have:

```
admin@upper-nso% show dispatch-map
dispatch-map ex0 {
    rfs-node lower-nso-1;
}
dispatch-map ex1 {
    rfs-node lower-nso-1;
}
dispatch-map ex2 {
    rfs-node lower-nso-1;
}
dispatch-map ex3 {
    rfs-node lower-nso-2;
}
dispatch-map ex4 {
    rfs-node lower-nso-2;
}
dispatch-map ex5 {
    rfs-node lower-nso-2;
}
```

So, we have template CFS code which does the dispatching to the right RFS node.

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
                 servicepoint="cfs-vlan">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <!-- Do this for the two leafs a-router and z-router -->
    <?foreach {a-router|z-router}?>
    <device>
      <!--
      Pick up the name of the rfs-node from the dispatch-map
      and do not change the current context thus the string()
      -->
      <name>{string(deref(current())/../rfs-node)}</name>
      <config>
```

```
        <vlan xmlns="http://com/example/rfsvlan">
          <!-- We do not want to change the current context here either -->
          <name>{string(/name)}</name>
          <!-- current() is still a-router or z-router -->
          <router>{current()}</router>
          <iface>{/iface}</iface>
          <unit>{/unit}</unit>
          <vid>{/vid}</vid>
          <description>Interface owned by CFS: {/name}</description>
        </vlan>
      </config>
    </device>
    <?end?>
  </devices>
</config-template>
```

This technique for dispatching is simple, and easy to understand. The dispatching might be more complex, it might even be determined at execution time dependent on CPU load. It might be (as in this example) inferred from input parameters or it might be computed.

The result of the template based service is to instantiate the RFS, at the RFS nodes.

First lets have a look at what happened in the upper-nso. Look at the modifications but ignore the fact that this is an LSA service:

```
admin@upper-nso% request cfs-vlan v1 get-modifications no-lsa
cli {
    local-node {
        data  devices {
                    device lower-nso-1 {
                        config {
              +             rfs-vlan:vlan v1 {
              +                 router ex0;
              +                 iface eth3;
              +                 unit 3;
              +                 vid 77;
              +                 description "Interface owned by CFS: v1";
              +             }
                        }
                    }
                    device lower-nso-2 {
                        config {
              +             rfs-vlan:vlan v1 {
              +                 router ex5;
              +                 iface eth3;
              +                 unit 3;
              +                 vid 77;
              +                 description "Interface owned by CFS: v1";
              +             }
                        }
                    }
                }
    }
}
```

Just the dispatched data is shown. As ex0 and ex5 resides on different nodes the service instance data has to be sent to both lower-nso-1 and lower-nso-2.

Now lets see what happened in the lower-nso. Look at the modifications and take into account these are LSA nodes (this is the default):

```
admin@upper-nso% request cfs-vlan v1 get-modifications
```

```
cli {
  local-node {
    .....
  }
  lsa-service {
    service-id /devices/device[name='lower-nso-1']/config/rfs-vlan:vlan[name='v1']
    data devices {
      device ex0 {
        config {
          r:sys {
            interfaces {
   +          interface eth3 {
   +            enabled;
   +            unit 3 {
   +              enabled;
   +              description "Interface owned by CFS: v1";
   +              vlan-id 77;
   +            }
   +          }
            }
          }
        }
      }
    }
  }
  lsa-service {
    service-id /devices/device[name='lower-nso-2']/config/rfs-vlan:vlan[name='v1']
    data devices {
      device ex5 {
        config {
          r:sys {
            interfaces {
   +          interface eth3 {
   +            enabled;
   +            unit 3 {
   +              enabled;
   +              description "Interface owned by CFS: v1";
   +              vlan-id 77;
   +            }
   +          }
            }
          }
        }
      }
    }
  }
```

Both the dispatched data and the modification of the remote service are shown. As ex0 and ex5 resides on different nodes the service modifications of the service rfs-vlan on both lower-nso-1 and lower-nso-2 are shown.

The communication between the NSO nodes is of course NETCONF.

```
admin@upper-nso% set cfs-vlan v1 a-router ex0 z-router ex5 iface eth3 unit 3 vid 78
[ok][2016-10-20 16:52:45]

[edit]
admin@upper-nso% commit dry-run outformat native
native {
    device {
        name lower-nso-1
        data <rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"
```

```
                 message-id="1">
              <edit-config xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
                <target>
                  <running/>
                </target>
                <test-option>test-then-set</test-option>
                <error-option>rollback-on-error</error-option>
                <with-inactive xmlns="http://tail-f.com/ns/netconf/inactive/1.0"/>
                <config>
                  <vlan xmlns="http://com/example/rfsvlan">
                    <name>v1</name>
                    <vid>78</vid>
                    <private>
                      <re-deploy-counter>-1</re-deploy-counter>
                    </private>
                  </vlan>
                </config>
              </edit-config>
            </rpc>
    }
              ...........
              ....
```

The YANG model at the lower layer, also know as the RFS layer, is similar, but different:

```
module rfs-vlan {

  ...

  list vlan {
    key name;
    leaf name {
      tailf:cli-allow-range;
      type string;
    }

    uses ncs:service-data;
    ncs:servicepoint "rfs-vlan";

    leaf router {
      type string;
    }
    leaf iface {
      type string;
      mandatory true;
    }
    leaf unit {
      type int32;
      mandatory true;
    }
    leaf vid {
      type uint16;
      mandatory true;
    }
    leaf description {
      type string;
      mandatory true;
    }
  }
}
```

and the task for the RFS provisioning code here is to actually provision the designated router. If we log into one of the lower layer NSO nodes, we can check.

```
admin@lower-nso-1> show configuration vlan
vlan v1 {
    router      ex0;
    iface       eth3;
    unit        3;
    vid         77;
    description "Interface owned by CFS: v1";
}
[ok][2016-10-20 17:01:08]
admin@lower-nso-1> request vlan v1 get-modifications
cli {
  local-node {
    data  devices {
             device ex0 {
               config {
                 r:sys {
                   interfaces {
    +                interface eth3 {
    +                  enabled;
    +                  unit 3 {
    +                    enabled;
    +                    description "Interface owned by CFS: v1";
    +                    vlan-id 77;
    +                  }
    +                }
                   }
                 }
               }
             }
    }
  }
}
```

To conclude this section, the final remark here is that to design a good LSA application, the trick is to identify a good layering for the service data models. The upper layer, the CFS layer is what is exposed northbound, and thus requires a model that is as forward looking as possible since that model is what system north of NSO integrates to, whereas the lower layer models, the RFS models can be viewed as "internal system models" and they can be more easily changed.

Greenfield LSA application designed for easy scaling

In this section we'll describe a lightly modified version of the example in the previous section. The application we describe here exists as a running example under: `examples.ncs/getting-started/developing-with-ncs/24-layered-service-architecture-scaling`

Sometimes it is desirable to be able to easily move devices from one lower LSA node to another. This makes it possible to easily expand or shrink the number of lower LSA nodes. Additionally, it is sometimes desirable to avoid HA-pairs for replication but instead use a common store for all lower LSA devices, such as a distributed data base, or a common file system.

The above is possible provided the LSA application is structured in certain ways.

- The lower LSA nodes only expose services that manipulate the configuration of a single device. We call these device RFSs, or dRFS for short.
- All services are located in a way that makes it easy to extract them, for example in /drfs:dRFS/device

```
container dRFS {
  list device {
    key name;
```

```
    leaf name {
      type string;
    }
  }
}
```

- No RFS takes place on the lower LSA nodes. This avoids the complication with locking and distributed event handling.
- The LSA nodes need to be set up with the proper NEDs, and with auth groups such that a device can be moved without having to install new NEDs or update auth groups.

Provided that the above requirements are met it is possible to move a device from one lower LSA node by extracting the configuration from the source node, and installing it on the target node. This, of course, requires that the source node is still alive, which is normally the case when HA-pairs are used.

An alternative to using HA-pairs for the lower LSA nodes is to extract the device configuration after each modification to the device, and store it in some central storage. This would not be recommended when high throughput is required but may make sense in certain cases.

In the example application there are two packages on the lower LSA nodes that provides this functionality. The package `inventory-updater` installs a database subscriber that is invoked every time any device configuration is modified, both in the prepare phase and in the commit phase of any such transaction. It extracts the device and dRFS configuration, including service meta data, during the prepare phase. If the transaction proceeds to a full commit, the package is again invoked and the extracted configuration is stored in a file in the directory `db_store`.

The other package is called `device-actions`. It provides three actions: `extract-device`, `install-device`, and `delete-device`. They are intended to be used by the upper LSA node when moving a device either from a lower LSA node, or from `db_store`.

In the upper LSA node there is one package for coordinating the movement, called `move-device`. It provides an action for moving a device from one lower LSA node to another. For example when invoked to move device ex0 from `lower-1` to `lower-2` using the action

```
request move-device move src-nso lower-1 dest-nso lower-2 device-name ex0
```

it goes through the following steps:

- A partial lock is acquired on the upper-nso for the path `/devices/device[name=lower-1]/config/dRFS/device[name=ex0]` to avoid any changes to the device while the device is in the process of being moved.
- The device and dRFS configuration is extracted in one of two ways:
  - Read the configuration from `lower-1` using the action

    ```
    request device-action extract-device name ex0
    ```
  - Read the configuration from some central store, in our case the file system in the directory. `db_store`.

  The configuration will look something like this

```
devices {
    device ex0 {
        address   127.0.0.1;
        port      12022;
        ssh {
        ...
           /* Refcount: 1 */
            /* Backpointer: [ /drfs:dRFS/drfs:device[drfs:name='ex0']/rfs-vlan:vlan[rfs-vlan
```

```
            interface eth3 {
            ...
            }
        ...
    }
}
dRFS {
    device ex0 {
        vlan v1 {
            private {
            ...
            }
        }
    }
}
```

- Install the configuration on the lower-2 node. This can be done by running the action:

  ```
  request device-action install-device name ex0 config <cfg>
  ```

  This will load the configuration and commit using the flags `no-deploy` and `no-networking`.
- Delete the device from `lower-1` by running the action

  ```
  request device-action delete-device name ex0
  ```
- Update mapping table

  ```
  dispatch-map ex0 {
      rfs-node lower-nso-2;
  }
  ```
- Release partial lock for `/devices/device[name=lower-1]/config/dRFS/device[name=ex0]`.
- Re-deploy all services that have touched the device. The services all have back pointers from `/devices/device{lower-1}/config/dRFS/device{ex0}`. They are `re-deployed` using the flags `no-lsa` and `no-networking`.
- Finally the action runs `compare-config` on `lower-1` and `lower-2`.

With this infrastructure in place it is fairly straightforward to implement actions for re-balancing devices among lower LSA nodes, as well as evacuating all devices from a given lower LSA node. The example contains implementations of those actions as well.

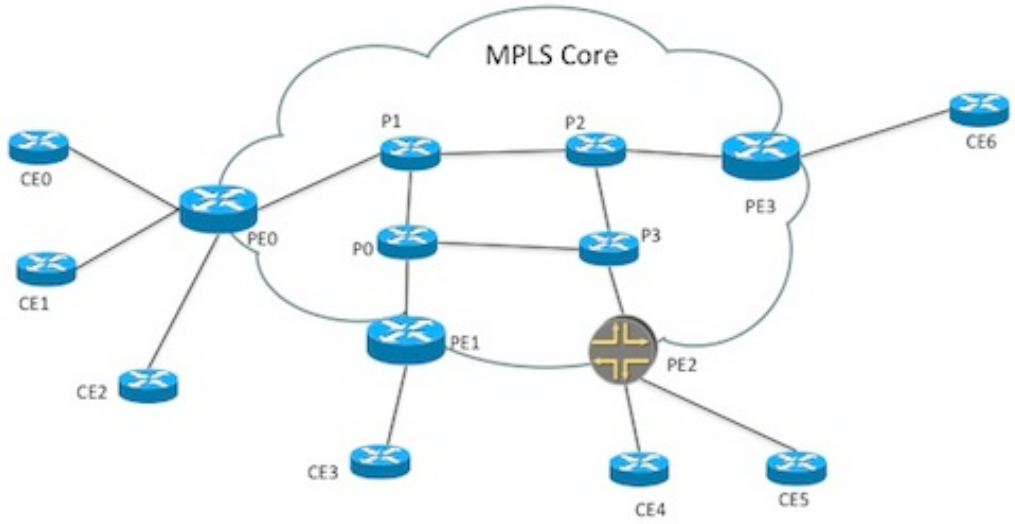
Rearchitecting an existing monolithic application into LSA

If we do not have the luxury of designing our NSO service application from scratch, but rather are faced with extending/changing an existing, already deployed application into the LSA architecture we can use the techniques described in this section.

Usually, the reasons for rearchitecting an existing application are performance related.

In the NSO example collection, one of the most popular real examples is the `examples.ncs/service-provider/mpls-vpn` code. That example contains an almost "real" VPN provisioning example whereby VPNS are provisioned in a network of CPEs, PEs and P routers according to this picture:

Figure 4. VPN network



The service model in this example, roughly looks like:

```
list l3vpn {
   description "Layer3 VPN";

   key name;
   leaf name {
     type string;
   }

   leaf route-distinguisher {
     description "Route distinguisher/target identifier unique for the VPN";
     mandatory true;
     type uint32;
   }

   list endpoint {
     key "id";
     leaf id {
       type string;
     }
     leaf ce-device {
       mandatory true;
       type leafref {
         path "/ncs:devices/ncs:device/ncs:name";
       }
     }

     leaf ce-interface {
       mandatory true;
       type string;
     }

     ....

     leaf as-number {
       tailf:info "CE Router as-number";
       type uint32;
```

```
    }
  }
  container qos {
    leaf qos-policy {
       ......
```

There are several interesting observations on this model code related to the Layered Service Architecture.

- Each instantiated service has a list of end points, CPE routers. These are modeled as a leafref into the /devices tree. This has to be changed if we wish to change this application into an LSA application since the /devices tree at the upper layer doesn't contain the actual managed routers, instead the /devices tree contains the lower layer RFS nodes.
- There is no connectivity/topology information in the service model, instead the `mpls-vpn` example has topology information on the side, and that data is used by the provisioning code. That topology information for example contains data on which CE routers are directly connected to which PE router.

  Remember from the previous section, one of the additional complications of an LSA application is the dispatching part. The dispatch problem fits well into the pattern where we have topology information stored on the side and let the provisioning FASTMAP code use that data to guide the provisioning. One straightforward way would be to augment the topology information with additional data, indicating which RFS node is used to manage a specific managed device.

By far the easiest way to change an existing monolithic NSO application into the LSA architecture, is to keep the service model at upper layer and lower layer almost identical, only changing things like leafrefs direct into the /devices tree which obviously breaks.

In this example the topology information is stored in a separate container `share-data` and propagated to the LSA nodes by means of service code.

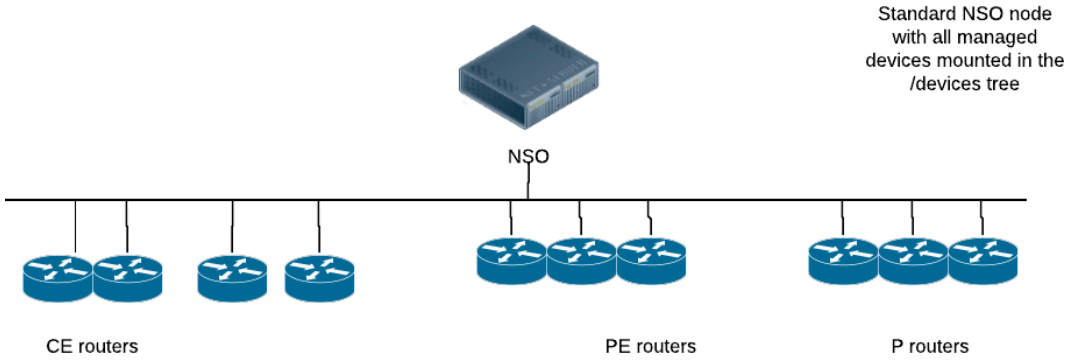
The example, `examples.ncs/service-provider/mpls-vpn-layered-service-architecture` does exactly this, the upper layer data model in `upper-nso/packages/l3vpn/src/yang/l3vpn.yang` now looks as:

```
list l3vpn {
   description "Layer3 VPN";

   key name;
   leaf name {
     type string;
   }

   leaf route-distinguisher {
     description "Route distinguisher/target identifier unique for the VPN";
     mandatory true;
     type uint32;
   }

   list endpoint {
     key "id";
     leaf id {
       type string;
     }
     leaf ce-device {
       mandatory true;
       type string;
     }
     .......
```

The `ce-device` leaf is now just a regular string, not a leafref.

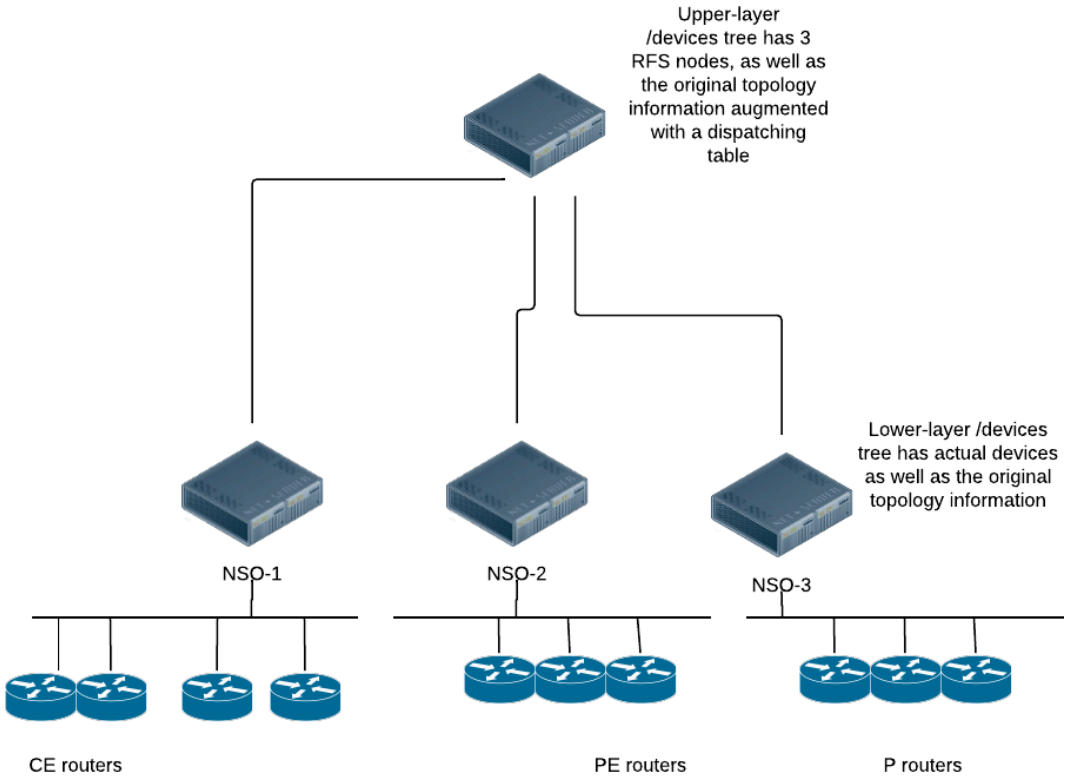
So, instead of a NSO topology that looks like

Figure 5. NSO topology



we want an NSO architecture that looks like:

Figure 6. NSO LSA topology



The task for the upper layer FastMap code is then to instantiate a copy of itself on the right lower layer NSO nodes. The upper layer FastMap code must:

- Determine which routers, (CE, PE or P) will be touched by it's execution.
- Look in it's dispatch table - which lower layer NSO nodes are used to host these routers
- Instantiate a copy of itself on those lower layer NSO nodes. One extremely efficient way to do that, is to use the `Maapi.copy_tree()` method. The code in the example contains code that looks like:

```
            public Properties create(
                ....
                NavuContainer lowerLayerNSO = ....

                Maapi maapi = service.context().getMaapi();
                int tHandle = service.context().getMaapiHandle();
                NavuNode dstVpn = lowerLayerNSO.container("config").
                        container("l3vpn", "vpn").
                        list("l3vpn").
                        sharedCreate(serviceName);
                ConfPath dst = dstVpn.getConfPath();
                ConfPath src = service.getConfPath();

                maapi.copy_tree(tHandle, true, src, dst);
```

Finally, we must make a minor modification to the lower layer (RFS) provisioning code too. Originally, the FastMap code wrote all config for all routers participating in the VPN, now with the LSA partitioning, each lower layer NSO node is only responsible for the portion of the VPN which involves devices that reside in its /devices tree, thus the provisioning code must be changed to ignore devices that do not reside in the /devices tree.

As an example of the different YANG files and namespaces, we'll walk through the actual process of splitting up a monolithic service, let's assume that our original service resides in a file, `myserv.yang` and looks like:

```
module myserv {

  namespace "http://example.com/myserv";
  prefix ms;

  .....

  list srv {
    key name;
    leaf name {
      type string;
    }

    uses ncs:service-data;
    ncs:servicepoint vlanspnt;

    leaf router {
       type leafref {
         path "/ncs:devices/ncs:device/ncs:name";
    .....
    }
}
```

In an LSA setting, we want to keep this module as close to the original as possible. We clearly want to keep the namespace, the prefix and the structure of the YANG identical to the original. This is to not disturb any provisioning systems north of the original NSO. Thus with only minor modifications, we want to run this module at the CFS node, but with non applicable leafrefs removed, thus at the CFS node we would get:

```
module myserv {

  namespace "http://example.com/myserv";
  prefix ms;

  .....
```

```
  list srv {
    key name;
    leaf name {
      type string;
    }

    uses ncs:service-data;
    ncs:servicepoint vlanspnt;

    leaf router {
       type string;
    .....
    }
}
```

Now, we want to run almost the same YANG module at the RFS node, however the namespace must be changed. For the sake of the CFS node, we're going to NED compile the RFS and NSO doesn't like the same namespace to occur twice, thus for the RFS node, we would get a YANG module `myserv-rfs.yang` that looks like:

```
module myserv-rfs {

  namespace "http://example.com/myserv-rfs";
  prefix ms-rfs;

  .....

  list srv {
    key name;
    leaf name {
      type string;
    }

    uses ncs:service-data;
    ncs:servicepoint vlanspnt;

    leaf router {
       type leafref {
         path "/ncs:devices/ncs:device/ncs:name";
    .....
    }
}
```

This file can - and should - keep the leafref as is.

The final and last file we get is the compiled NED, that should be loaded in the CFS node. The NED is directly compiled from the RFS model, as an LSA NED.

```
$ ncs-make-package --lsa-netconf-ned /path/to-rfs-yang  myserv-rfs-ned
```

Thus we end up with 3 distinct packages from the original one.

1. The original, slated for the CFS node, with leafrefs removed.
2. The modified original, slated for the RFS node, with the namespace and the prefix changed.
3. The NED, compiled from the RFS node code, slated for the CFS node.

CHAPTER 3

Setting up LSA deployments

LSA setup

The purpose of the upper CFS node is to manage all CFS services and to push the resulting service mappings to the RFS services. The lower RFS nodes are configured as devices in the device tree of the upper CFS node and the RFS services are located under the `/devices/device/config` accordingly.

This is almost identical to the relation between a normal NSO node and the normal devices. However there are differences when it comes to commit-parameters and to the commit-queue and there are also some other specific features that are special to LSA.

In a deployment of an LSA cluster all the nodes can have the same version of NSO running this called *Single Version Deployment* in contrast to deployments where different versions of NSO are running *Multi Version Deployment*.

The choice between *Single Version Deployment* and *Multi Version Deployment* is dependent on the functional needs. The former is far easier to maintain and is a good starting point. It is always possible to migrate from a *Single Version Deployment* to a *Multi Version Deployment*, the opposite is a lot harder.

Only the upper CFS node is affected of the choice between *Single Version Deployment* and *Multi Version Deployment*.

RFS node prerequisites

Changes to ncs.conf

Here we expect the lower RFS nodes to be already configured and setup as stand-alone nodes. This implies also that the distribution of devices that each lower RFS node manages are already decided and setup.

Also the actual RFS services need to be implemented and loaded as packages.

All in all the RFS nodes are ordinary NSO nodes managing a certain number of a devices and containing a number of RFS. The only other requirement is that these nodes enable NETCONF communication north-

bound since this is how the upper CFS node will interact with the lower RFS nodes. To enable NETCONF north-bound the following config snippet is necessary in ncs.conf on the lower RFS nodes:

```
  <netconf-north-bound>
    <enabled>true</enabled>
    <transport>
      <ssh>
        <enabled>true</enabled>
        <ip>0.0.0.0</ip>
        <port>2022</port>
      </ssh>
    </transport>
  </netconf-north-bound>
```

Single Version Deployment

The lower RFS nodes as devices

In the upper CFS node the lower RFS nodes are configured as devices in the `/devices/device` tree. These devices are communicating via NETCONF and the ned-id must be set to *tailf-ncs-ned:lsa-netconf* or ned-id derived from that. This ned-id is set under `/devices/device/device-type/netconf/ned-id` for each lower RFS node.

The example `examples.ncs/getting-started/developing-with-ncs/22-lsa-single-version-deployment` illustrates the different steps to set up an LSA cluster.

In the example the upper CFS node is called *upper-nso* and there are two lower RFS nodes called *lower-nso-1* and *lower-nso-2* respectively. The device configuration in the upper CFS node is the following:

```
admin@upper-nso% show devices device | display-level 4
device lower-nso-1 {
    lsa-remote-node lower-nso-1;
    authgroup       default;
    device-type {
        netconf {
            ned-id lsa-netconf;
        }
    }
    state {
        admin-state unlocked;
    }
}
device lower-nso-2 {
    lsa-remote-node lower-nso-2;
    authgroup       default;
    device-type {
        netconf {
            ned-id lsa-netconf;
        }
    }
    state {
        admin-state unlocked;
    }
}
```

Note The RFS nodes has a lot more configuration than what is visible through the RFS NED package, out-of-sync will thus be frequent and must therefore be expected. Hence it is a strong recommendation to accept the lower RFS nodes to be out of sync. The following command will handle this.

```
admin@upper-nso% set devices device lower-nso-* out-of-sync-commit-behaviour accept
```

Device compiled RFS services

On the upper CFS node we need to load device compiled RFS YANG models that comes from the lower RFS nodes. This resembles the way we compile device YANG models in normal NED packages.

For this purpose the *ncs-make-package* tool has a specific argument *--lsa-netconf-ned* which creates a basic package that compiles the RFS YANG models with the ned-id *tailf-ncs-ned:lsa-netconf* as default.

The Makefile of the example uses the tool *ncs-make-package* to create a NED package in the upper-nso based on the RFS YANG models from the lower RFS node *lower-nso-1*.

```
upper-nso/packages/rfs-vlan-ned:
        ncs-make-package --no-netsim --no-java --no-python           \
            --lsa-netconf-ned package-store/rfs-vlan/src/yang        \
            --dest $@ --build $(@F)
```

The *ncs-make-package* tool will in this example create a NED package for the rfs-vlan service.

`--no-netsim` no netsim is needed in the example as an actual NSO node will be running.

`--no-java`, `--no-python` as the services running on the upper CFS node are pure template services so no java or python code needs to be generated.

`--lsa-netconf-ned package-store/rfs-vlan/src/yang` the package to be generated is an LSA NETCONF NED based on the YANG files of the service on the lower RFS node

`--dest $@` the name of the destination directory. The special make variable `$@` denotes the target in the Makefile, in this case it will be `upper-nso/packages/rfs-vlan-ned`

`--build` build the package after it has been created. This is the same as executing the command: `make -C upper-nso/packages/rfs-vlan-ned/src`

`$(@F)` is the name of the package, this is a special make variable meaning the filename of the target, in this case it is `rfs-vlan-ned`

In the directory `upper-nso/packages/rfs-vlan-ned/src` you can find the generated `Makefile` of the package:

```
all: fxs
.PHONY: all

# Include standard NCS examples build definitions and rules
include $(NCS_DIR)/src/ncs/build/include.ncs.mk

SRC = $(wildcard yang/*.yang)
FXS = $(SRC:yang/%.yang=ncsc-out/modules/fxs/%.fxs)
DIRS =  ncsc-out ../load-dir

NCSVER     = $(shell $(NCS) --version | sed 's/\([0-9]*\.[0-9]*\).*/\1/')
CISCO_NSO  = $(NCS_DIR)/packages/lsa/cisco-nso-nc-$(NCSVER)
NED_ID_ARG = --ncs-ned-id tailf-ncs-ned:lsa-netconf
```

```
YANGPATH   =$(CISCO_NSO)/src/yang

NCSCPATH   = $(YANGPATH:%=--yangpath %)

fxs: $(DIRS) ../package-meta-data.xml ncsc-out/.done
.PHONY: fxs

$(DIRS):
 mkdir -p $@

../package-meta-data.xml: package-meta-data.xml.in
 rm -rf $@
 cp $< $@;                                          \
 chmod -w $@

ncsc-out/.done: $(SRC)
 $(NCSC) --ncs-compile-bundle yang                  \
     --ncs-device-dir ncsc-out                      \
     --fail-on-warnings                             \
     --ncs-device-type netconf                      \
     $(NCSCPATH)                                    \
     $(NED_ID_ARG)                                  \
     $(NCSC_EXTRA_FLAGS)
 cp ncsc-out/modules/fxs/*.fxs ../load-dir
 for f in `echo ../load-dir/*.fxs`; do \
    true; \
 done
 touch ncsc-out/.done

clean:
 rm -rf $(DIRS)
 rm -rf ../package-meta-data.xml
.PHONY: clean
```

In this makefile we find a variable

```
NED_ID_ARG = --ncs-ned-id tailf-ncs-ned:lsa-netconf
```

which tells NSO this is an LSA NETCONF NED and will be handled as such. Also note the YANG models are compiled using the `--ncs-compile-bundle` option this ensures the YANG compiler to compile all the YANG models in the correct order based on dependencies among the models.

Note When a service YANG model from the lower RFS node is compiled as a NED for the upper CFS node there are caveats correlated to this. The original RFS service YANG model can have dependencies to other YANG models in the lower RFS node that are not present in the upper CFS node.

The solution to this problem is to remove the dependencies in the YANG model before compilation. Normally this can be solved by changing datatype in the NED compiled copy of the YANG model, e.g. from `leafref` or `instance-identifier` to string. There will then be an implicit conversion between types, at runtime, in the communication between the upper CFS node and the lower RFS node.

It is only the NED compiled copy of the YANG model in the upper CFS node that needs to change. The lower RFS node YANG model could remain the same.

If there are dependencies on the ncs namespace in the RFS YANG models which cannot be changed a different approach is needed. To handle this configure and set up the LSA cluster as in *Multi Version Deployment*.

Multi Version Deployment

The lower RFS nodes as devices

In the upper CFS node the lower RFS nodes are configured as devices in the `/devices/device` tree. These devices are communicating via NETCONF and the ned-id for each such device must be a ned-id from a *cisco-nso-nc-X.Y* package where X.Y is the two first numbers in the version number of the NSO of the lower RFS node. This ned-id is set under `/devices/device/device-type/netconf/ned-id` for each lower RFS node.

In the example: `examples.ncs/getting-started/developing-with-ncs/28-lsa-multi-version-deployment` the upper CFS node is called *upper-nso* and there are two lower RFS nodes called *lower-nso-1* and *lower-nso-2* respectively. The device configuration in the upper CFS node is the following:

```
admin@upper-nso% show devices device | display-level 4
device lower-nso-1 {
    lsa-remote-node lower-nso-1;
    authgroup       default;
    device-type {
        netconf {
            ned-id cisco-nso-nc-5.4;
        }
    }
    state {
        admin-state unlocked;
    }
}
device lower-nso-2 {
    lsa-remote-node lower-nso-2;
    authgroup       default;
    device-type {
        netconf {
            ned-id cisco-nso-nc-5.4;
        }
    }
    state {
        admin-state unlocked;
    }
}
```

Note The lower RFS nodes has a lot more configuration than what is visible through the package cisco-nso-nc-X.Y and through the RFS NED, out-of-sync will thus be frequent and must therefore be expected. Hence it is a strong recommendation to accept the RFS nodes to be out of sync. The following command will handle this.

```
admin@upper-nso% set devices device lower-nso-* out-of-sync-commit-behaviour accept
```

The upper CFS node has to have the same or a newer version than the version running on the lower RFS nodes. In the example all nodes have the same version running but the principle is the same to use *cisco-nso-nc-X.Y* ned-id to run different versions as well.

The supported versions of the lower RFS node can be seen in the directory `$NCS_DIR/packages/lsa`.

If the NSO version in CFS is 5.5 or greater, and the NSO version in the RFS is 5.4.x or lower, then the config `/ncs-config/logs/trace-id` should be set to false in ncs.conf on the CFS node. Setting it

to true (or leaving it unspecified, since true is the default value) will lead to commits towards those RFS nodes failing.

Device compiled RFS services

On the upper CFS node you need to load device compiled RFS YANG models that comes from the lower RFS nodes. This resembles how device YANG models in normal NED packages are compiled.

For this purpose the *ncs-make-package* tool has a specific argument *--lsa-netconf-ned* which creates a package that compiles the RFS YANG models with the specified ned-id *cisco-nso-nc-X.Y*.

In the example: `examples.ncs/getting-started/developing-with-ncs/28-lsa-multi-version-deployment` the Makefile uses the tool *ncs-make-package* to create a NED package in the upper-nso for the lower-nso-1 RFS YANG model;

```
upper-nso/packages/rfs-vlan-nc-5.4:
 ncs-make-package --no-netsim --no-java --no-python           \
     --lsa-netconf-ned package-store/rfs-vlan/src/yang        \
     --lsa-lower-nso cisco-nso-nc-5.4                         \
     --package-version $* --dest $@ --build $(@F)
```

The *ncs-make-package* tool will in this example create a NED in `upper-nso/packages` for the rfs-vlan service.

`--no-netsim` no netsim is needed in the example as an actual NSO node will be run

`--no-java`, `--no-python` as the services running on the upper CFS node are pure template services so no java or python code needs to be generated.

`--lsa-netconf-ned package-store/rfs-vlan/src/yang` the package to be generated is an LSA NETCONF NED based on the YANG files of the service on the lower RFS node

`--dest $@` the name of the destination directory. The special make variable `$@` denotes the target, in this case it will be `upper-nso/packages/rfs-vlan-nc-5.4`. Note as this setup should cater for multiple versions of lower RFS nodes a naming convention needs to be used to distinguish packages compiled for those different versions of NSO.

`--build` build the package after it has been created. This is the same as executing the command: `make -C upper-nso/packages/rfs-vlan-nc-5.4/src`

`$(@F)` is the name of the package, this is a special make variable meaning the filename of the target in this case it is `rfs-vlan-nc-5.4`

In the directory `upper-nso/packages/rfs-vlan-nc-5.4/src` you can find the generated Makefile of the package:

```
all: fxs
.PHONY: all

# Include standard NCS examples build definitions and rules
include $(NCS_DIR)/src/ncs/build/include.ncs.mk

SRC = $(wildcard yang/*.yang)
FXS = $(SRC:yang/%.yang=ncsc-out/modules/fxs/%.fxs)
DIRS =  ncsc-out ../load-dir

MNAME      = cisco-nso-nc-5.4
CISCO_NSO  = $(NCS_DIR)/packages/lsa/$(MNAME)
NED_ID_ARG = --ncs-ned-id cisco-nso-nc-5.4:cisco-nso-nc-5.4

YANGPATH   =$(CISCO_NSO)/src
```

```
NCSCPATH   = $(YANGPATH:%=--yangpath %)

fxs: $(DIRS) ../package-meta-data.xml ncsc-out/.done
.PHONY: fxs

$(DIRS):
 mkdir -p $@

../package-meta-data.xml: package-meta-data.xml.in
 rm -rf $@
 cp $< $@;                                              \
 chmod -w $@

ncsc-out/.done: $(SRC)
 $(NCSC) --ncs-compile-bundle yang                      \
     --ncs-device-dir ncsc-out                          \
     --fail-on-warnings                                 \
     --ncs-depend-package $(CISCO_NSO)                  \
     --ncs-device-type netconf                          \
     $(NCSCPATH)                                        \
     $(NED_ID_ARG)                                      \
     $(NCSC_EXTRA_FLAGS)
 cp ncsc-out/modules/fxs/*.fxs ../load-dir
 for f in `echo ../load-dir/*.fxs`; do \
    true; \
 done
 touch ncsc-out/.done

clean:
 rm -rf $(DIRS)
 rm -rf ../package-meta-data.xml
.PHONY: clean
```

In this makefile we find a variable

```
NED_ID_ARG = --ncs-ned-id cisco-nso-nc-5.4:cisco-nso-nc-5.4
```

which tells NSO this is an LSA NETCONF NED and will be handled as such. Also note the YANG models are compiled using the `--ncs-compile-bundle` option this makes the YANG compiler to compile all the YANG models in the correct order based on dependencies among the models.

Note When a service YANG model from the lower RFS node is compiled as a NED for the upper CFS node there are caveats correlated to this. The original RFS service YANG model can have dependencies to other YANG models in the lower RFS node that are not present in the upper CFS node.

The solution to this problem is to remove the dependencies in the YANG model before compilation. Normally this can be solved by changing datatype in the NED compiled copy of the YANG model, e.g. from `leafref` or `instance-identifier` to string. There will then be an implicit conversion between types, at runtime, in the communication between the upper CFS node and the lower RFS node.

It is only the NED compiled copy of the YANG model in the upper CFS node that needs to change. The lower RFS node YANG model could remain the same.

The standard cisco-nso NED LSA package

The NSO release includes packages specifically tailored for LSA to be used by the upper CFS node if the lower RFS nodes are running a version different than the one on the upper CFS node. The packages are

named `cisco-nso-nc-X.Y` where X.Y are the two most significant numbers of the NSO release (the package names changed from `tailf-nso-nc-X.Y` with NSO 5.4).

The supported packages can be found in the `$NCS_DIR/packages/lsa` directory. Each package contains the complete model of the `ncs` namespace for the corresponding NSO version, compiled as an LSA NED. The ned-id matches the package name.

The cisco-nso package needs to be installed in the packages directory of the upper CFS node if the RFS service contains references into the ncs name space or if the ncs namespace needs to be accessed from the upper CFS node.

If there are no such requirements the `cisco-nso-nc-X.Y` does not need to be installed. However, it is always needed at compile time so the right groupings and data types are used.

Please always use the `cisco-nso-nc-X.Y` package included with the NSO version of the upper CFS node and not some older variant (such as the one from lower RFS node) as it may not work correctly.

Example 28-lsa-multi-version-deployment

In the README file of the example: `examples.ncs/getting-started/developing-with-ncs/28-lsa-multi-version-deployment` the different steps how to set up an LSA cluster are described.

First build the example for manual set up.

```
$ make clean manual
$ make start-manual
$ make cli-upper-nso
```

Then configure the nodes in the cluster. This is needed so the upper CFS node can receive notifications from the lower RFS node and prepare the upper CFS node to be used with the commit-queue.

```
> configure

% set cluster device-notifications enabled
% set cluster remote-node lower-nso-1 authgroup default username admin
% set cluster remote-node lower-nso-1 address 127.0.0.1 port 2023
% set cluster remote-node lower-nso-2 authgroup default username admin
% set cluster remote-node lower-nso-2 address 127.0.0.1 port 2024
% set cluster commit-queue enabled
% commit
% request cluster remote-node lower-nso-* ssh fetch-host-keys
```

To be able to handle the lower nso node as an LSA node the correct version of the cisco-nso-nc package needs to be installed. In this example 5.4 is used.

Create a link to the cisco-nso package in the packages directory of the upper CFS node:

```
$ ln -sf ${NCS_DIR}/packages/lsa/cisco-nso-nc-5.4 upper-nso/packages
```

Reload the packages:

```
% exit
> request packages reload

e>>> System upgrade is starting.
>>> Sessions in configure mode must exit to operational mode.
>>> No configuration changes can be performed until upgrade has completed.
>>> System upgrade has completed successfully.
reload-result {
```

```
    package cisco-nso-nc-5.4
    result true
}
```

Now when the cisco-nso-nc package is in place configure the two lower nso nodes and sync-from them:

```
> configure
Entering configuration mode private

% set devices device lower-nso-1 device-type netconf ned-id cisco-nso-nc-5.4
% set devices device lower-nso-1 authgroup default
% set devices device lower-nso-1 lsa-remote-node lower-nso-1
% set devices device lower-nso-1 state admin-state unlocked
% set devices device lower-nso-2 device-type netconf ned-id cisco-nso-nc-5.4
% set devices device lower-nso-2 authgroup default
% set devices device lower-nso-2 lsa-remote-node lower-nso-2
% set devices device lower-nso-2 state admin-state unlocked

% commit
Commit complete.

% request devices fetch-ssh-host-keys
fetch-result {
    device lower-nso-1
    result updated
    fingerprint {
        algorithm ssh-ed25519
        value 4a:c6:5d:91:6d:4a:69:7a:4e:0d:dc:4e:51:51:ee:e2
    }
}
fetch-result {
    device lower-nso-2
    result updated
    fingerprint {
        algorithm ssh-ed25519
        value 4a:c6:5d:91:6d:4a:69:7a:4e:0d:dc:4e:51:51:ee:e2
    }
}

% request devices sync-from
sync-result {
    device lower-nso-1
    result true
}
sync-result {
    device lower-nso-2
    result true
}
```

Now for example the configured devices of the lower nodes can be viewed:

```
% show devices device config devices device | display xpath | display-level 5

/devices/device[name='lower-nso-1']/config/ncs:devices/device[name='ex0']
/devices/device[name='lower-nso-1']/config/ncs:devices/device[name='ex1']
/devices/device[name='lower-nso-1']/config/ncs:devices/device[name='ex2']
/devices/device[name='lower-nso-2']/config/ncs:devices/device[name='ex3']
/devices/device[name='lower-nso-2']/config/ncs:devices/device[name='ex4']
/devices/device[name='lower-nso-2']/config/ncs:devices/device[name='ex5']
```

or alarms inspected:

```
% run show devices device lower-nso-1 live-status alarms summary
```

```
live-status alarms summary indeterminates 0
live-status alarms summary criticals 0
live-status alarms summary majors 0
live-status alarms summary minors 0
live-status alarms summary warnings 0
```

Now create a netconf package on the upper CFS node which can be used towards the rfs-vlan service on the lower RFS node, in the shell terminal window do;

```
$ ncs-make-package --no-netsim --no-java --no-python               \
    --lsa-netconf-ned package-store/rfs-vlan/src/yang              \
    --lsa-lower-nso cisco-nso-nc-5.4                               \
    --package-version 5.4 --dest upper-nso/packages/rfs-vlan-nc-5.4 \
    --build rfs-vlan-nc-5.4
```

No netsim, java or python is needed as the cfs-vlan running on the upper nso is a pure template service.

The created NED is an lsa-netconf NED based on the YANG files of the rfs-vlan service:

```
--lsa-netconf-ned package-store/rfs-vlan/src/yang
```

the version of the NED reflects the version of the nso on the lower node:

```
--package-version 5.4
```

The package will be generated in the packages directory of the upper nso CFS node:

```
--dest upper-nso/packages/rfs-vlan-nc-5.4
```

and the name of the package will be:

```
rfs-vlan-nc-5.4
```

Install the cfs-vlan service on the upper CFS node. In the shell terminal window do:

```
$ ln -sf ../../package-store/cfs-vlan     upper-nso/packages
```

Reload the packages once more to get the cfs-vlan package. In the CLI terminal window do:

```
% exit

> request packages reload

>>> System upgrade is starting.
>>> Sessions in configure mode must exit to operational mode.
>>> No configuration changes can be performed until upgrade has completed.
>>> System upgrade has completed successfully.
reload-result {
    package cfs-vlan
    result true
}
reload-result {
    package cisco-nso-nc-5.4
    result true
}
reload-result {
    package rfs-vlan-nc-5.4
    result true
}

> configure
Entering configuration mode private
```

Now when all packages are in place a cfs-vlan service can be configured. The cfs-vlan service will dispatch service data to the right lower RFS node depending on the device names used in the service.

In the CLI terminal window verify the service:

```
% set cfs-vlan v1 a-router ex0 z-router ex5 iface eth3 unit 3 vid 77

% commit dry-run
.....
    local-node {
        data  devices {
                  device lower-nso-1 {
                      config {
                          services {
              +               vlan v1 {
              +                   router ex0;
              +                   iface eth3;
              +                   unit 3;
              +                   vid 77;
              +                   description "Interface owned by CFS: v1";
              +               }
                          }
                      }
                  }
                  device lower-nso-2 {
                      config {
                          services {
              +               vlan v1 {
              +                   router ex5;
              +                   iface eth3;
              +                   unit 3;
              +                   vid 77;
              +                   description "Interface owned by CFS: v1";
              +               }
                          }
                      }
                  }
             }
.....
```

As ex0 resides on lower-nso-1 that part of the configuration goes there and the ex5 part goes to lower-nso-2.

Migration and Upgrades

Since an LSA deployment consists of multiple NSO nodes (or HA pairs of nodes), each can be upgraded to a newer NSO version separately. While that offers a lot of flexibility, it also makes upgrades more complex in many cases. For example, performing a major version upgrade on the upper CFS node only, will make the deployment Multi Version even if it was Single Version before the upgrade, requiring additional action on your part.

In general, staying with the Single Version Deployment is the simplest option and does not require any further LSA-specific upgrade action (except perhaps recompiling the packages). However, the main downside is that, at least for a major upgrade, you must upgrade all the nodes at the same time (otherwise, you no longer have a Single Version Deployment).

If that is not feasible, the solution is to run a Multi Version Deployment. Along with all of the requirements, the section called “Multi Version Deployment” describes a major difference from the Single Version variant: the upper CFS node uses a version-specific *cisco-nso-nc-X.Y* ned-id to refer to lower RFS

nodes. That means, if you switch to a Multi Version Deployment, or perform a major upgrade of the lower-layer RFS node, the ned-id should change accordingly. However, do not change it directly but follow the correct NED upgrade procedure described in ????. Briefly, the procedure consists of these steps:

1. Keep the currently configured ned-id for an RFS device and the corresponding packages. If upgrading the CFS node, you will need to recompile the packages for the new NSO version.
2. Compile and load the packages that are device compiled with the new ned-id, alongside the old packages.
3. Use the **migrate** action on a device to switch over to the new ned-id.

The procedure requires you to have two versions of the device compiled RFS service packages loaded in the upper CFS node when calling the **migrate** action: one version compiled by referencing the old (current) ned-id and the other one by referencing the new (target) ned-id.

To illustrate, suppose you currently have an upper-layer and a lower-layer nodes both running NSO 5.4. The nodes were set up as described in the Single Version Deployment option, with the upper CFS node using the *tailf-ncs-ned:lsa-netconf* ned-id for the lower-layer RFS node. The CFS node also uses the `rfs-vlan-ned` NED package for the rfs-vlan service.

Now you wish to upgrade the CFS node to NSO 5.7 but keep the RFS node on the existing version 5.4. Before upgrading the CFS node, you create a backup and recompile the `rfs-vlan-ned` package for NSO 5.7. Note that the package references the *lsa-netconf* ned-id, which is the ned-id configured for the RFS device in the CFS node's CDB. Then, you perform the CFS node upgrade as usual.

At this point the CFS node is running the new, 5.7 version and the RFS node is running 5.4. Since you now have a Multi Version Deployment, you should migrate to the correct ned-id as well. Therefore, you prepare the `rfs-vlan-nc-5.4` package, as described in the Multi Version Deployment option, compile the package and load it into the CFS node. Thanks to the NSO CDM feature, both packages, `rfs-vlan-nc-5.4` and `rfs-vlan-ned`, can be used at the same time.

With the packages ready, you execute the **devices device lower-nso-1 migrate new-ned-id cisco-nso-nc-5.4** command on the CFS node. The command configures the RFS device entry on CFS to use the "new" *cisco-nso-nc-5.4* ned-id, as well as migrate the device configuration and service meta-data to the new model. Having completed the upgrade, you can now remove the `rfs-vlan-ned` if you wish.

Later on you may decide to upgrade the RFS node to NSO 5.6. Again, you prepare the new `rfs-vlan-nc-5.6` package for the CFS node in a similar way as before, now using the *cisco-nso-nc-5.6* ned-id instead of *cisco-nso-nc-5.4*. Next, you perform the RFS node upgrade to 5.6 and finally migrate the RFS device on the CFS node to the *cisco-nso-nc-5.6* ned-id, with the **migrate** action.

Likewise, you can return to the Single Version Deployment, by upgrading the RFS node to the NSO 5.7, reusing the old, or preparing anew, the `rfs-vlan-ned` package and migrating to the *lsa-netconf* ned-id.

All these ned-id changes stem from the fact that the upper-layer CFS node treats the lower-layer RFS node as a managed device, requiring the correct model, just like it does for any other device type. For the same reason, minor (bug fix or patch) NSO upgrades do not result in a changed ned-id, so for those no migration is necessary.