NSO Layered Service Architecture

First Published: October 9, 2016

Last Modified: September 1, 2017

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

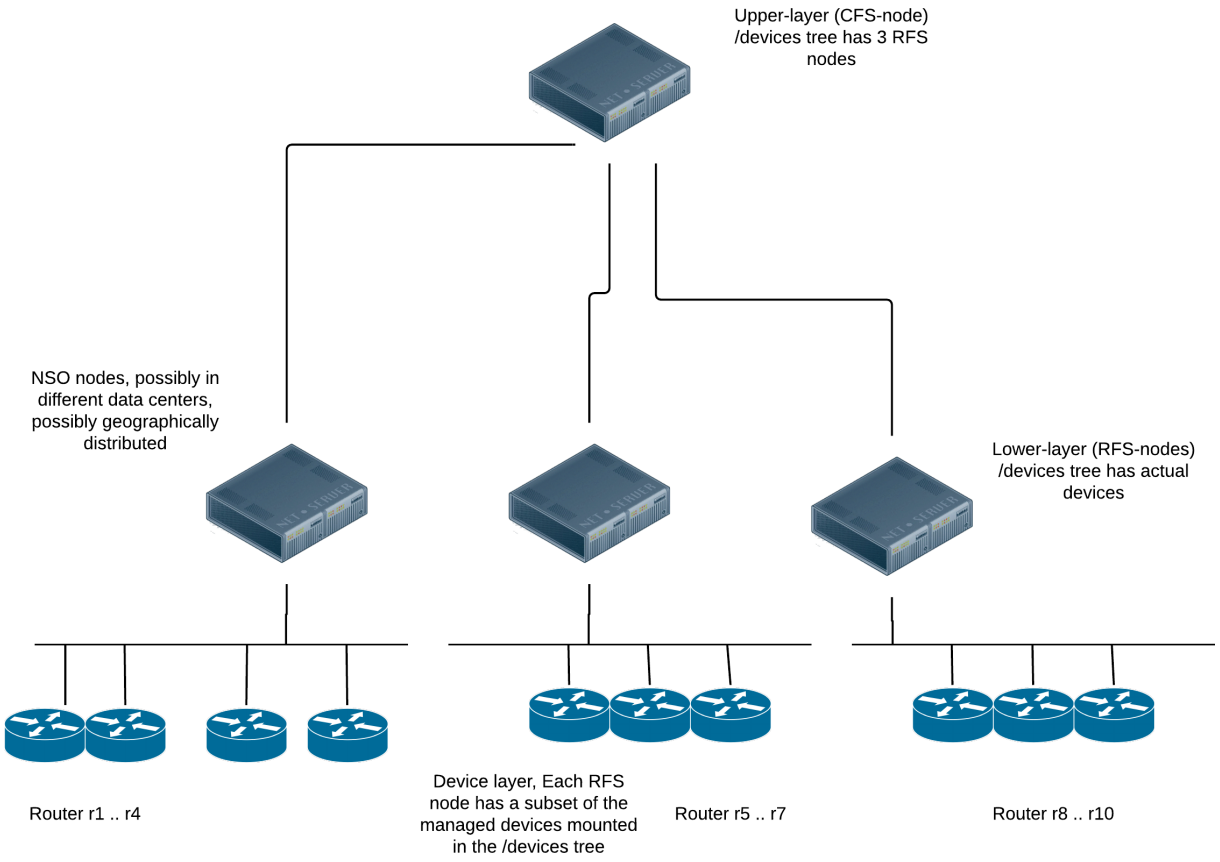
Going big - Layered service architecture

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 several lower level parts. This can be viewed as splitting the service into a customer-facing (CFS) and a resource-facing (RFS) part. The CFS code (upper-level) runs in one (or several) NSO cfs-nodes, and the RFS code (lower-level) 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 arbitrary 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 will show 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 constructs for everything the customer, or an order capture system north of the service node can enter and order. The RFS YANG constructs however can be designed differently. They are now system internal data models, and you as a designer are free to construct 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's possible to split up a monolithic application into an upper-level and lower-level service, we show how to do that in the following chapters. However, the decision to 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 that 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, 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's 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 level. 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 commit-queues. In general we recommend to enable the commit-queue in LSA applications, if commit queues 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-queues, 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 initial northbound request from the customer portal is returned.

The entire sequence is serialized though the system as a whole, if yet another northbound request arrives 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 disjunct to 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 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.

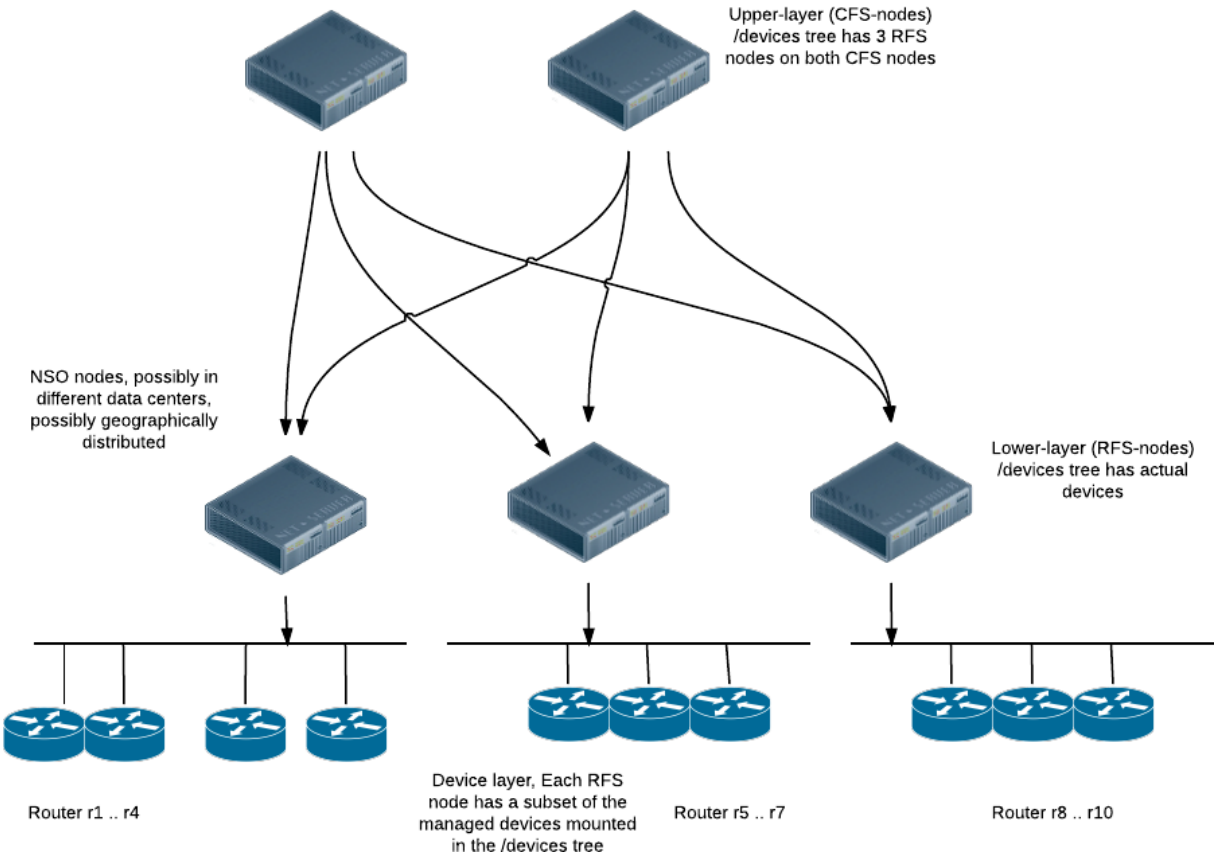
Regardless of when additional data is received from the northbound of the CFS node, data is processed and the level of concurrency in a system configured this way is very high.

If the RFS nodes FastMap code is reactive, we get similar behavior since usually Reactive FastMap applications are very fast during it's 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 commit queues 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, they would all have to run the RFS code for all services. Typically the set services (YANG code) related to L2 VPNs would be completely disjunct from the set of services (YANG code) 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, thus, 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 almost limitless, both at the upper as well as 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 level 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 e.g 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

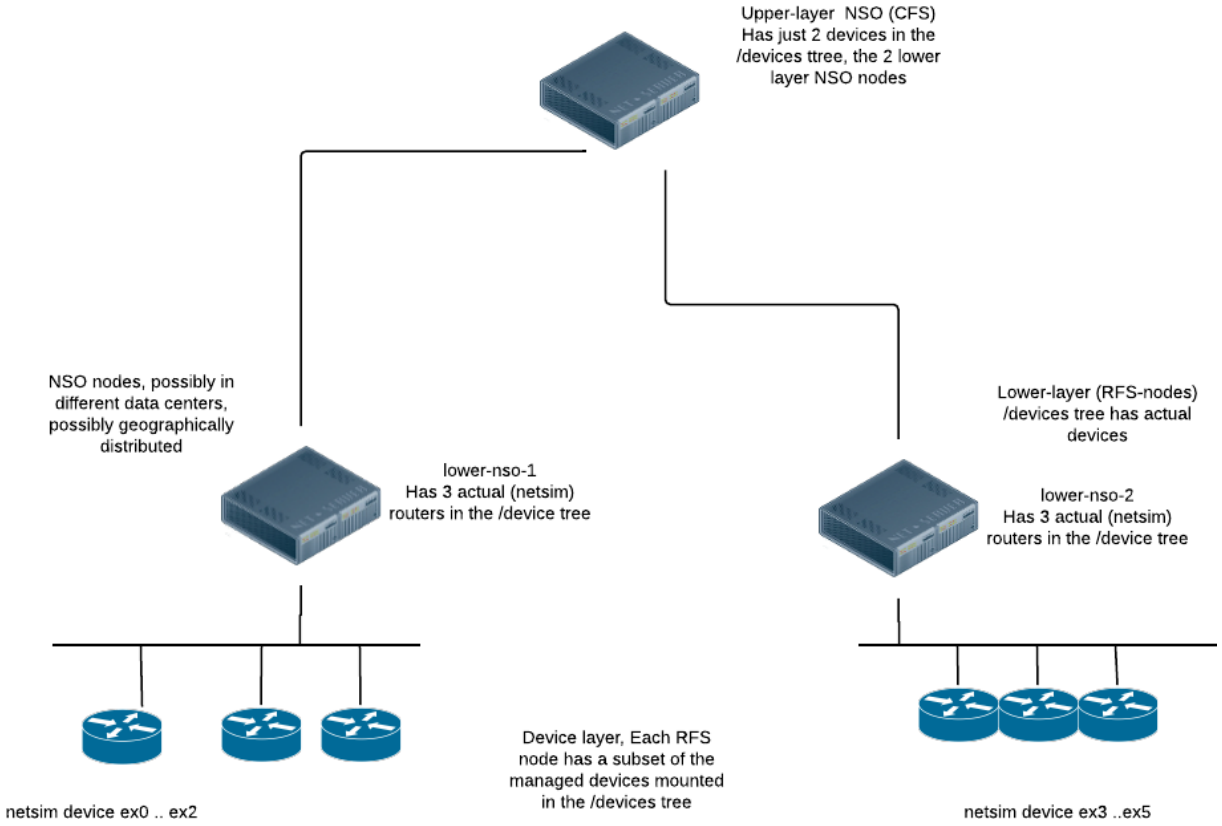
LSA examples

Greenfield LSA application

In this section we'll describe a very small Greenfield 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
    -->
    <device foreach="{*[re-match(name(), 'cfs-vlan:[az]-router')]}">

      <!--
      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>

  </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 here 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. It looks as:

```
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.

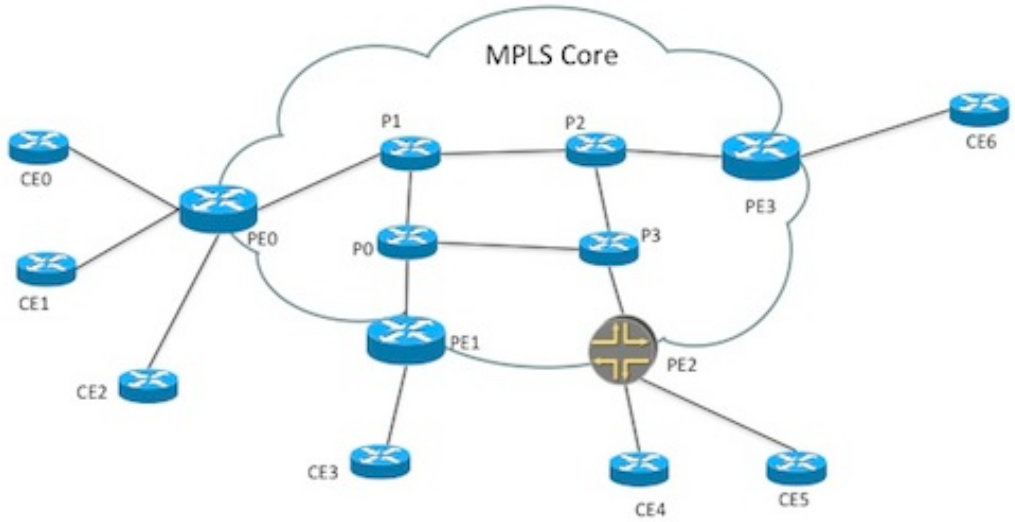
Rearchitecting an existing monolithic application into LSA

If we do not have the luxury of designing our NSO service application from scratch, but rather we 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.

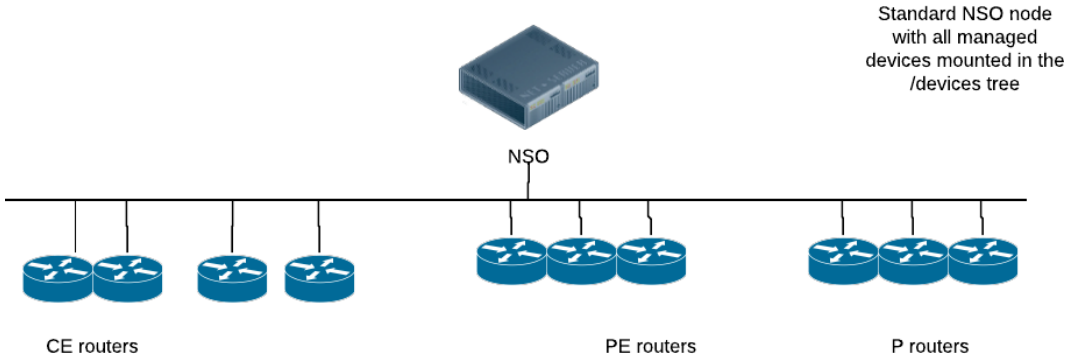
The example, `examples.ncs/service-provider/mpls-vpn-layered-service-architecture` does exactly this, the upper layer data model in `upper-nso/packages/l3vpn/src/yang/l3vpn.yang` now looks as:

```
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.

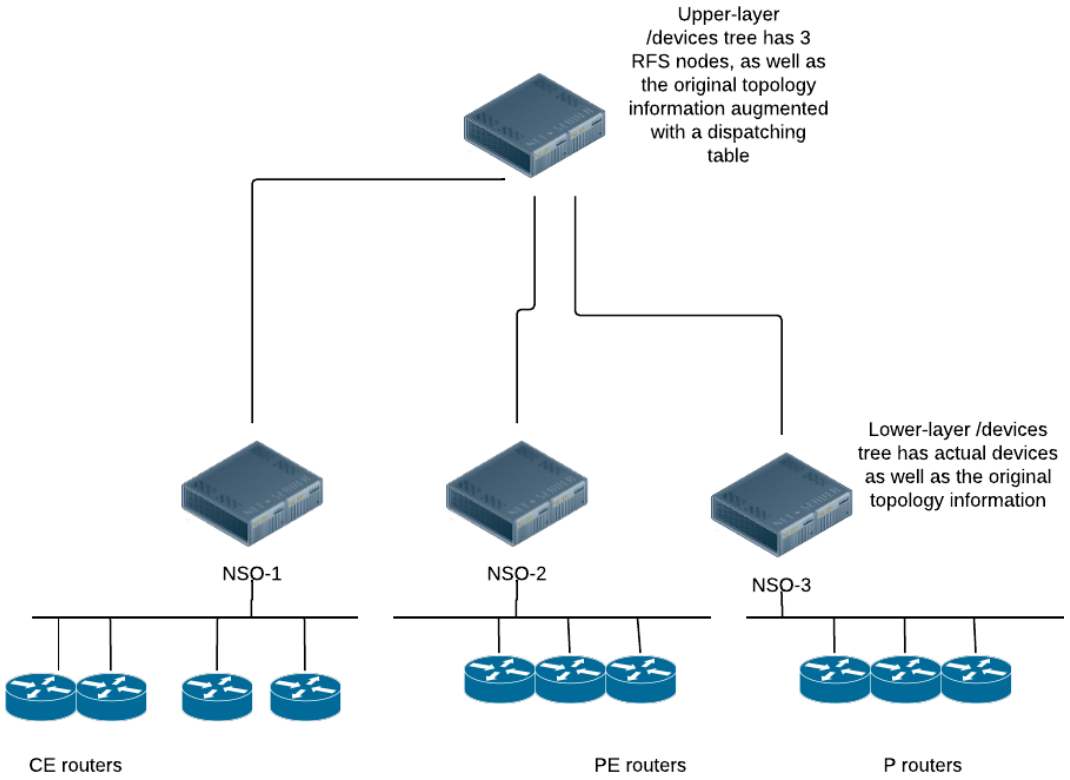
So, instead of an NSO topology that looks like

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 lowerLevelNSO = ....

                Maapi maapi = service.context().getMaapi();
                int tHandle = service.context().getMaapiHandle();
                NavuNode dstVpn = lowerLevelNSO.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.

```
$ ncs-make-package --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

LSA compared to NSO clustering

LSA compared to NSO clustering

The NSO clustering technology, as described in Chapter 7, *NSO Clustering* in *NSO 4.4.2.3 Administration Guide* is a technique to build larger scale NSO installations. However, experience with clustering over the years has taught us that clustering didn't work as well as we thought it would. In this section we will try to list some of the characteristics that the clustering technique has and compare them to LSA.

Clustering works well when we only wish to manage a large number of devices through the device manager, however it doesn't always work that well together with NSO services and FastMap, especially when some of the managed devices have very large configurations.

With clustering, the devices are remotely mounted at the top node, and they are fully visible there. If the operator executes from the CLI:

```
admin@upper-nso> show configuration devices device x config interfaces interface
```

This is internally executed in the NSO transaction manager as a series of data provider API calls to functions `get_next()` `get_elem()` and `exists()`.

For example, the NSO transaction manager would execute a call:

```
get_next(transaction, "/devices/device[name='x']/config/interfaces/interface")
```

Since the `interfaces/interface` part is remotely mounted at the NSO device node, the call will be transformed into a NETCONF call to the NSO device node to execute the `get_next(trans, "/interfaces/interface")` remotely. And so forth for all the calls required to show the configuration. Each low level call is transformed to the equivalent NETCONF call.

The advantage of this scheme is scalability, very little data is stored at the upper NSO node for each managed device and we can thus from the same NSO node see and manage a very large number of devices. The obvious disadvantage is performance, a call like the `show configuration` above can potentially translate into a very large number of NETCONF RPCs.

If you have performance problems with a clustered NSO installation, it's a good idea to first investigate the number of RPCs getting executed for the operations you perform. This can be done by turning on traffic trace for the clustered node and simply count the number of NETCONF RPCs that occur in the trace log. Obviously, network latency between the upper and the lower NSO nodes is important here too.

The solution for performance problems with clustered NSO installations is to turn on the cluster cache, paying with memory for performance. This is configured under `/cluster/global-settings/`

`caching`. The cluster cache will make the NSO transaction manager order the cluster code to cache the *entire* configuration of a device. The above `show configuration` call would then initially issue a command to get the entire configuration of device `x` and then issue all the remaining data provider API calls towards that cache. Note, the call is executed from the upper NSO node to the lower NSO node, thus we do not get the configuration from the device itself. The lower NSO node already has a complete copy of the device configuration. Usually NSO is much faster than the typical managed device to produce the full configuration.

Remember how FastMap works for NSO services. When the service is modified or re-deployed the entire output result of the service code is deleted, and then the FastMap `create()` code is once again invoked.

Typically the FastMap code modifies the device tree, this is usually the purpose of NSO services, to implement a higher level service in the network. These FastMap operations, deleting the previous output and also re-creating the same (or similar) output from the `create()` code, operates on the device tree. The device tree is remotely mounted, thus typically FastMap results in a massive amount of NETCONF RPCs, usually unacceptably many. Hence, if we combine FastMap and clustering, it's almost always required to turn on the cluster cache.

So, to conclude, we have the following scenarios and configuration combinations:

- The case with large number of devices, and no FastMap services at all. This will work just fine with clustering. The provisioning system works directly towards the `/devices` tree.

  The cache may or may not be turned on, depending on performance/memory considerations.
- The case with large number of devices, and FastMap services. If FastMap is used we must always turn on the cache, if the cache is not used, the number of RPCs will be excessive. If all managed devices are small, this will still work fine, however if some of the devices have very large configurations, populating the cache for those devices can be too time consuming.

  Thus, if your service code modifies devices that have large configuration, the LSA architecture is better. Cluster architecture will be too slow due to the time to populate the cache. Luckily it's not an impossible task to change an NSO clustered solution into an LSA solution as described above.