Designing for Performance

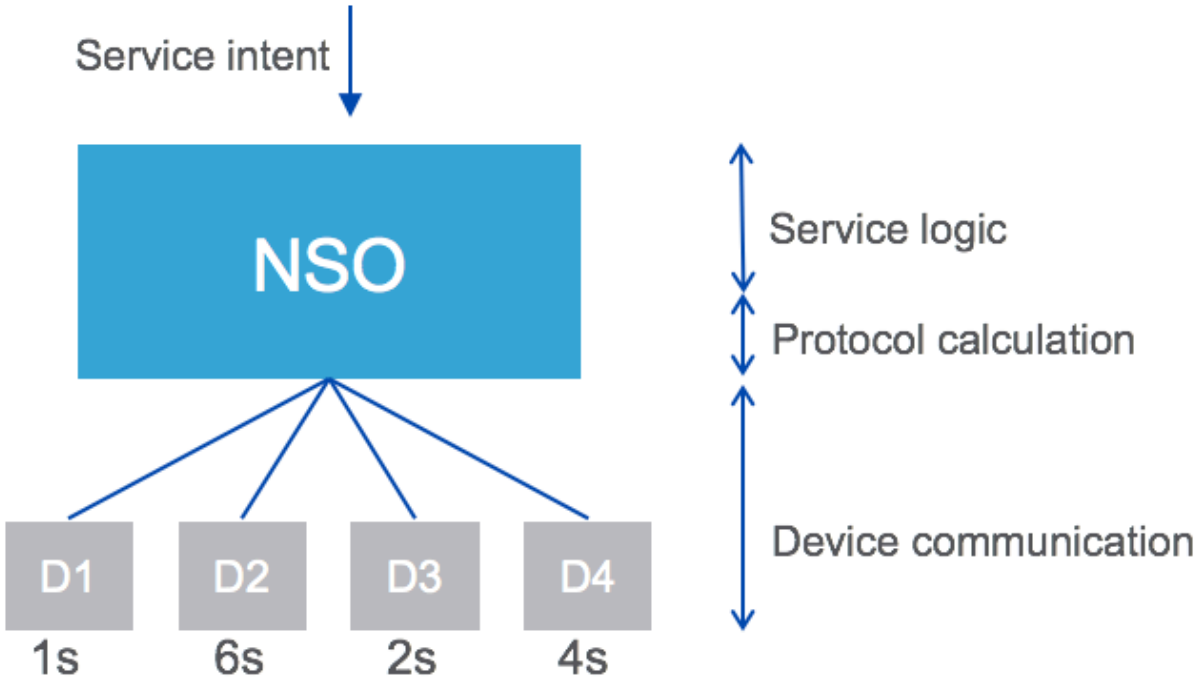
Introduction

When designing with NSO there are many different choices to make.

This document aims to give an overview of the different choices, when to choose, tradeoffs etc.

Where is Time Spent?

Before optimizing the services, you have to understand where the possible bottlenecks are. When an NSO service is executed, the time is spent in three different places.



Service Logic

Mapping of the service intent to device data models. When a lot of time is spent here, it is typically related to the service doing the wrong this. Typical examples are doing consistency checks, requests to external system or syncing from devices. One should remember that this is a critical section during which the database is locked. Only pure mapping logic should go there.

To solve performance issues, you need to carefully analyze service code, YANG models and templates.

Protocol Calculation

Where data model changes are converted into device native protocols. E.g. CLI commands or REST payload.

If you suspect something is wrong here, please contact TAC.

Device Communication

Sending the native device protocol down to the device and ensuring the device accepts and saves the changes.

In almost all the cases the device communication part are orders of magnitude more time consuming than the other parts due to the device. If this is the case, please see the information on commit queues.

In NSO standard mode, a transactions critical section spans service logic, protocol calculation and device communication. This allows NSO to roll back the entire transaction in case something fails.

Communication to the device layer is done in parallel but the transaction time will be limited by the slowest device. I.e. If the slowest device takes 6 seconds to configure the transaction will take roughly 6 seconds.

If you have two transactions: T1 touches D1 and D2 and T2 which touches D3 and D4. Given that service code and protocol calculations are negligible in comparison with apply device configuration, T1 will take about six seconds (max of one and six seconds) and T2 will take roughly four seconds (max of two and four seconds). Please note that T2 will be blocked until T1 is done, so the total time will be about ten seconds.

Please see commit queues below for an alternate approach.

Testing During Design

When designing services, a key factor to success is to test on devices (can be netsims) with production like configuration. There have been many cases where services have worked well in the lab but failed miserably in the production environment.

The main reason for this has been that the lab devices' configuration has been too small in comparison with the production environment. If testing is done on devices with a few thousand lines of configuration and the production devices has hundreds of thousands of lines of configuration, there is a huge risk of running into performance issues.

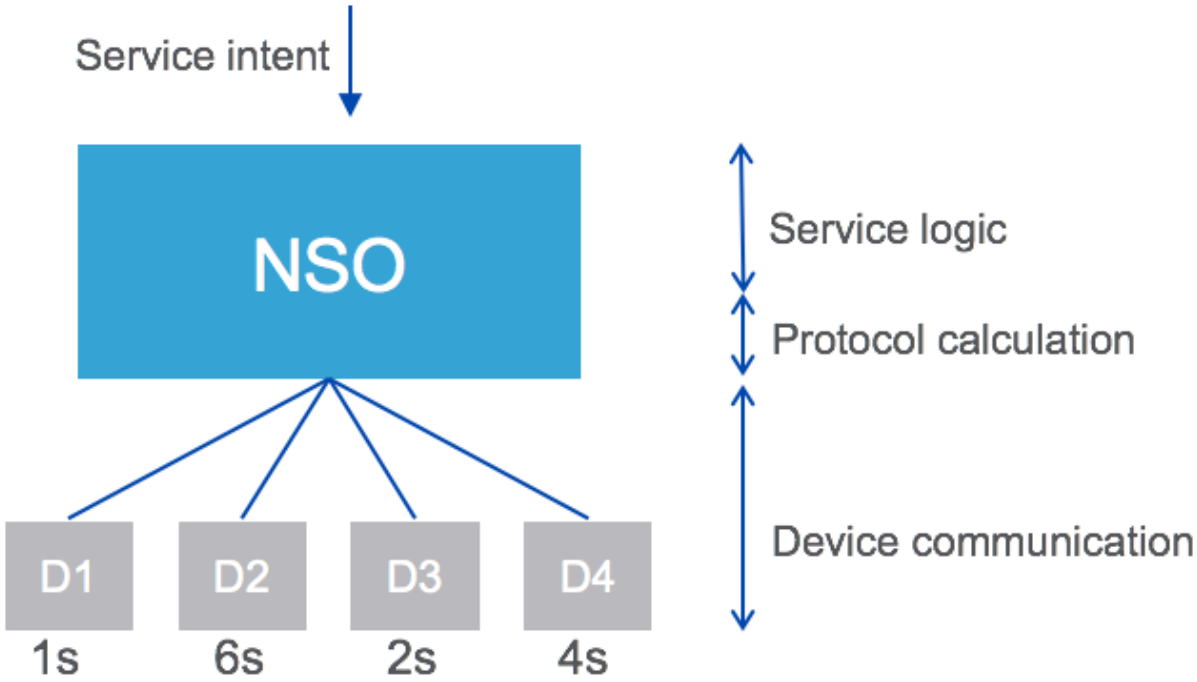
The network itself may also be a limiting factor in the production environment, i.e. if management network is slow…. REWRITE!

Hardware

We have seen cases where running NSO in a VM can hurt performance. Adding more RAM to existing hardware may be a cheaper solution to scale than adding more NSO instances.

Commit Queues

In the NSO commit queue mode the device communication takes place outside the transaction's critical section. Starting with NSO 4.4, the protocol calculations are also done outside the critical section.



This dramatically increases throughput in the system but you lose parts of the atomicity in the transaction. The failure case when one of the devices does not accept the configuration changes is handled differently.

Given the example above with T1 and T2. Since T1 and T2 don't touch the same devices, they can now execute in parallel (again assuming that the time spent in service logic and protocol calculation is negligible). The total time will therefore be six seconds.

Please not that if two transactions touch the same device, the transactions will again be run in serial.

Enabling Commit Queues

Enabling commit queues is very simple and can be done on a device bases or as a global setting. No change has to be made to the service itself to utilize commit queues.

Error Handling

In commit queue mode, the service intent is still handled as a transaction however it is queued up per device. All validation etc. is done exactly the same.

The failure case where a device does not accept the changes is handled differently. Let's assume you have a service that writes to devices D1, D2 and D3. If device D2 does not accept the changes written to device D1 and D3 are **not** reverted. Also, NSO will not try to revert the changes written to device D2 and NSO will continue to write other items in the queue for device D2.

An alarm will be raised by NSO and the API (if run in sync mode) will receive an error with information about what failed in the transaction.

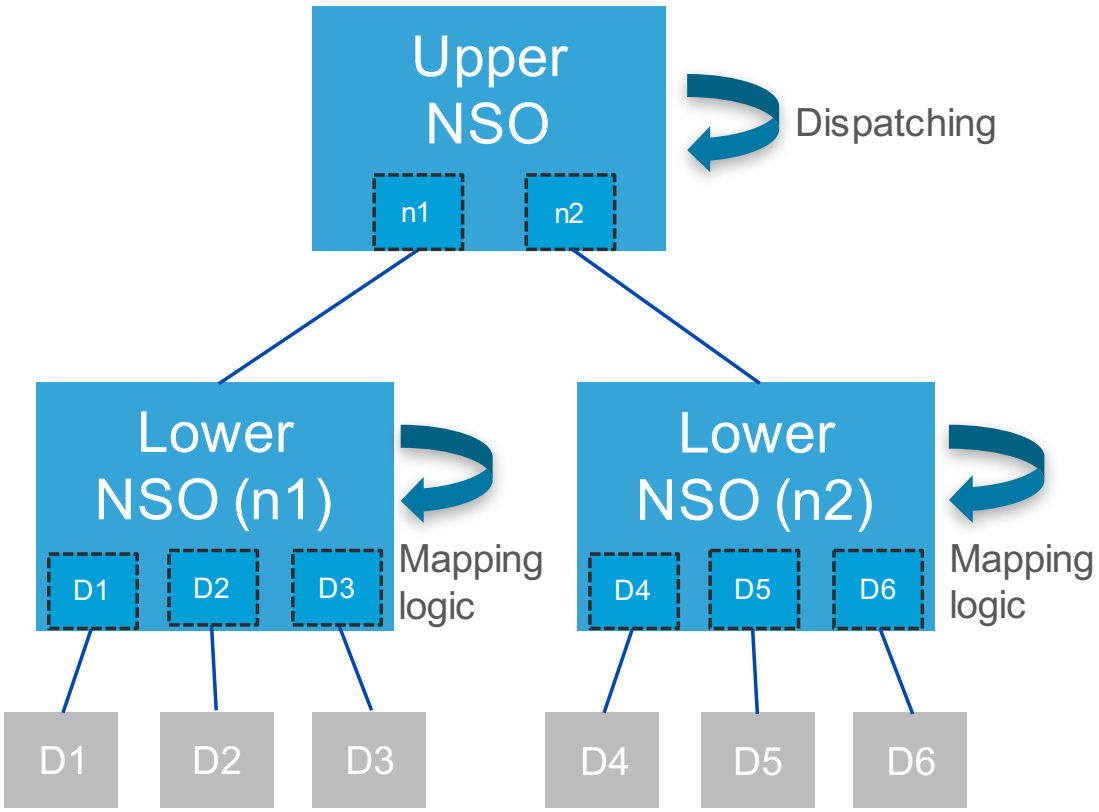
This adds complexity to the services and possibly to the north bound system.

NED Support

Please make sure the NEDs support commit queues, starting with NSO 4.4 any NED can be used with commit queues.

Layered Service Architecture (LSA)

LSA is a way to scale the device list, distribute computing and achieve scalability without the performance and throughput penalties of clustering.



An LSA deployment may have several upper NSO nodes and several lower NSO nodes. The Upper NSO node sees the lower NSO nodes and their service instances, rather than the network devices.

The Upper NSO node can either function as a dispatcher to the Lower NSO nodes or provide a further level of abstraction of the services, e.g. CFS->RFS.

Each Lower NSO is autonomous and connects to a distinct group of devices.

Typically, all service mapping logic and REACTIVE FASTMAP loops are done down on the lower nodes, distributing computation throughput the deployment.

It is possible to have service logic in the upper layer, this could for example be to cherry-pick what lower NSO nodes are involved in the service.

LSA Standard Mode

In standard mode, i.e. no commit queues are enabled, you have full transactional guarantees. Let's say you have a service spanning e.g. devices D1, D2, D4 and D5. The lock is taken on the upper NSO and both lower NSOs for the duration of the transaction, including writing to the devices.

In case a device fails, everything is rolled back.

LSA Commit Queues to Lower NSO

It you enable commit queues between the upper NSO and the lower NSOs you get transaction guarantees within the lower NSOs. So if device D5 fails, the transaction is rolled back in n2 but **not** in n1.

Please note that you'll have to write code to send notifications of success and failure from the lower nodes to the upper node(s) yourself.

LSA Commit Queues to Devices

The next step is to enable commit queues between the lower NSO and the devices. This way you can achieve "full parallelism". Please note that nothing is rolled back in case of a failed device.

Please note that you'll have to write code to send notifications of success and failure from the lower nodes to the upper node(s) yourself.

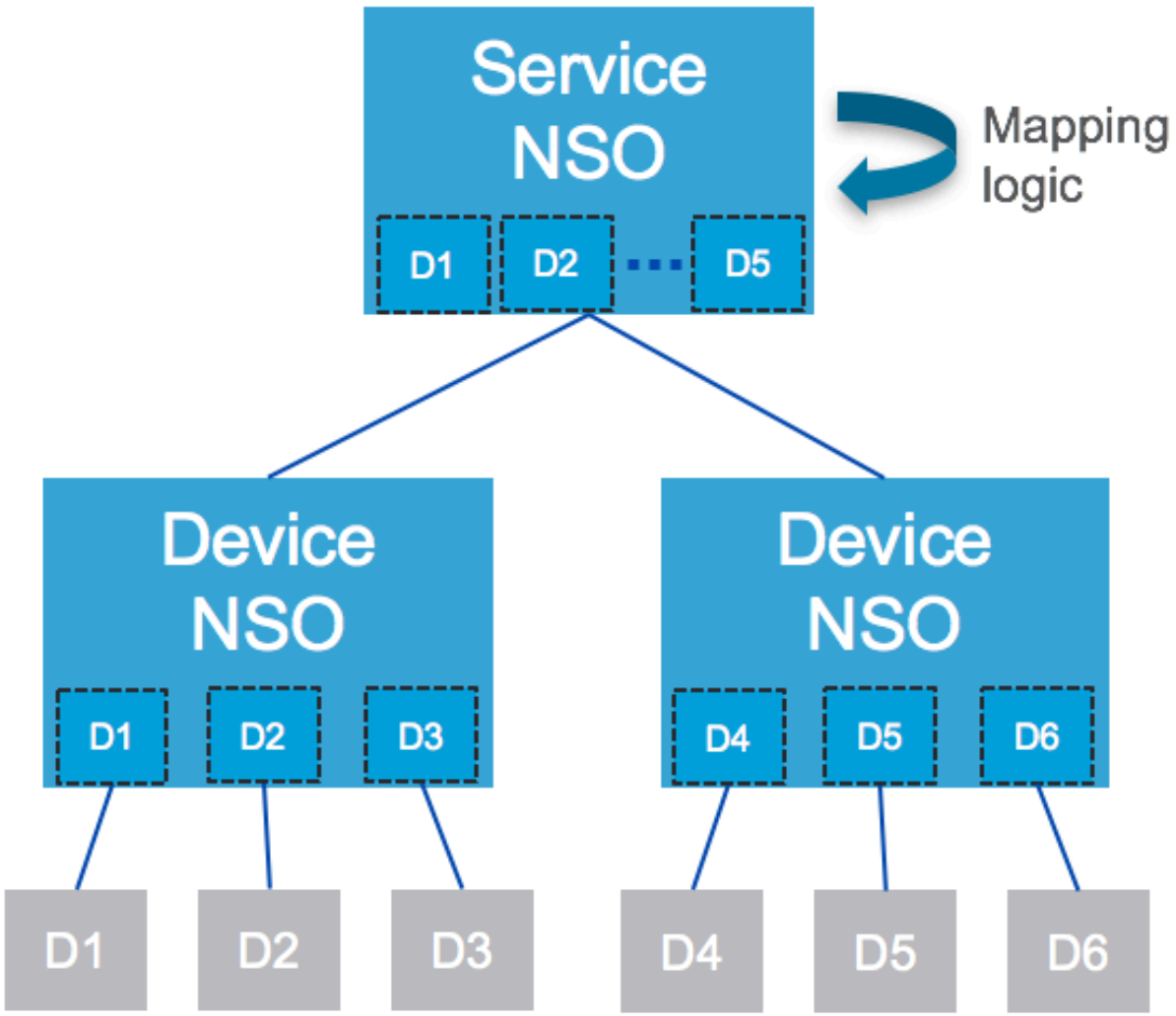
LSA Pros

- All mapping logic is done in the lower layer giving the possibility to gain very good throughput and parallelism.
- The deployment can be designed with a lot more flexibility.
    - Multiple Upper NSO nodes for additional parallelism.
    - Grouping devices on Lower NSO nodes per function to achieve further throughput.
    - The Upper NSO can function as a multi domain orchestrator, dispatching services to different orchestration domains.
    - HA in the deployment will be easier to manage since every NSO node is autonomous and can work in a HA pair, rather than failing over an entire cluster.

LSA Cons

Implementation can be become more complex since logic needs to be done in both the Upper and Lower layers.

Clustering



Clustering is basically a way to scale and shard the device list in NSO. A clustered deployment has one Service NSO and *n* number of Device NSO nodes. Each Device NSO is connected to a distinct group of devices.

The Service NSO has information for all devices managed by Device NSO instances. The configuration is stored in the Device NSO and is fetched by the Service NSO as needed. All Service Mapping logic is done on the service node.

The Service NSO has a complete view of all the devices managed in the cluster. Since the mapping logic is in one place; the service implementation is easy and similar to a non-clustered deployment

The way Clustered deployment scales devices come with a price in performance and throughput. The performance penalty can be big, especially if you have devices with big configurations or very frequent service changes. This becomes even more evident with NSO

services leveraging REACTIVE FASTMAP, e.g. NFV services. The cluster mandates one orchestration, limiting flexibility and cross domain orchestration.

For scalability, we recommend LSA over clusters.

Commit Details

In the NSO CLI it is possible to get detailed information on where in NSO time is spent during a commit, an example.

```
admin@ncs(config)# commit | details
entering validate phase for running...
  validate: run pre-trans-lock service callbacks...
  validate: run transforms and transaction hooks... ok [0.00 sec]
  validate: pre-trans-lock service callbacks done [0.00 sec]
  validate: grabbing transaction lock... ok [0.00 sec]
  validate: creating rollback file... ok [0.00 sec]
  validate: run transforms and transaction hooks... ok [0.10 sec]
  validate: mark inactive... ok [0.00 sec]
  validate: pre validate... ok [0.00 sec]
  validate: run validation over the change set...
  validate: validation over the change set done [0.00 sec]
  validate: run dependency-triggered validation...
  validate: dependency-triggered validation done [0.00 sec]
  validate: check configuration policies...
  validate: configuration policies done [0.00 sec]
entering write phase for running...
  cdb: write_start
entering prepare phase for running...
  cdb: prepare
  ncs-internal-device-mgr: prepare
  ncs: prepare: calculating southbound diffs...
  ncs: prepare: calculating southbound diffs done [0.16 sec]
  ncs: device ios0: send NED prepare
  ncs: device ios1: send NED prepare
  ncs: device ios1: send NED commit
  ncs: device ios0: send NED commit
entering commit phase for running...
  cdb: commit
  cdb: delivering commit subscription notifications at prio 1
  cdb: all commit subscription notifications acknowledged
  ncs-internal-device-mgr: commit
  ncs: device ios0: send NED persist
  ncs: device ios1: send NED persist
Commit complete.
```

Performance Affecting Settings in NSO

Please check the NSO documentation for details on these settings.

Rollbacks

Using rollbacks can affect the performance. To disable rollbacks in NSO, disable them in ncs.conf.

```
<rollback>
  <enabled>false</enabled>
  …
</rollback>
```

Logs

Most logs in NSO can be enabled and disabled in ncs.conf.

Every XPATH evaluation in NSO can be logged to file. While this can be used to find issues and potentially identify performance issues, the tracing itself has a negative effect on performance.

```
<xpath-trace-log>
  <enabled>false</enabled>
  …
</xpath-trace-log>
```

The developer log can we set to only log errors

```
<developer-log-level>error</developer-log-level>
```

The audit log can be disabled

```
<audit-log>
  <enabled>false</enabled>
  …
</audit-log>
```

NETCONF trace log

```
<netconf-trace-log>
  <enabled>false</enabled>
  …
</netconf-trace-log>
```

Please also see “man ncs.conf” for additional logs that can be enabled or disabled.