FRINX

FRINX

AUTOMATE NETWORKS FASTER

FRINX CASE STUDY

Network Service Automation

FRINX

Table of Contents

1 EXECUTIVE SUMMARY

In this document, we introduce the FRINX Machine solution and discuss how it is used by customers to automate their network services and address their business needs.

About FRINX

FRINX builds software that enables customers to create automated, repeatable, digital processes to build, grow and operate their digital communication infrastructure. Enterprise and service provider customers are choosing FRINX products and solutions to automate cloud assets, branch offices, core, edge and access networks. The goal of automation is to provide programmatic interfaces to customers (internal or external), to save time and resources in deploying infrastructure changes and to provide a cost-effective basis for adding new functionality in the infrastructure. FRINX provides software that enables low-code workflow design and operation, analytics to support machine learning and intent based infrastructure control to integrate devices and services from many networking vendors. FRINX solutions are operated and deployed by industry leaders like Facebook, SoftBank, Vodafone and other Global Fortune 500 companies to support their automation needs. FRINX is a privately held company with offices in Bratislava, Slovakia and New York, NY.

2 WHAT PROBLEM DO WE SOLVE?

FRINX builds SW products and solutions to deliver real and sustainable productivity gain by automating processes required to build, operate and grow communication networks.

Typical examples are the automation of services that span resources in the cloud and physical assets, the automation of capacity increases in mobile networks, the activation and change management of L2 and L3VPN services, the management of Internet and Infrastructure services and the automation of core and access network functions.

We have created a solution called FRINX Machine with the purpose of automating network services. FRINX Machine is based on open source components and consists of the following products: UniConfig for network control, UniFlow for creating and operating workflows and UniResource for managing an inventory of physical and logical assets and resources.

3 INTRODUCTION TO FRINX MACHINE

3.1 Overview & Architecture

FRINX Machine enables customers to create automated, repeatable, digital processes to build, grow and operate their digital communication infrastructure. FRINX Machine is based on open source components and enables infrastructure and network engineers to manage service intent, implement configuration changes and obtain operational data from their heterogeneous networks and clouds.

FRINX Machine includes UniFlow, a workflow automation engine, a GUI-based workflow builder for low-code or no-code workflow design and operation, a microservice layer that comes pre-loaded with key automation functions in Python and that is extensible in any programming language. In addition, FRINX Machine includes the UniConfig network control layer that uses an open-source device library to connect to network elements via NETCONF, CLI and gRPC. FRINX Machine further includes UniResource an inventory management tool for physical and logical assets and resources. In support of these functions, FRINX Machine includes a persistence layer including REDIS, Elasticsearch, PostrgreSQL/CockroachDB and GraphQL/MySQL.

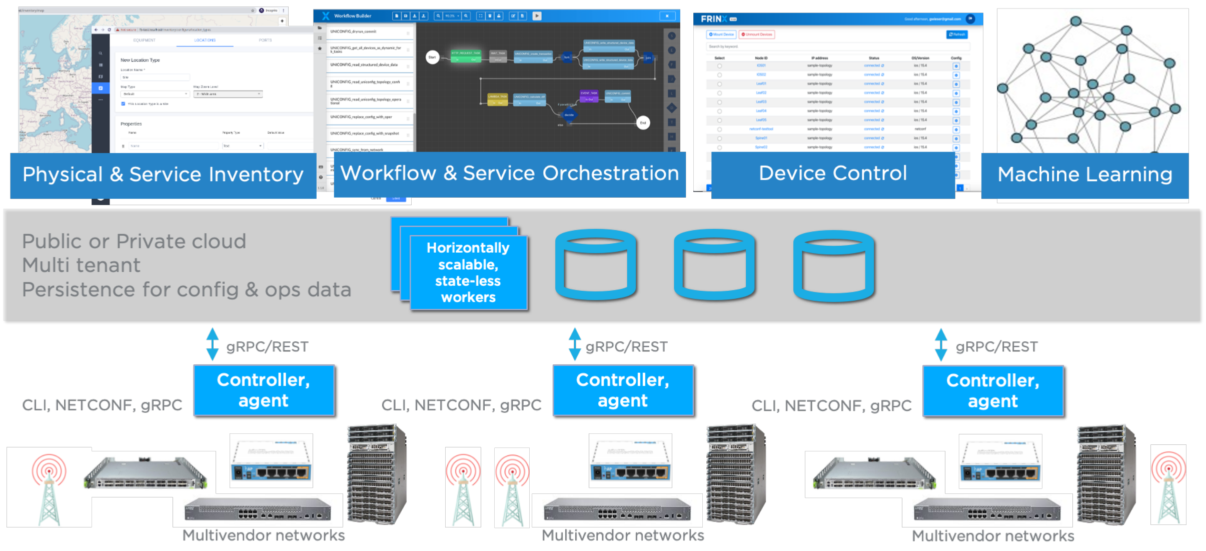


FIGURE 1: FRINX ARCHITECTURE

3.2 FRINX Machine Components

FRINX Machine consists of multiple open source-based components that are packaged in containers. The following components are included in FRINX Machine:

- FRINX UniConfig
  - Connects to the devices in network
  - Retrieves and stores configuration from devices
  - Pushes configuration data to devices
  - Builds diffs between actual and intended config to execute atomic configuration changes
  - Retrieves operational data from devices
  - Manages transactions across one or multiple devices
  - Translates between CLI and private model and standard data models (OpenConfig) via open source device library
  - Reads and stores private data models from network devices (any YANG model)
  - Provides high availability
  - Provides parallel command execution on devices
  - UniConfig UI to interact with the network controller

- FRINX UniFlow
  - Chains atomic tasks into complex workflows
  - Defines, executes and monitors workflows (via REST or UI)
  - Sources: https://github.com/Netflix/conductor
  - Docs: https://netflix.github.io/conductor
  - Enables users to create, edit and run workflows and monitor tasks
  - Enables users to mount and view device status and configuration. Provides access to UniConfig operations like read, edit, commit configurations and sync it from network.
  - View inventory, workflow execution, metadata and UniConfig log files

- FRINX UniResource
  - Provides inventory functions for physical and logical assets
  - Stores device and service inventory data
  - Provides a resource manager function for infrastructure consumables (IP addresses, RTs, RDs,
  - Persistence layer with support for GraphQL, MySQL, PostgreSQL, CockroachDB, Elasticsearch, MongoDB

3.3 FRINX UniFlow

UniFlow is based on the conductor workflow engine developed and open-sourced by NETFLIX. We chose Conductor because it is used in a high scale, high visibility production environment and has performed demonstrably well. Conductor has proven to be a scalable open-source technology and it integrates seamlessly with other components in FRINX Machine.

In addition to the workflow engine, FRINX has built a workflow builder frontend that allows users to create workflows with low or no code. The workflow builder UI allows the design, test, scheduling and execution of workflows via a single interface. All workflows that are created in the workflow builder can be accessed via REST API with role-based access control.

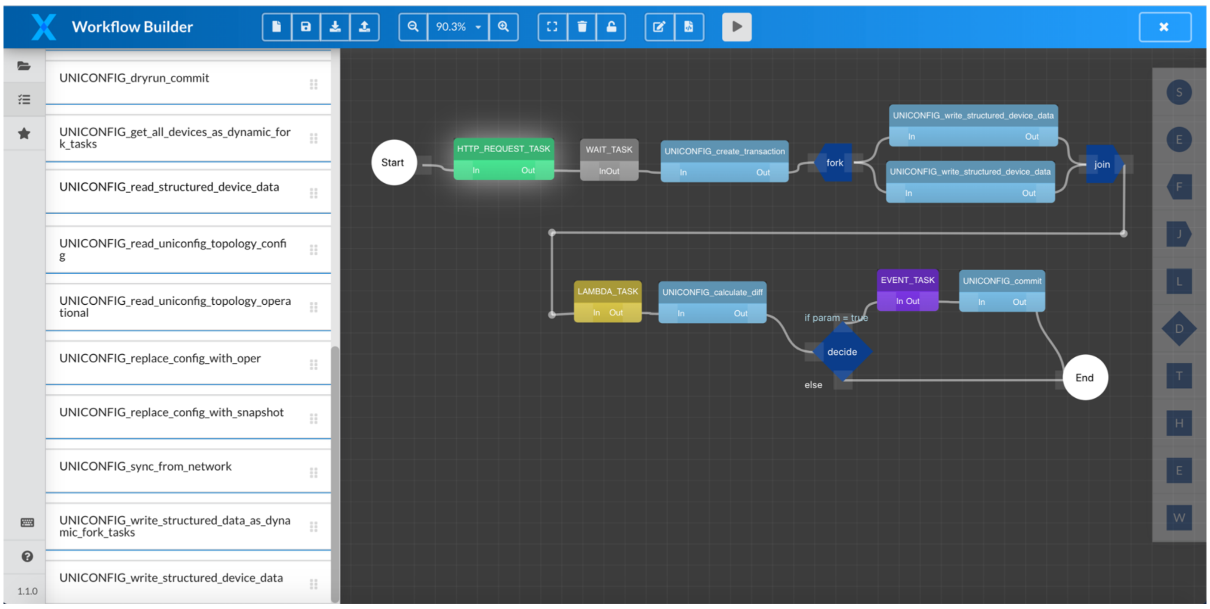
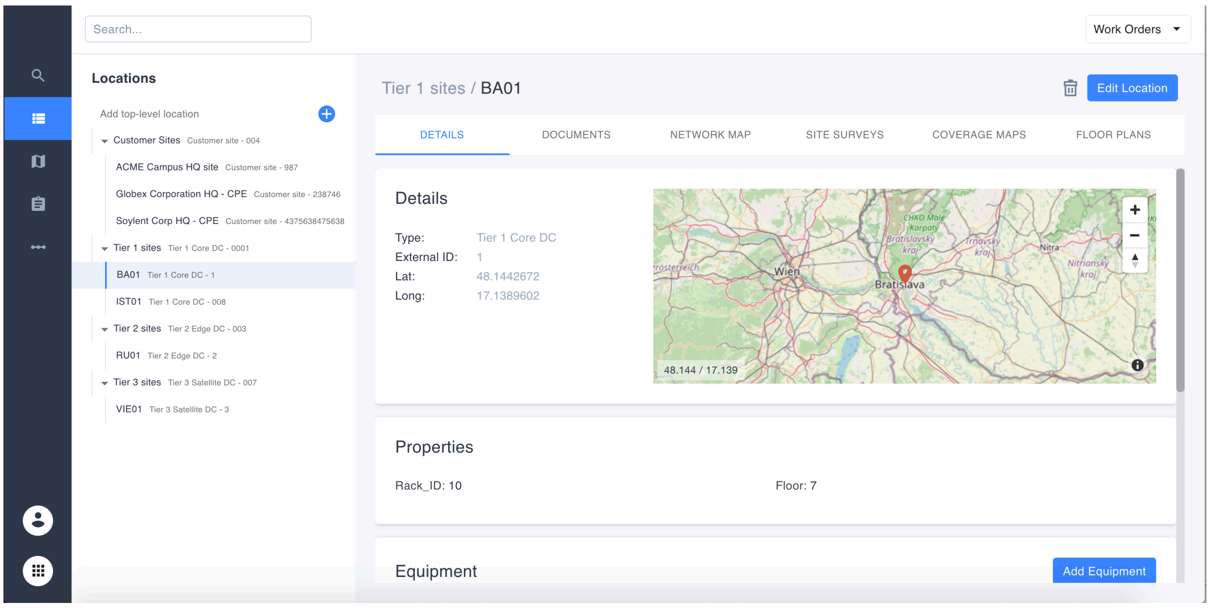

Workflow Builder
90.3%
UNICONFIG_dryrun_commit
UNICONFIG_get_all_devices_as_dynamic_fork_tasks
UNICONFIG_read_structured_device_data
UNICONFIG_read_uniconfig_topology_config
UNICONFIG_read_uniconfig_topology_operational
UNICONFIG_replace_config_with_oper
UNICONFIG_replace_config_with_snapshot
UNICONFIG_sync_from_network
UNICONFIG_write_structured_data_as_dynamic_fork_tasks
UNICONFIG_write_structured_device_data
1.1.0
Start
HTTP_REQUEST_TASK
WAIT_TASK
UNICONFIG_create_transaction
fork
UNICONFIG_write_structured_device_data
UNICONFIG_write_structured_device_data
join
LAMBDA_TASK
UNICONFIG_calculate_diff
if param = true
decide
else
EVENT_TASK
UNICONFIG_commit
End

FIGURE 2: UNIFLOW - WORKFLOW BUILDER UI

3.4 FRINX UniResource

FRINX UniResource was developed for network operators and infrastructure engineers to manage their physical and logical assets and resources. Examples for assets are locations, equipment, ports and services. Examples for resources are IP addresses, VLAN IDs and other consumables required for operating data services. UniResource was developed specifically to address the needs of network and infrastructure engineers working with communication networks.

FRINX UniResource provides GUI and a GraphQL based API with a Python client library to create, read, update and delete assets.



Search...

Work Orders

Locations

Add top-level location

- Customer Sites Customer site - 004
  - ACME Campus HQ site Customer site - 987
  - Globex Corporation HQ - CPE Customer site - 238746
  - Soylent Corp HQ - CPE Customer site - 4375638475638
- Tier 1 sites Tier 1 Core DC - 0001
  - BA01 Tier 1 Core DC - 1
  - IST01 Tier 1 Core DC - 008
- Tier 2 sites Tier 2 Edge DC - 003
  - RU01 Tier 2 Edge DC - 2
- Tier 3 sites Tier 3 Satellite DC - 007
  - VIE01 Tier 3 Satellite DC - 3

Tier 1 sites / BA01

Edit Location

DETAILS | DOCUMENTS | NETWORK MAP | SITE SURVEYS | COVERAGE MAPS | FLOOR PLANS

Details

Type: Tier 1 Core DC
External ID: 1
Lat: 48.1442672
Long: 17.1389602

48.144 / 17.139

Properties

Rack_ID: 10

Floor: 7

Equipment

Add Equipment

FIGURE 3: FRINX UNIRESOURCE - INVENTORY MANAGER

3.5 FRINX UniConfig

The purpose of UniConfig is to manage the intent (desired state) of physical and virtual networking devices through a single network API. In addition, UniConfig enables device and network wide transactions, so that the network will always remain in a well-defined state without leftovers from failed configuration attempts. UniConfig can be run as an application on bare metal in a VM or in a container. UniConfig has a built-in data store that can be run in memory or with persistence on disk and to external database.

UniConfig enables users to communicate with their network infrastructure via four options:

1) **Execute & Read API** - Unstructured data via SSH and Telnet
2) **OpenConfig API** – Translation provided by our open source device library
3) **UniConfig native API** – Direct access to vendor YANG models native to the connected devices plus UniConfig functions like diff, commit and snapshots
4) **UniConfig native CLI API** – Programmatic access to CLI without the need for translation units

Option 1) gives users similar capabilities like access through Ansible, TCL scripts or similar tools and allows to pass strings to the device and receive strings from the device via REST in a programmatic way. UniConfig provides the mechanism to authenticate and provide a channel to send and receive data but does not interpret the data.

Option 2) provides users with an OpenConfig API that is translated into device specific CLI or YANG models. This requires "translation units" to be installed for the devices under control. FRINX provides an open source device library that includes many devices from widely deployed network vendors. Anyone can contribute and consume content of this library providing an ever-growing list of supported devices.

Option 3) "UniConfig native" provides the ability to configure devices with any YANG model that is supported by the device. After mounting a device, UniConfig native maps the vendor models into its UniConfig data store and provides stateful configuration capabilities to applications and users.

Option 4) "UniConfig native CLI" provides the ability to interact with device CLI in a programmatic way. No translation units are required, only a schema file has to be provided. This option provides programmatic access to devices without the need for writing translation units.

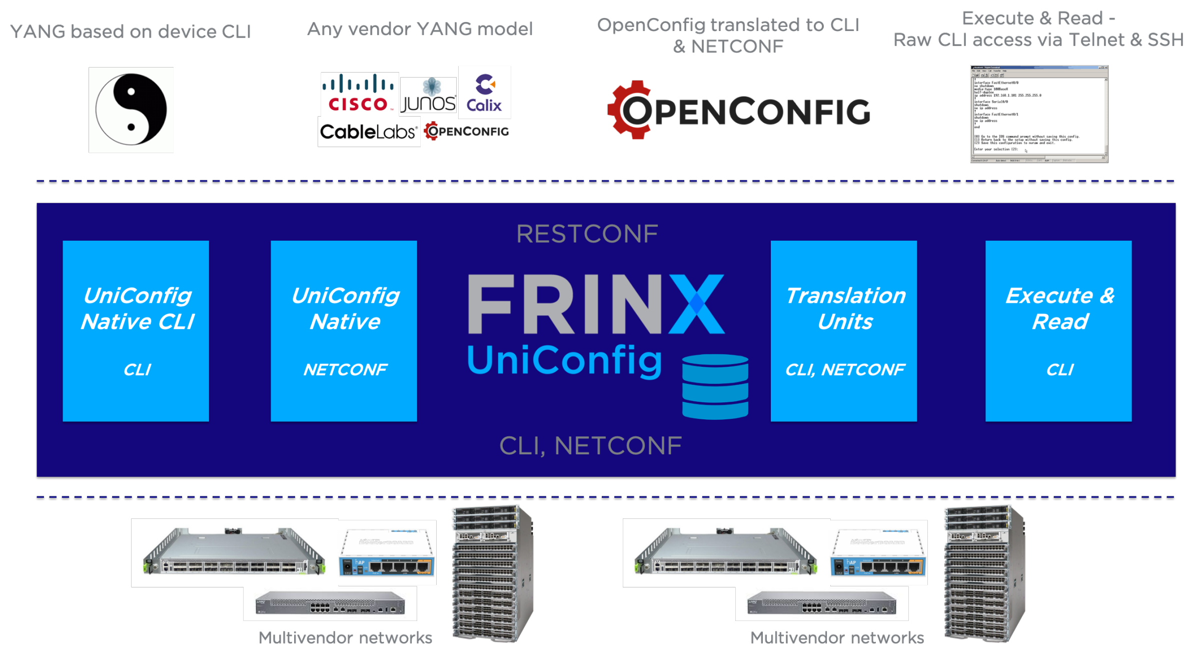


FIGURE 4: UNICONFIG APIS

UniConfig consists of three layers that can be accessed individually or via the UniConfig node manager API.

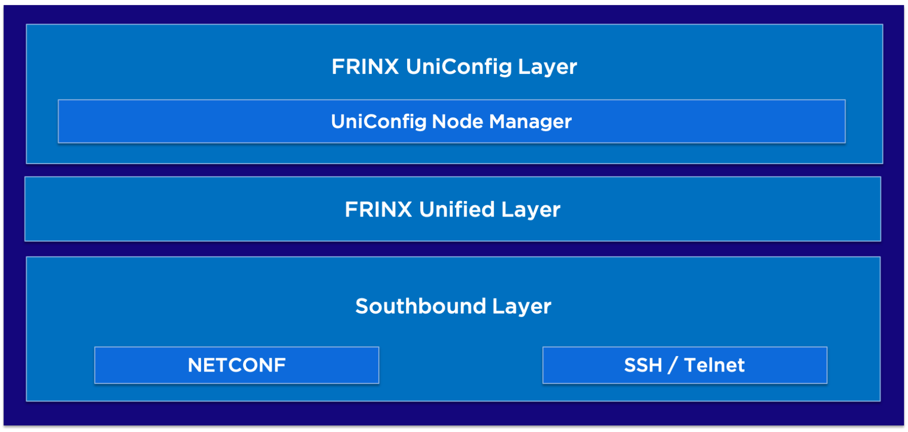


FIGURE 5: UNICONFIG LAYERS

The lowest layer (Southbound layer) provides connectivity to the network devices through NETCONF and CLI via Telnet and SSH. It provides transparent access to CLI devices and It includes translation units that map data in OpenConfig format to vendor specific CLI implementations and vice versa. This translation unit library is open source and publicly available on GitHub (https://github.com/FRINXio/cli-units).

The middle layer (Unified layer) combines devices mounted under various protocols (e.g. NETCONF and CLI) and makes them accessible under a unified mount point to higher layers. This provides a layer of abstraction between the southbound protocols and the user intent that is to be applied to the network. The unified layer also provides native YANG models as well as OpenConfig YANG to vendor specific YANG translation capabilities. This translation unit library is open source and publicly available on GitHub (https://github.com/FRINXio/unitopo-units)

The top layer (UniConfig layer) provides the ability to read and write YANG based configurations to and from devices. It also adds the capability to create configuration snapshots that can be committed and can be rolled back by the system if they have failed. The UniConfig layer also adds the capabilities to compare network intent (located in the configuration data store), analyze the diffs between intended state and actual state (located in the operational data store) and finally apply the new state to the devices connected through lower layers via atomic operations (commit). This means that only configuration elements that were changed in the most recent operation need to be sent to the devices and not the complete configuration, saving resources and allowing for higher transaction throughput. The UniConfig layer also adds capabilities to build snapshots of all or a subset of devices and move from the current configuration to any snapshot in one single transaction. Finally, the UniConfig layer includes the "Dry-Run manager" that allows testing of NETCONF and CLI configuration changes before they are applied to the network.

The functionality of the UniConfig layer is accessible via a REST interface and documented via Swagger API documentation. In addition, we provide Java, Python and GO client libraries. Those client libraries make the UniConfig API available through popular programming languages and allow users to build applications using the UniConfig functionality without having to interact with the REST API directly.

3.5.1 UniConfig key features

Reconciliation

We built a system that reconciles the configuration on each connected device. UniConfig reads and translates vendor specific device configurations and represents them internally via native YANG, OpenConfig or native CLI YANG data models. This functionality allows users to sync device configurations with the data store in UniConfig, compare and create diffs between intended and actual state and between platforms.

Open Source device libraries

Customers and integration partners can freely contribute, modify and create additional device models that work with UniConfig. If they wish, they can contribute the translation code and schemas, so they become available for all other users of UniConfig. The UniConfig device library includes 280 person months of coding effort from FRINX customers, partners and employees and provides support for 25 vendor platforms as of today (Q2CY20).

Network transactions

UniConfig offers the ability to perform transactions on a single or across multiple network devices. This provides the benefit for users that the network devices can be rolled back to the state before the configuration attempt should one device within the transaction fail.

Native YANG & OpenConfig

UniConfig supports OpenConfig YANG data models, vendor specific YANG data models and native CLI YANG data models. Using OpenConfig simplifies the configuration portability between network devices and simplifies the development of applications that configure heterogeneous networks.

Sync-from-network, commit, transaction, rollback and snapshots

UniConfig provides the necessary RPCs (Remote Procedure Calls) to implement the following basic functions to automate the configuration of network devices: Sync-from-network reads configuration from network devices to the controller. Commit writes atomic configuration data to network devices. Transactions and rollback enable to configure one or multiple devices at the same time with the ability to revert back to the state before the configuration attempt if the transaction fails. Snapshots capture the configuration state of network devices at a point in time.

REST API & Client libraries (Java, Go, Python)

UniConfig offers a REST API and client libraries implemented in Java, Python and Go. The UniConfig API is documented in Swagger and Postman.

3.5.2 UniConfig Components

UniConfig components use a layered design where the functionality of the upper layers depends on the functionality of the layer underneath. Each layer thus provides a higher level of abstraction from the network elements.

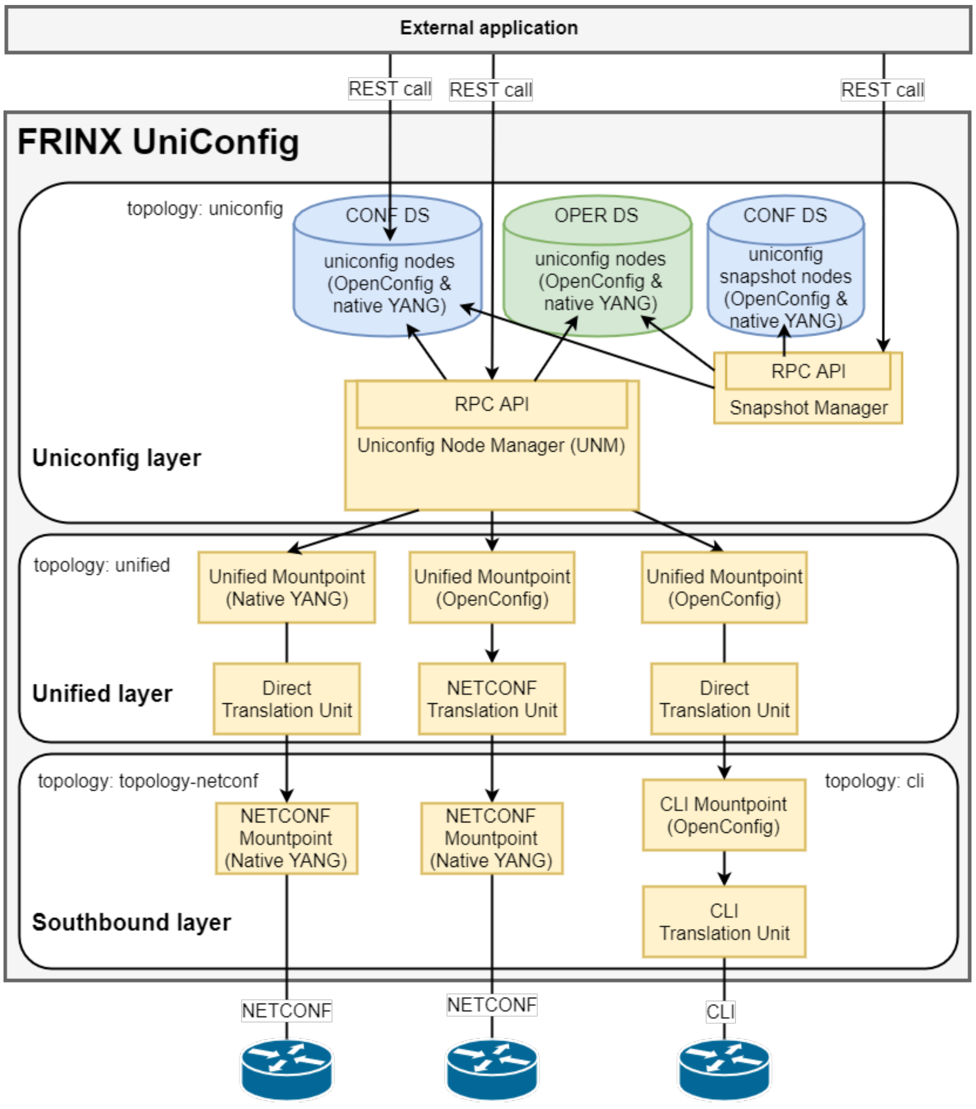

External application
REST call
REST call
REST call
FRINX UniConfig
topology: uniconfig
CONF DS
uniconfig nodes (OpenConfig & native YANG)
OPER DS
uniconfig nodes (OpenConfig & native YANG)
CONF DS
uniconfig snapshot nodes (OpenConfig & native YANG)
RPC API
Snapshot Manager
RPC API
Uniconfig Node Manager (UNM)
Uniconfig layer
topology: unified
Unified Mountpoint (Native YANG)
Unified Mountpoint (OpenConfig)
Unified Mountpoint (OpenConfig)
Unified layer
Direct Translation Unit
NETCONF Translation Unit
Direct Translation Unit
topology: topology-netconf
NETCONF Mountpoint (Native YANG)
NETCONF Mountpoint (Native YANG)
CLI Mountpoint (OpenConfig)
topology: cli
Southbound layer
CLI Translation Unit
NETCONF
NETCONF
CLI

FIGURE 6: UNICONFIG ARCHITECTURE

Applications can utilize any of the layers in the system. There are 3 main layers represented by these components (from top to bottom):

- UniConfig layer (UniConfig Node Manager with data store)
- Unified layer (Unified mount point with translation units)
- Southbound layer (NETCONF mount point, CLI mount point with translation units)

The data store is a component in UniConfig which stores structured data described by YANG models. There are two separate data stores:

- Config data store (CONF DS) - contains intended state (intended device configuration). This data store is persistent and external (outside ODL) applications have read/write access.
- Operational data store (OPER DS) - contains actual state (actual device configuration). OPER DS is not persistent and external applications have read only access.

Mount points in UniConfig represent a communication interface with an external system. Mount points are usually registered under a node in a topology.

3.5.2.1 CLI Mount Point

The CLI mount point provides a management API for a network device over the CLI. OpenConfig models are used for structured data describing the device configuration and state. The CLI mountpoint uses CLI translation units for translation between OpenConfig data and CLI data. The CLI mount point API supports device transactions and automatic rollback functionality (in case an error occurs during device configuration). CLI mount point is registered under a node in CLI topology. Each CLI mount point always includes a generic CLI translation unit which provides an RPC for sending raw CLI commands and returning raw CLI output.

3.5.2.2 CLI Translation Units

A CLI translation unit defines the mapping between YANG models and the CLI for a specific device type and software version. It is used by the FRINX ODL controller to perform translations between device specific CLI data and standardized structured (OpenConfig YANG) data. The translation unit can read and write configuration or read the state of a device. It uses the CLI over SSH or telnet for communication with the CLI device. The CLI translation unit is usually created for a combination of device type and OpenConfig main section (folder) e.g. ios-local-routing, ios-ospf, etc.

3.5.2.3 CLI Dry-run Mount Point

The CLI dry-run mount point mocks the management API for a network device over CLI. It uses the CLI dry-run journal for storing to-be-executed CLI commands instead of configuring the network device directly. Just as with a regular CLI mount point, it uses the same set of CLI translation units and the same set of OpenConfig YANG models.

3.5.2.4 NETCONF Mount Point

The NETCONF mount point provides a management API for the network device over a NETCONF session. Data are usually described by a set of vendor specific YANG models. The NETCONF mount point provides device transactions and rollback (if supported by the device). The NETCONF mount point is registered under a node in topology-netconf topology.

3.5.2.5 Unified Mount Point

The Unified mount point unifies the API for various southbound protocols like NETCONF and CLI. The API is described using OpenConfig YANG models and uses translation units to translate between OpenConfig data and southbound mount point data. The Unified mount point is registered under a node in unified topology and is created automatically.

3.5.2.6 Direct Translation Unit

This unit simply passes OpenConfig data to any mount point with OpenConfig available capabilities. This is possible because northbound data and southbound data are described by the same OpenConfig YANG model.

3.5.2.7 UniConfig Native

UniConfig native allows to communicate with network devices using their native YANG data models (e.g.: Cisco YANG models, JunOS Yang models, CableLabs YANG models, ...). UniConfig native allows you to use the same features with native YANG models as with regular UniConfig OpenConfig models (e.g. sync-from-network, commit, checked-commit, calculate-diff, replace-config-with-operational, snapshots). UniConfig native works alongside CLI and NETCONF UniConfig translation units. This means users can mount some devices as UniConfig native using their vendor specific YANG models while they mount other devices on the same server as UniConfig devices using translation units. UniConfig native is available starting with FRINX ODL release 4.2.0.

3.5.2.8 NETCONF Translation Unit

The NETCONF translation unit translates OpenConfig data to data described by device specific YANG models. It uses the NETCONF mount point for communication with a NETCONF device, and implements device transaction with automatic rollback if not provided by the device itself.

The JunOS NETCONF translation unit is a simplified example. The NETCONF translation unit is usually created for a combination of device type and OpenConfig main module (e.g. ospf, bgp, network-instance, rib, acl, qos, …). An example is xr-6-network-instance, xr-6-ospf, etc.

3.5.2.9 UniConfig Node Manager

The responsibility of this component is to provide a well-defined network API to the applications north of FRINX ODL and to maintain the configuration on devices based on intended configuration and operational data.

The UniConfig config data store provides a non-blocking high performance API that supports CRUD operations. Applications write their updates into the UniConfig config data store and call commit on one, multiple or all devices. The commit operation is scheduled in parallel across non-overlapping sets of devices. The application does not have to deal with device irregularities and receives feedback about the success of the commit operation.

Each device and its configuration is represented as a node in the UniConfig topology and the configuration of this node is stored based on YANG models (OpenConfig and native YANG models). The Northbound API of the UniConfig Node Manager is RPC (Remote Procedure Call) driven and provides functionality for commit with automatic rollback, manual rollback and synchronization of configuration from the network.

When a commit is called, the UniConfig Node Manager creates a diff based on intended state from the UniConfig CONFIG data store and actual state from the OPER data store. This diff is used as the basis for device configuration. The UniConfig Node Manager prepares a transaction on one or on multiple devices and uses the unified mount points to communicate with different types of devices.

In the case where the configuration attempt on one device fails, the UniConfig Node Manager executes automatic rollback across all devices involved in that transaction: The previous configuration is restored on all modified devices. Manual rollback enables simple reconfiguration of the entire network using one of the previous states saved in the UniConfig Node Manager. Synchronization from the network reads configuration from devices and stores it as an actual state to the OPER DS.

3.5.2.10 Dry-run Manager

The dry-run manager provides functionality for mock configuration of CLI and NETCONF devices where CLI or NETCONF commands are sent to the dry-run journal instead of the device.

The dry-run manager uses UniConfig Node Manager for getting the diff of the intended configuration and uses the dry-run mount points for sending CLI or NETCONF commands to the dry-run journal.

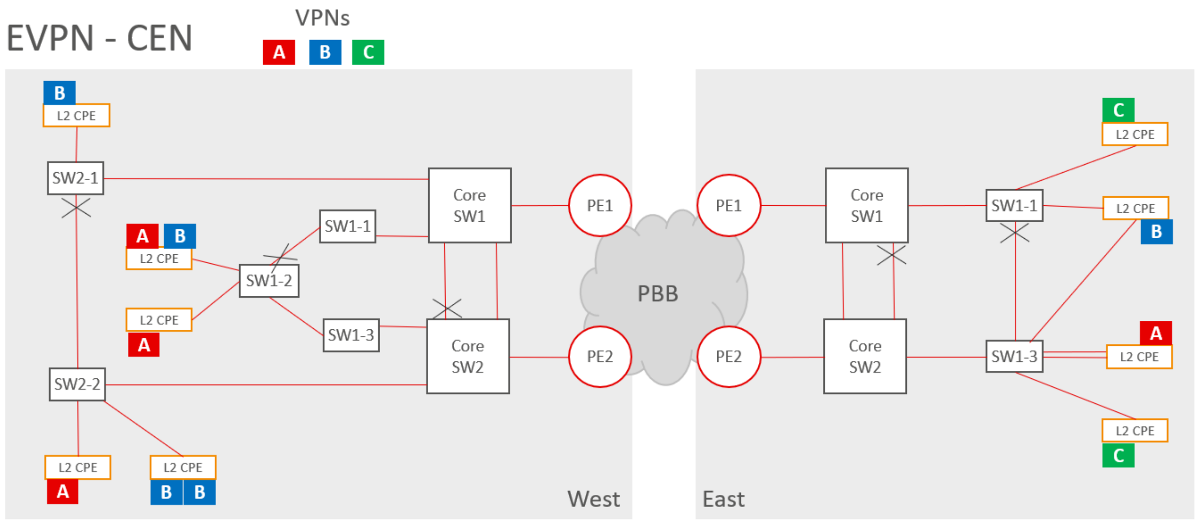
4 USE CASES

4.1 EVPN Service Automation

4.1.1 Overview

The EVPN use case implements reconciliation (reading of existing EVPN configuration from devices and parsing it to service data) and provisioning of the EVPN service (translating service data to device specific EVPN configuration). The EVPN service is deployed on PE devices (vendor x, vendor y), on layer 2 devices (vendor z) in multiple access rings and on CPEs (vendor x, vendor z). All devices have to be provisioned correctly for the service to be functional.

The EVPN service is deployed on an existing brownfield network and automation was added after customers have already been in service. Reconciliation had to be deployed as a first step to provide visibility into existing service configurations. Once reconciliation was tested and implemented successfully, provisioning of new services was automated.



Regions (West/East) includes PE routers (PE1, PE2), the main ring (Core SW1, Core SW2), multiple subrings (SW1-2, SW1-1, SW1-3; SW2-1, SW2-2), and multiple CPEs (L2 CPE) connected to subring switches. The core of the network is implemented with Provider Backbone Bridge technology (PBB).

Automation was implemented on PEs, core switches, ring switches, and CPEs. The PBB layer between PEs is not part of the automation solution in this implementation phase since it is statically configured, and it is changed infrequently.

Reconciliation

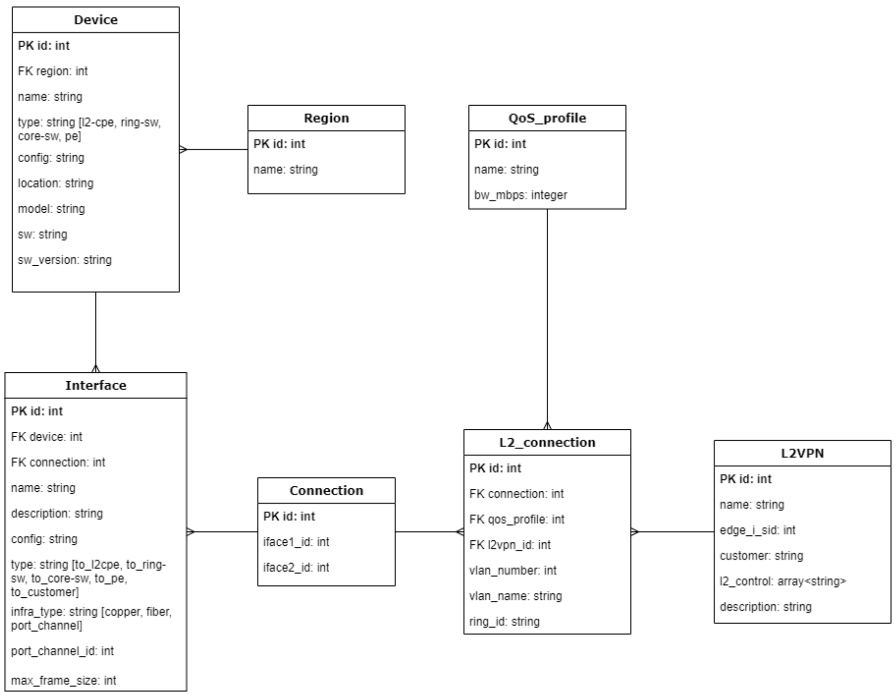
Reconciliation capability is critical in this case, because the solution runs on a brown field network (devices are already configured and provide services to CPE customers). That means the implementation has to read the configuration from all device types, recognize EVPN services and finally store that information as service data to the inventory database.

Provisioning

The main goal of EVPN provisioning is to attach or delete customers on a CPE port or change existing customer configurations. The demarcation point between a customer's and the operator's network is a physical interface on the CPE. CPE tags traffic from the customer with a VLAN ID that is specific to the customer in a region. That means VLANs are region specific. A single customer can have connections in different regions and VLANs can be different in each region for that customer. Attaching a new customer requires the configuration of the CPE, ring-switches composing a ring where the CPE is attached, core-switches where the ring is terminated and PE routers where core-switches are connected. In the case where an existing customer, that already has a connection in the region and ring, connects an additional CPE in that ring, only the CPE needs to be configured.

4.1.2 Service Models

The service model represents the EVPN service instances configured on devices in the regions.



The service model has been implemented in the FRINX Machine inventory. The inventory was designed to store the information about EVPNs, devices where the service is configured, and information about the connections between devices.

Each network element (PE, core-switch, ring-switch, CPE) is represented as a device. Physical and aggregated interfaces are stored in interface table. The connection table contains point to point connections between these interfaces and references multiple logical L2 connections. QoS profile and L2VPN tables contain information about customer specific settings of EVPN.

4.1.3 Workflows

Workflows are used as northbound REST API towards the operator's IT systems. Workflows provide the business logic that is required for execution of the reconciliation and the provisioning tasks. Reconciliation uses workflows that are specific to the device type (PE, core-switch, ring-switch, CPE). Before the reconciliation starts, devices need to be registered in the inventory. Reconciliation can be scheduled to run periodically for all devices, or per device type, or per region where the devices are located. The capacity to perform reconciliation can be scaled up with additional workers.

Provisioning workflows allow the operator to create a new EVPN service for a new or an existing customer by defining the interface of the CPE that the customer is connected to and the service attributes that are part of the contract.

4.1.4 Translation Units

Devices used as CPEs, ring-switches, and core-switches do not support the NETCONF management interface or the implementation of the NETCONF server is not working reliably on those devices. Therefore, CLI translation units have been developed for reconciliation and provisioning of EPVN via UniConfig. PE devices have full support of NETCONF protocol so the UniConfig native API was used.

4.1.5 High Availability

FRINX Machine is deployed together with two HAProxy servers. Keepalived runs on the HAProxy servers and is used to coordinate which HAProxy server should be the MASTER and which the BACKUP. The MASTER uses a floating IP, that is defined in the Keepalived config file. Keepalived uses the VRRP protocol. In case the MASTER node becomes unavailable, the BACKUP node assigns the floating IP address to itself and sends a gratuitous ARP reply.

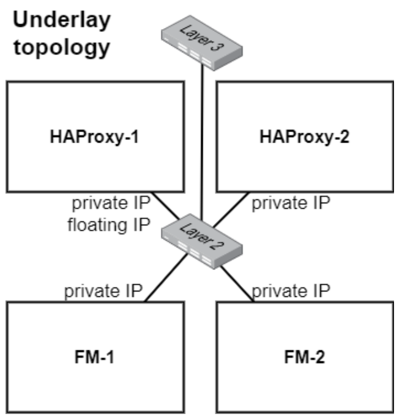


FIGURE 7: FRINX MACHINE HIGH AVAILABILITY

HAProxy forwards the requests to the UniConfig-UI component of FRINX-machine. Since there are two FRINX-machine instances running, it is set up in such a way, that all services running on the first instance have to be available in order the requests to be forwarded here. If any of the services become unavailable, requests are going to be forwarded to second FM instance.

An auto-heal service was implemented to automatically restart containers that have become unresponsive. Each container, that is being health-checked, runs a command that determines if the service that it is running is in a healthy state. In case it isn't, the auto-heal service will restart it.

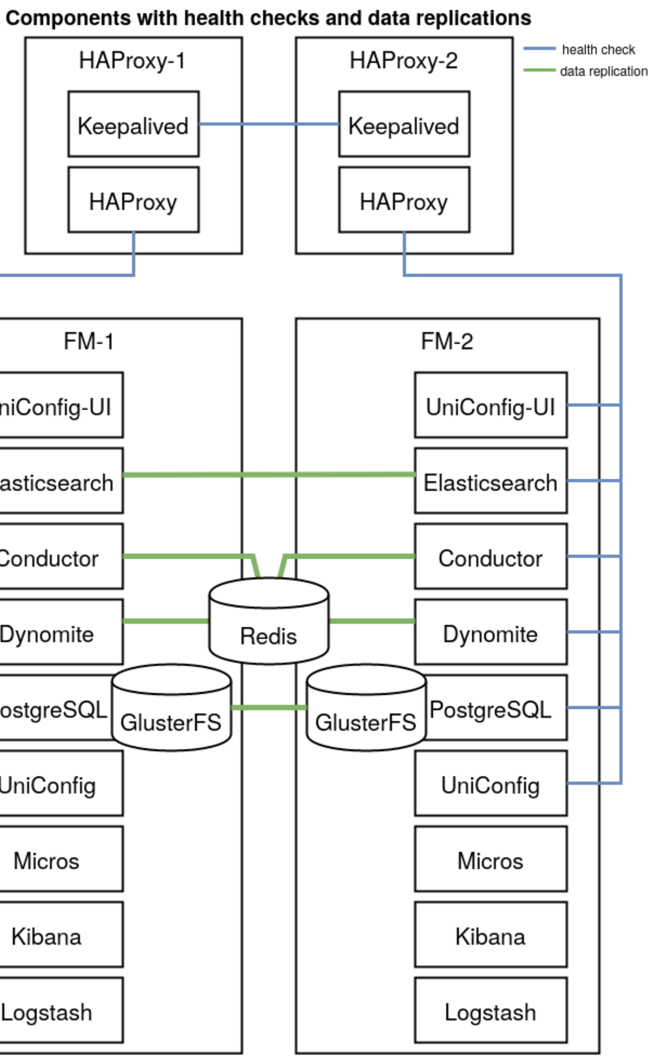


FIGURE 8: FRINX MACHINE DATA REPLICATION

GlusterFS is used to replicate the file system in use by PostgreSQL. We use the clustering technology built-in to Elasticsearch and Dynomite/REDIS to replicate data between the two instances. The remaining services are state-less.

4.2 VPLS, P2P Ethernet & Internet Service Automation

In this use case, the operator has deployed a Metro Ethernet network of IP/MPLS based Ethernet switches. The operator offers VLL (Virtual Leased Line - a point to point Ethernet service), VPLS (Virtual Private LAN services - a multi-point Ethernet service) and Business Internet on their infrastructure to business customers. The automation architecture implemented with FRINX UniConfig is shown in the following diagram:

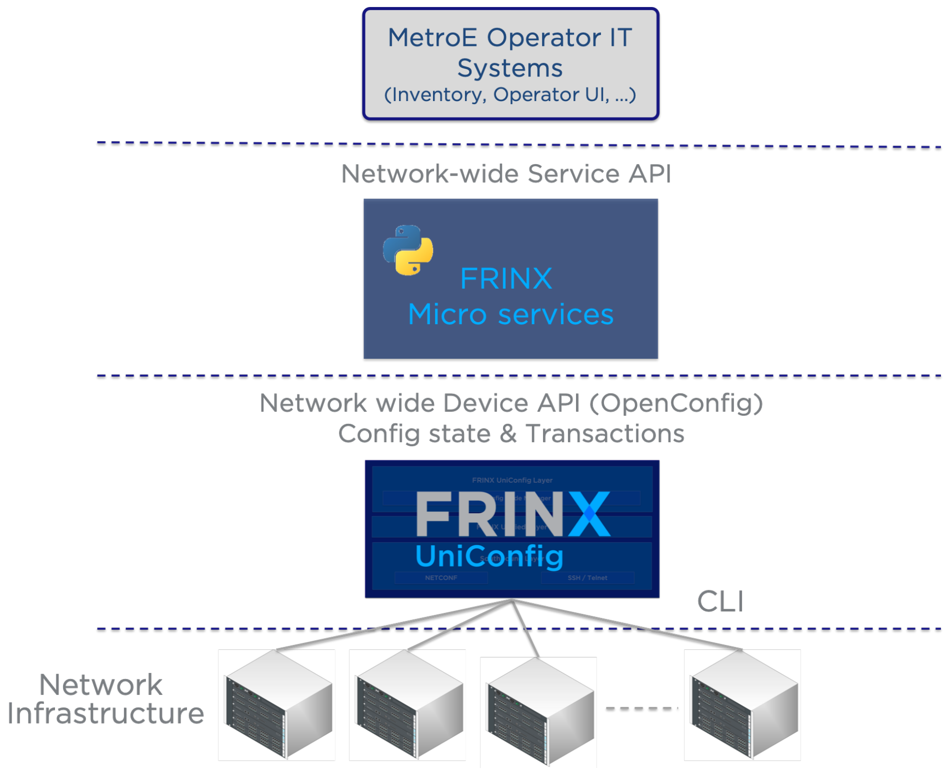

MetroE Operator IT Systems
(Inventory, Operator UI, ...)
Network-wide Service API
FRINX Micro services
Network wide Device API (OpenConfig)
Config state & Transactions
FRINX UniConfig
CLI
Network Infrastructure

FIGURE 9: OPERATOR AUTOMATION SOLUTION STACK

Starting from the bottom of the stack, we have the network infrastructure, consisting of at least two different models of switches with minor variations in command set. FRINX has created a translation unit for this vendor and hence provides translations from and to OpenConfig to the device specific CLI. UniConfig is capable of reading the device configuration when it connects to the device for the first time and whenever the “sync-from-network” API endpoint is called. This provides the operator with network-wide device configuration state in a common data model (OpenConfig) in one place (the UniConfig data store). When the operator upgrades to or adds a new vendor for their network infrastructure, only a new translation unit has to be added. The rest of the provisioning stack will not be impacted.

The UniConfig layer manages all device configuration changes with transactions and has the ability to perform rollback whenever configurations across one or multiple devices have not completed successfully.

The next higher layer implements a set of micro services in Python, that present a service API to the operator IT systems. The service API has been crafted based on the requirements of the operator and only contains necessary data elements required for the services (VLL, VPLS, BI). The micro service layer is simple to understand and simple to modify by the operator or by FRINX.

The micro service layer can be accessed directly via the operator IT system stack and via the FRINX Machine workflow engine. The latter provides the ability to assemble workflows with the help of a UI and to interact with the micro services as horizontally scalable worker tasks. FRINX Machine also adds persistence for workflow definitions, executions and logs and for inventory and services.

4.2.1 Example service configuration

4.2.1.1 Example VLL service configuration

The following API request body is sent from the operator's IT system to the FRINX micro services. The micro services are state-less and can be horizontally scaled. The micro services transform the request below in a sequence of atomic configurations (configure policies, configure interface and sub-interface, VLANs and P2P connection settings via MPLS). The complexity of applying and maintaining these steps is handled by the FRINX micro services and UniConfig. The operator system only needs to implement a high-level abstraction of the service based on its requirements.

```
{
"service":
{
        "id": "abcd",
        "vccid": 1234,
        "mtu": 1500,
        "devices": [
             {
                "id": "B21",
                "interface": "ethernet 3/4",
                "remote_ip": "100.10.10.100"
             },
             {
                "id": "B28",
                "interface": "ethernet 1/3",
                "remote_ip": "100.10.10.101"
            }
        ]}
}
```

4.2.1.2 Example service API – Read VLL all

The following service API reads all device configurations from UniConfig and reconciles the VLL services that it found in the device configurations. This service API can be executed on all devices or by scoping it with a service ID, only on the devices that include that service ID.

```
PUT
https://<microservice-host-ip>:6454/VLL_service_read_all
{
        "datastore": "actual",
        "reconciliation": "name"
}
```

Output example:

```
{
    "logs": [
        "VLL instances found successfully: 12"
    ],
    "output": {
        "services": [
            {
                "devices": [
                    {
                        "id": "B21",
                        "interface": "ethernet 3/3"
                    },
                    {
                        "id": "B21",
                        "interface": "ethernet 3/4",
                        "vlan": 51
                    }
                ],
                "id": "testL2P2P-local"
            },
[truncated]
            {
                "devices": [
                    {
                        "id": "B21",
                        "interface": "ethernet 3/3",
                        "remote_ip": "14.14.14.14",
                        "vlan": 59
                    },
                    {
                        "id": "B28",
                        "interface": "ethernet 1/22",
                        "remote_ip": "14.14.14.15",
                        "vlan": 59
                    }
                ],
                "id": "testL2P2P-remote-tag",
                "mtu": 1500,
                "vccid": 1000002021
            }
        ]
    },
    "status": "COMPLETED"
}
```

The implementation of the VLL described above as well as VPLS and BI micro services can be found here:

https://github.com/FRINXio/FRINX-machine/tree/master/microservices/netinfra_utils/workers

4.2.2 Example device configuration (OpenConfig)

By using an OpenConfig device API, customers can use a common data model to configure different device types or operating systems. UniConfig implements the translation from OpenConfig to CLI including vendor specific augmentations and extensions.

4.2.2.1 Example device interface configuration

```
{
    "frinx-openconfig-interfaces:interface": [
        {
            "name": "ethernet 3/4",
            "config": {
                "type": "iana-if-type:ethernetCsmacd",
                "enabled": true,
                "description": "FRINX-TEST l2p2p",
                "frinx-brocade-if-extension:priority": 3,
                "frinx-brocade-if-extension:priority-force": true,
                "frinx-openconfig-vlan:tpid": "frinx-openconfig-vlan-types:TPID_0X8A88",
                "name": "ethernet 3/4"
            }
        }
    ]
}
```

4.2.2.2 Example device P2P configuration

The following device configuration template needs to be applied on two ends of the VLL connection within the scope of one network wide transaction.

```
{
    "network-instance": [
        {
            "name": "testL2P2P-remote",
            "interfaces": {
              "interface": [
                     {
                       "id": "ethernet 3/4",
                        "frinx-openconfig-network-instance:config": {
                        "id": "ethernet 3/4"
                     }
                  }
              ]
            },
            "connection-points": {
                "connection-point": [
                    {
                        "connection-point-id": "1",
                        "config": {
                            "connection-point-id": "1"
                        },
                        "endpoints": {
                            "endpoint": [
                                {
                                    "endpoint-id": "default",
                                    "config": {
                                        "endpoint-id": "default",
                                        "precedence": 0,
                                        "type": "frinx-openconfig-network-instance-
types:LOCAL"
                                    },
                                    "local": {
                                        "config": {
                                            "interface": "ethernet 3/4"
                                        }
                                    }
                                }
                            ]
                        }
                    },
                    {
                        "connection-point-id": "2",
                        "config": {
                            "connection-point-id": "2"
                        },
                        "endpoints": {
                            "endpoint": [
                                {
                                    "endpoint-id": "default",
                                    "config": {
                                        "endpoint-id": "default",
                                        "precedence": 0,
                                        "type": "frinx-openconfig-network-instance-
types:REMOTE"
                                    },
                                    "remote": {
                                        "config": {
                                            "remote-system": "14.14.14.15",
                                            "virtual-circuit-identifier": 1000002020
                                        }
                                    }
                                }
                            ]
                        }
                    }
                ]
            },
            "config": {
                "name": "testL2P2P-remote",
                "type": "frinx-openconfig-network-instance-types:L2P2P",
                "mtu": 1500
            }
        }
    ]
}
```

UniConfig translates the OpenConfig configuration data into CLI commands with the correct syntax and sequence required for the vendor, platform and software version.

The implementation of the UniConfig translation units can be found here:

https://github.com/FRINXio/cli-units

5 CONTACTS & RESOURCES

Send us an email at: info@frinx.io

Visit us at the following locations:

https://frinx.io/

https://frinxio.blogspot.com/

https://github.com/FRINXio/

https://frinx.io/contact

https://www.youtube.com/channel/UCeRL-J2ppxdBRs8tdWQz2jA/videos

follow us on Twitter:

@Frinxio

@G_wieser

or call us at: +421 2 209 101 41

Download the FRINX Machine including UniConfig here:

https://github.com/FRINXio/FRINX-machine

FRINX Documentation

https://docs.frinx.io/

List of supported networking devices:

https://docs.frinx.io/frinx-odl-distribution/supported-devices.html