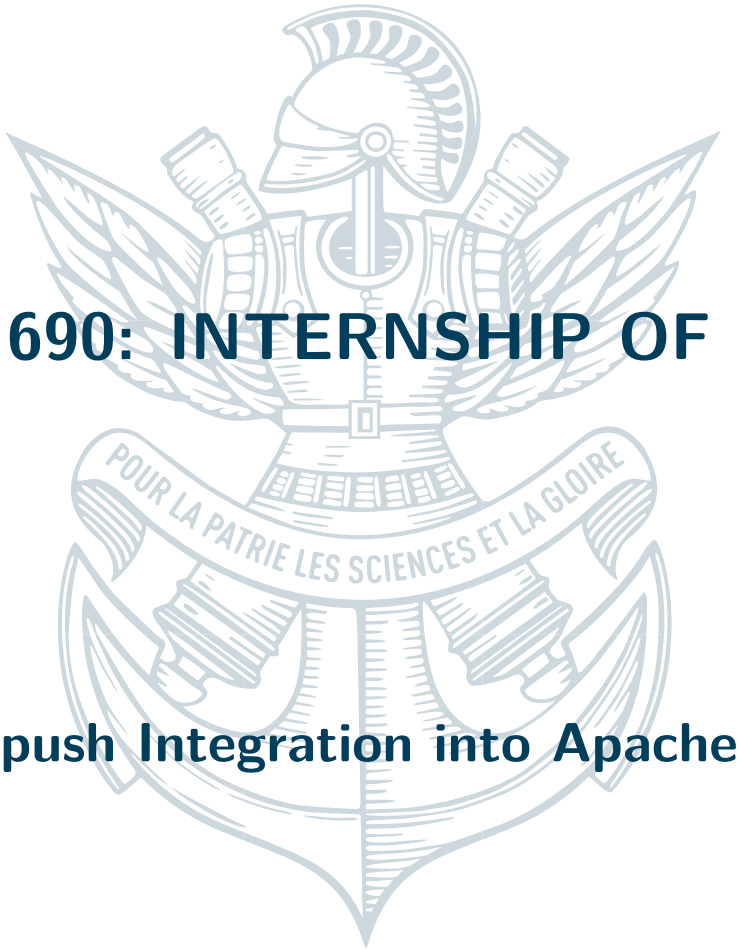

INF 690: INTERNSHIP OF IOT

YANG push Integration into Apache Kafka

September 4, 2023

Author: Zhuoyao Lin

Tutor: Benoit Claise, Jean Quilbeuf

Supervisor: Prof. Juan-Antonio Cordero-Fuertes

30. March. 2023 - 19. September. 2023

ÉCOLE POLYTECHNIQUE

INSTITUT POLYTECHNIQUE DE PARIS

HUAWEI

Acknowledgement

I would like to thank Benoit, Jean, Thomas Joubert, Yanhao, Olga from Huawei, Thomas Graf, Uwe, Ahmed from Swisscom and Alex from INSA-lyon for their valuable support and guidance regarding all aspects of this project. A special thanks goes to Benoit for his guidance and suggestions from the view of an senior network engineer, the opportunity he provided for me participating into IETF Hackathon, as well as his effort for helping me get my career. Another special thanks goes to Jean, who lead me throughout the development of libyangpush, provided me with important technical guidance, and also collaborated with me during the Hackathon demo. I would also like to thank Thomas Joubert for the developing suggestions and the code review.

At l'X, I would like to thank Prof.Thomas Clausen and Prof.Juan-Antonio for their guidance during my two-years study in IOT, and for helping me find my interesting field.

I would also like to thank myself for the perseverance and courage when facing all difficulties.

Abstract

With the incorporation and acceptance of YANG[1] within the router infrastructure, the industry has been able to witness its profound capability in effectively managing network configurations and delineating operational metrics. However, it is currently not possible to stream YANG data into Apache Kafka without losing the semantic. Consequently, there arises the necessity for the development of a system capable of imbuing YANG data with semantic significance, and enabling real-time data processing and analysis.

This thesis discusses a data mesh system[2] built upon Apache Kafka[3] and that combines the YANG and YANG push concepts[4][5]. The system consists of three main parts: a YANG push collector, an Apache Kafka message broker for real-time data sharing, and a schema registry that helps producers and consumers understand the structure and version of each message. This setup not only allows new YANG models to be registered with the schema registry but it also enables the monitoring system to gather metrics, allowing self-service onboarding and closed-loop (closed loop is one of the compulsory steps to realize the vision of the autonomous networks). The thesis provides a detailed explanation of how the data collector and schema registry are connected, achieved through the development a library named "libyangpush."

CONTENTS

1 INTRODUCTION

The goal of this project is to realize a network monitoring system that gives generic support to YANG[1] and YANG push[4][5] where new model and metrics can be quickly onboarded to a Data Mesh environment. In the current telemetry systems that relies on Kafka, there are two main problems:

- **missing semantic**
- **need human intervention, non closed loop**

Missing semantic occurs when we are using Kafka to transmit YANG data, and it is inevitable because all operator use Kafka. Without YANG schema, message transmitted to Kafka consumer will be unable to be interpreted. This becomes even worse when two device populate the data under the model with the same name but with different types, which can make the data very confusing. If we try to directly use this approach to monitor a big network, it can make analysing data a tough job.

With the increasing usage of YANG in devices, new models are introduced regularly. However, if the existing database is based on outdated configurations, adding these new models can lead to a disorganized database. Otherwise, a significant amount of manual effort is required to ensure that all configurations in the process are consistent. Nevertheless, this manual approach does not align with the goal of achieving full automation in the system.

In order to realize data fast onboard and maintain semantic in the whole pipeline, Apache Kafka[3] is used in the project. Apache Kafka is a streaming platform that gathers and distribute data as stream using with pub/sub mechanism. It is scalable and has high throughput, which meet the requirements of fast onboard of new model and metrics in big networks. To help with maintaining the semantic of YANG model, schema registry is used to preserve the YANG schema. The Kafka serialization process transforms YANG telemetry data and the schema id into byte to facilitate transmission. After data has been ingested into Times Series Database(TSDB)[7] from Kafka, the YANG schema from schema registry is used to define times series database schema and the ingestion rule for operational metrics. The diagram of the project is shown in Fig.1.

As part of the collector process, a YANG push receiver program needs to be extended to receive and parse real-time YANG push message; get the subscribed YANG model and its dependencies and register them into schema registry. Since April 2020, Confluent(Kafka Provider Company) extended their schema registry to be pluggable, which enables user to define custom schema type and data serialization. The schema registry needs to be extended to support YANG. The YANG data(payload of the NETCONF notification message) needs to be serialized with schema ID of the corresponding YANG schema.

This thesis focus on extending the YANG push receiver, which is integrated as an open source library ***libyangpush***, and is going to be used as an internal library of network monitoring software pmacct[8].

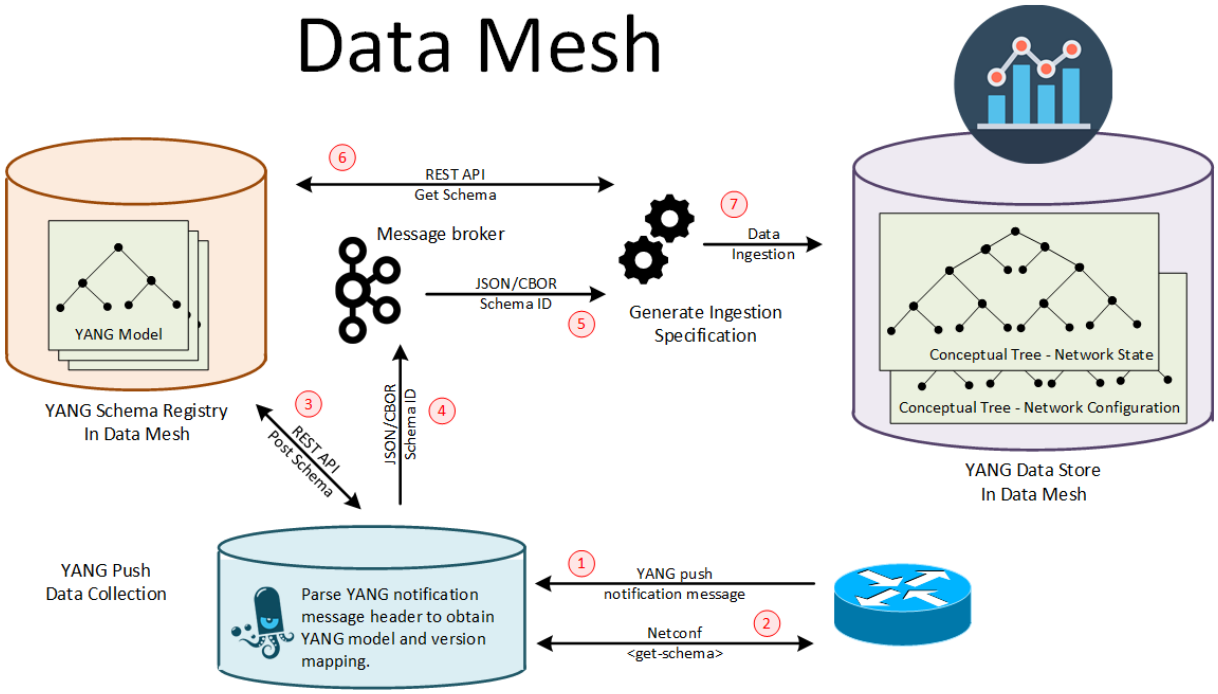

Figure 1: Diagram of the data mesh for YANG push

2
BACKGROUND AND RELATED WORK

2.1 YANG Data Modelling Language

According to definition in RFC6020[9], **"YANG is a data modelling language used to model configuration and state data manipulated by the network Network Configuration Protocol(NETCONF), NETCONF remote procudure calls, and NETCONF notifications"**. YANG models are designed to be human-readable and machine-processable, making it easier to create, modify, and manage network configurations. It defines a hierarchical data structure that can be used for operations.

YANG model has four types of dependency statements, and each is used to augment the definition of the original model in different ways:

- **import** The "import" statement makes definitions from one module available inside another module or submodule.
- **include** The "include" statement is used to make content from a submodule available to that submodule's parent module, or to another submodule of that parent module.
- **augment** The "augment" statement allows a module or submodule to add to the schema tree defined in an external module, or the current module and its submodules, and to add to the nodes from a grouping in a "uses" statement.
- **deviate** The "deviate" statement defines how the device's implementation of the target node deviates from its original definition.

The definition of YANG has some similarities with the MIB(Management Information Base)[10] of SNMP(Simple Network Management Protocol)[11]. MIB is a text file that describes the attributes and values of SNMP objects. An SNMP object is a variable that represents some aspects of a network device, such as its status, configuration, or performance. From the perspective of YANG, the MIB can also be understood as the YANG model. SNMP defines several commands to fetch data from MIB, while for YANG the equivalent command is defined by the NETCONF operations.

MIB and YANG have some key differences. For instance, MIB utilizes ASN.1[12] which is more compact and efficient, while XML[13] used in YANG is more human-readable and widely supported. MIB is more compatible and interoperable, but is not as expressive or adaptable. It is widely supported by SNMP agents and managers, but has some limitations when defining complex data types, constrains, and operations. On the other hand, YANG is more flexible and powerful, but less consistent and flexible. It can define more complex data models with

richer semantics, validation, and manipulation. Given the outlined comparison, the decision is made in favor of YANG.

2.2 YANG push subscription mechanism

YANG push is a new IETF standard that allows a client to monitor changes to a YANG datastore with notifications instead of with <get> RPC request. It is based on the subscription mechanism, and it enables the client to specify the information it wants to collect and the frequency at which the information should be delivered. It is generally used to monitor configuration or operational datastore changes.

The NETCONF working group has delivered a set of RFCs that can be collectively referred to as "YANG push". The most essential ones are the following:

- Subscription to YANG Notifications (**RFC8639**[4])
- Dynamic Subscription to YANG Events and Datastores over NETCONF(**RFC8640**[14])
- Subscription to YANG Notifications for Datastore Updates(**RFC8641**[5])

From the YANG push specification[5], there are two kinds of subscriptions:

1 **Event Stream Subscriptions**: Existing subscriptions to NETCONF event stream like <netconf-capability-change>.

2 **Datastore Subscriptions**: Subscription to a target in the datastore. Server generates a datastore update that is only sent to specific subscription.

There are two kinds of subscriptions under the Datastore Subscription:

1 **Periodic Subscription**: Data is sent to the client every interval, even if the data content did not change

2 **On-Change Subscription**: The data is sent to the client only when the data changes.

Nowadays for vendors, another approach called gRPC[15] is also widely used to stream YANG data. Similar with many RPC system, gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. gRPC uses Protocol Buffers[16] for serializing structured data. Protocol Buffer compiler generated source code for accessing the structured data. The disadvantages for gRPC is that for each YANG service model, it needs a .proto file for serialization, which introduced a lot of manual work when onboarding new YANG models.

2.3 Kafka and Schema Registry

Kafka is a distributed streaming platform that is often used as a foundation for building real-time data pipelines and event-driven architectures. In the context of Data Mesh, Kafka can play a critical role in enabling domain teams to own and manage their own data.

During the transmission of messages in Kafka, the client and server agree on the use of a common syntactic format. Kafka brings default converters(such as String and Long) but also supports custom serializers for specific use cases. Serialization is the process of converting objects into bytes. Deserialization is the inverse process - converting a stream of bytes into an object. See Fig.2 for the workflow for Kafka serializer and deserializer.

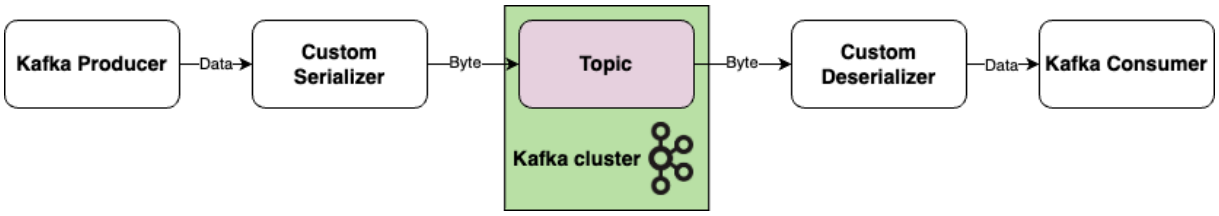

Figure 2: Workflow for Kafka Serializer and Deserializer

The Kafka producers and consumers do not communicate with each other directly, but rather information transfer happens via Kafka topic. However, the consumer still needs to know the type of data the producer is sending in order to deserialize it.

With the schema registry in place, before sending the data to Kafka, the producer talks to the schema registry first and checks if the schema for that data is available. If it does not find the schema, it then creates and registers the schema for the data in the schema registry. Once the producer gets the schema, it will serialize the data with the schema and send it to Kafka in binary format prepended with a unique schema ID. When the consumer processes this message, it will communicate with the schema registry using the schema ID it got from the producer and deserialize it using the same schema. If there is a schema mismatch, the schema registry will throw an error letting the producer know that it's breaking the schema agreement. See Fig.3 for the workflow of Kafka schema registry.

2.3.1 • Schema Evolution

YANG model can be augmented or updated in the device. These changes need to be reflected into the schema registry, and this process is called the schema evolution. Schema Evolution is to update the schema with the introduction of new fields or update of existing fields. When schema evolution happens, it is critical for new downstream consumers to be able to handle data encoded with both the old and the new schema seamlessly. There are two important types of compatibility.

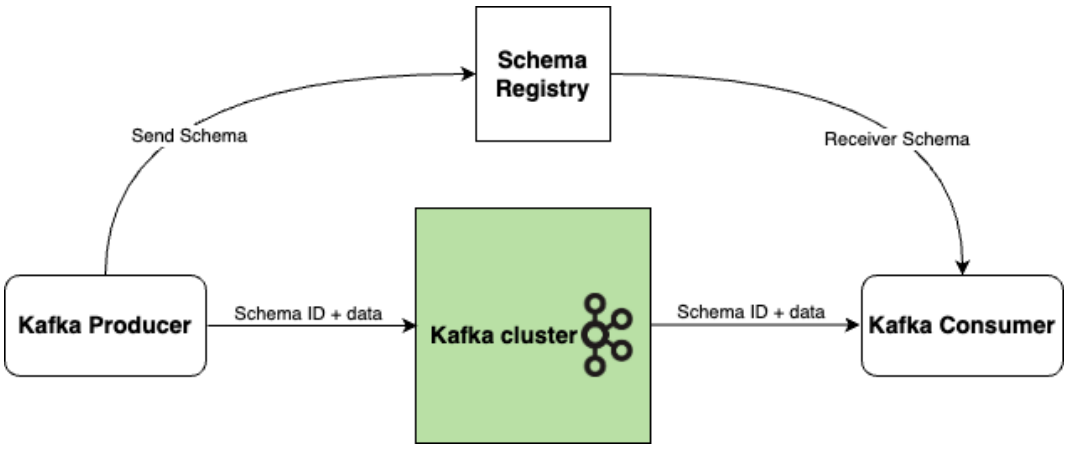

Figure 3: Workflow for Kafka Schema Registry

- **Forward Compatibility** a design that the schema is compatible with future versions of itself, which means that the consumers using the new schema can read data produced with the last schema.
- **Backward Compatibility** a design that the schema is compatible with previous versions of itself, which means that data produced with a new schema can be read by consumers using the last schema.

The default compatibility type of Confluent Schema Registry is Backward Compatibility. YANG schema registry also needs to respect the backward compatibility. The tool used for compatibility check in YANG schema registry is the YANG comparator[17]. YANG comparator is a tool which can compare two versions of YANG releases. It help users to identify the differences of the two version and returns the result of the specific type of compatibility check. The functionalities of YANG comparator, YANG validator have been integrated into the currently using YANG schema registrys.

2.4 Data mesh and closed loop operation

Data mesh is an architectural framework for managing analytical data through distributed, decentralized ownership. Organization used to invest in central data lake with a data team to drive business based on data. This approach centralized all data tasks to the central data team which cannot answer all analytical questions. With data mesh, data is handled by central data team and distributed to other team in the right domain for analysis. The core for data mesh is **Domain-oriented decentralization for analytical data**.

Closed loop operation automatically regulates a system to maintain a desired state. In the networks that operator are managing today, the device and network topology is always changing

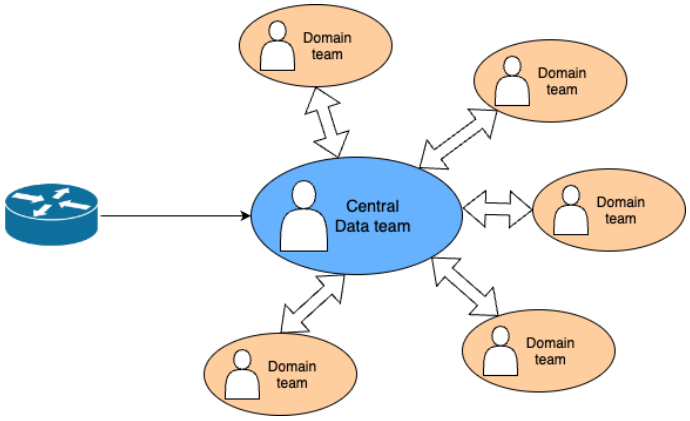

Figure 4: Principle of the data mesh

or upgrading, which will lead to a series of changes in the collector database. It is hard to manually synchronize each component in the pipeline whenever there is a change. Therefore, a closed loop operation that can real-time capture changes in the device and network, reflect it to database and make corresponding changes is essential for realizing Network Autonomous.

In the context of Network Monitoring, the purpose of applying Data Mesh is to deliver the data as a product. The operator aims to perform accurate anomaly detection/prediction, because of which reliable data collection is required. In that scenarios, Data Mesh is applied to deliver the data to data analyst with losing any semantics. Closed loop operation is applied to collect real-time data and make sure change happens in the pipeline would not crash the entire system. In general, the Data Mesh is used to decentralize the data and facilitate data analysis, while closed loop is used to realize autonomous data collection system.

2.5 Problem Statement

To implement the Data Mesh system, it is important to guarantee that each part natively support YANG and YANG push, especially the schema registry and data collector. The job of this thesis is to make data collector support parsing YANG push message and create YANG schema using ***libyangpush*** in preparation for the next step in the toolchain, the Kafka injection.

As a library that will be integrated and used in pmacct[8], the job of libyangpush can be simplified as to create schema and map it with the YANG push subscription. The schema of the subscription must describe the format and semantics of every possible message received through the subscription. In order to do that, we need to:

- Find which model(s) is/are directly involved
- Get the YANG source of the involved models and of the other related models needed to interpret messages of the subscriptions

For finding the directly involved models, we aim to provide a minimal and complete set of models. In that context, "minimal" means that all models provided are actually required to interpret at least on message of the subscription and "complete" means that the set of models can be used to interpret any message of the subscription. For getting the YANG source code of the involved models and other models needed to interpret the contents of the subscription, we consider mainly the efficiency and feasibility.

However, the difficulty is that the relationship between YANG models are complicated. To create the schema for one subscription, we need to include all directly involved models as well as their dependencies. Each model can have four kinds of dependencies: import, include, augment and deviate. The first two kinds of dependencies are directly specified in the model content, while the last two kinds are reverse dependencies, which means that only the model that defines the augmentation or the deviation knows about this relationship. Therefore, we cannot obtain the deviations or augmentations of a given YANG model by parsing its code. We need more information about the set of YANG models defined on the device to find these reverse dependencies.

Section 4 describes different possible approaches we could apply to do these two jobs and makes a comparison among them. One solution is chosen and the workflow is given.

3
DEVELOP ENVIRONMENT AND RESOURCES

3.1 LIBNETCONF2[6]

libnetconf2[18] is a NETCONF library in C intended for building NETCONF clients and servers which handles NETCONF authentication and all NETCONF RPC communication.

The NETCONF protocol specification allows to use the protocol on top of several transport protocols. libnetconf2 provides support for SSH and TLS transport. For the SSH tranport[19] we use in the project, there are three types of authentication:

- RSA public-key and private-key authentication
- password authentication
- Interactive SSH authentication

After authentication passes and NETCONF handshake is successful, libnetconf2 needs to check and fill the libyang context(refers to Section 3.2). It firstly checks the list of capability to see if get-schema and some functional YANG model(ietf-yang-library, ietf-netconf-monitoring etc.)are supported or not.

For the YANG model ietf-netconf-monitoring defined in RFC6022[20], the container netconf-state:schemas contains the list of data model schemas supported by the server. libnetconf2 parses the list of schema and send <get-schema> request to the router to request for all YANG models. The models will be either store in cache(deleted after the NETCONF session shuts down), or in the disk(so that libnetconf2 do not do get-all-schema in the next connection)

The get-all-schema is a default action of libnetconf2. The advantages for this action is that: it enables the client to know the entire YANG model relationship, including the reverse relationship(augmentation and deviation) and once all models are stored in disk, libnetconf2 do not do get-all-schema in the next connection unless the schema is not found in the search path. The disadvantage is that it does not facilitate the usage of libnetconf2 with real device. The reason is that for real device, there is usually a large amount of YANG models. The process of get-all-schema could take around 20 minutes. It is not acceptable for production since sending one NETCONF request would take 20 minutes when it is not a standard action, but rather a design choice.

libnetconf2 is used by libyangpush to implement the standard NETCONF operations. libyangpush also makes use of the default get-all-schemas of libnetconf2 to get the YANG model dependencies.

3.2 LIBYANG[6]

libyang[21] is a C library implementing the procession of the YANG schemas and data which are modelled by the YANG language. An important concept for libyang is the context, which allows callers to work in environments with different sets of YANG models. A context contains a set of successfully parsed models. The model contents and relationship will be fully recorded in the context. The relationship between libyang and libnetconf2 is that each NETCONF session works with one context. The models fetched from device will be stored in context. libyang's features include:

- **Parsing and validating schema in YANG/YIN format** To be able to work with YANG data instances, libyang has to represent YANG data models. All the processed models are stored in libyang context and loaded using parser functions.
- **Parsing, validating and printing instance data in XML/JSON** libyang's ability to parse the NETCONF XML/JSON message
- **Manipulation with the instance data** libyang's ability to create or modify an existing data tree.
- **Support for default values in the instance data(RFC6243)**
- **Support for YANG extensions and user types**
- **Support for YANG Metadata (RFC7952)**
- **Support for YANG Schema Mount (RFC8528)**

libyangpush uses the concept of context in libyang to store all schemas of a device, with which the dependency relationship can be directly parsed. The YANG tree structure defined in libyang is also used for parsing the YANG push message.

3.3 NETOPEER2

With NETCONF and YANG modelling becoming the de-facto standard for remote configuration and management of network infrastructure and devices, their implementation varies from vendors. Netopeer2[22] and Sysrepo[23] provide a fully open source and standards compliant implementation of a NETCONF server and YANG configuration datastore.

Netopeer2-server is a full-fledged NETCONF server. It configures itself based on ietf-netconf-server YANG model data in sysrepo[23]. It uses ietf-netconf-acm access control of sysrepo.

Netopeer2-client serves as a generic NETCONF client providing a simple interactive command line interface. It allows user to establish a NETCONF session with a NETCONF-enabled device on the network and to obtain and manipulate its configuration data.

Netopeers2 is used to simulate the device by populating standard YANG push notification message and NETCONF reply(to <get>, <get-schema> requests etc.).

3.4 Development Environment

Due to limitation in the lab, that YANG push is not yet fully supported in the device, the router is simulated using netopeer2[22]. Netopeer2 uses sysrepo to simulated the YANG datastore of router and libnetconf2 to implement standard NETCONF actions and YANG push.

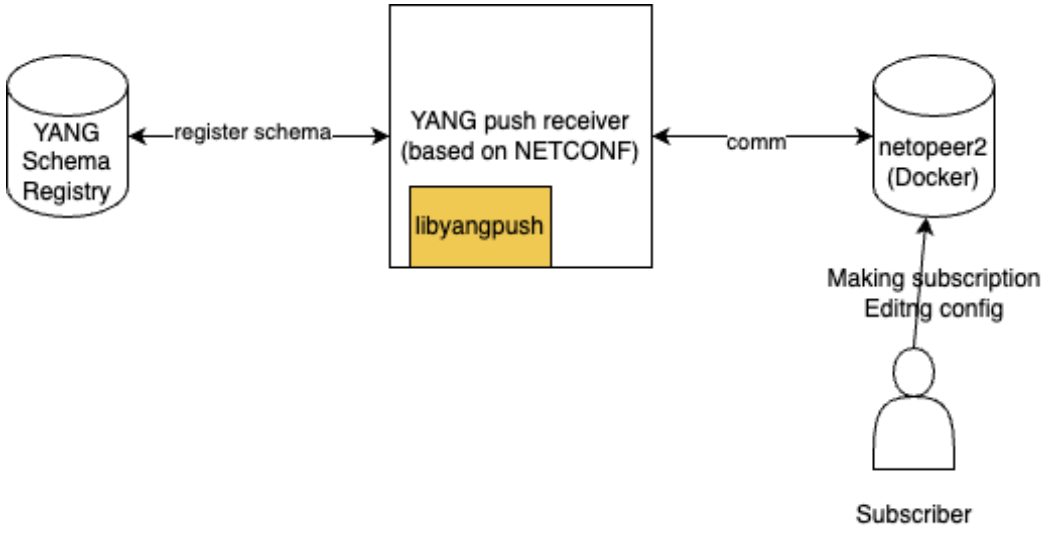

Figure 5: Development Environment

A third party subscriber is making subscription to netopeer2 and modifying its datastore by sendig <edit-config>. The NETCONF client has established two types of subscription with the netopeer2:

- subscription to *ietf-subscribed-notification*
- subscription to other model interested

The first subscription is used to monitor the status of subscription. When there is a new subscription established, this subscription to *ietf-subscribed-notification* enables the client to receive a push-change-update and therefore create and register schema for it. The subscribed data is received from the actual subscription to the interested model. The client should dump it to Kafka along with the schema ID for it.

4 DESIGN

This chapter discuss and compare the possible solutions for finding model and creating schema. The workflow and details of the chosen implementation choice is intorduced at the end.

4.1 Identifying the models involved in a subscription

We have investigated three methods to identify the involved models, they are receptively:

- Subscribe to ietf-subscribe-notification. Get the subscription information by parsing the push-update/push-change-update
- Obtain the subscription id in the YANG push message. Send <get> request to request for the same subscription information
- Parse the prefix occurs in first message we receive from one YANG push subscription.

4.1.1 • ietf-subscribed-notification push-update

When creating a subscription to a certain Xpath, the information of subscription will be interpreted using ***ietf-subscribed-notification***. According to RFC8641[5], by configuring YANG-Push to xpath ***/ietf-subscribed-notifications:****, we could receive the datastore-contents of it in the push-update.

The field **"datastore-xpath-filter"** of push-update stores the Xpath that we are subscribing to(refer to the red line in Fig.6). By parsing the Xpath of each subscription, we know the model that we are subscribing to. In the example XML message in Fig.6, the element value for **"datastore-xpath-filter"** is ***"/sn:subscriptions"***. The prefix "sn" stands for the namespace of ***ietf-subscribed-notifications*** which is specified in the ***xmlns:sn*** attribute of the enclosing tag. In this way, we know that the node we are subscribing to is the container **subscriptions** of model ***ietf-subscribed-notifications***.

Similarly, for field **"datastore-subtree-filter"**, we get the subtree filter of one subscription. The subtree corresponding to the Xpath mentioned above is **"<subscriptions xmlns="urn:ietf:params:xml:ns:yang:ietf-subscribed-notifications"/>"** . In this subtree filter, the model is specified in the namespace **"urn:ietf:params:xml:ns:yang:ietf-subscribed-notifications"**. The subscribed container for this model is the name **"subscription"**. Therefore, we get the same result as that for Xpath.

By configuring on-change yang push with selection filter as ***/ietf-subscribed-notifications:subscriptions***, we receive updates whenever changes to the datastore content in the selection filter occurs. According to RFC8641[5], a "push-update" will be sent if there is a need to synchronize the receiver at the start of a new subscription. We can use this "push-update" to find out YANG model for already established subscriptions which will not be received at the "push-change-update" later. For the "push-change-update" message, by parsing the YANG patch record in the **datastore-contents**, we can learn what has been modified, newly added or deleted in terms of subscriptions in **ietf-subscribed-notifications** datastore. And then we can change the schema according to the subscription changes we receive.

For periodic YANG push, the method will be more complicated because we receive "push-update" every time, and the "push-update" always contains the whole content of the datastore of the selection filter. We will have to check every time if one subscription has been processed or not, or if one subscription still existed, or if new subscriptions are added. Therefore, it is not recommended to use periodic yang push for receiving subscription update.

```
<datastore-contents>
   <subscriptions xmlns="urn:ietf:params:xml:ns:yang:ietf-subscribed-notifications">
    <subscription>
     <id>2222</id>
     <datastore xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push"
      xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">ds:operational</datastore>
     <datastore-xpath-filter xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push"
      xmlns:sn="urn:ietf:params:xml:ns:yang:ietf-subscribed-notifications">/sn:subscriptions</datastore-xpath-filter>
     <revision xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push-revision">2019-09-09</revision>
     <transport xmlns:unt="urn:ietf:params:xml:ns:yang:ietf-udp-notif-transport">unt:udp-notif</transport>
     <encoding>encode-xml</encoding>
     <configured-subscription-state>valid</configured-subscription-state>
     <receivers>
      <receiver>
       <name>subscription-specific-receiver-def</name>
       <state>active</state>
       <receiver-instance-ref xmlns="urn:ietf:params:xml:ns:yang:ietf-subscribed-notif-receivers">global-udp-notif-
receiver-def</receiver-instance-ref>
      </receiver>
     </receivers>
     <periodic xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push">
      <period>30000</period>
     </periodic>
    </subscription>
</datastore-contents>
```

Figure 6: An example of *ietf-subscribed-notification* push-update in XML

4.1.2 • Obtain subscription id from yang push message

According to the yang tree structure of **ietf-yang-push** notification(push-update or push-change-update), the subscription-id will always be specified. As a result, except for configuring

yang push to **ietf-subscribed-notifications** for receiving subscription update (refer to Section 4.1.1), we can make use of the subscription-id that appears in every yang push telemetry message to request for the subscription information.

With the subscription-id, we perform a get request to this subscription-id in container **subscriptions** of model **ietf-subscribed-notifications**. The content of reply we receive for this get request is exactly the information for subscription[subscription-id]. With the subscription information, we do the same as in Section 4.1.1 to parse the model name.

The difference between 4.1.1 and 4.1.2 is that in 4.1.2 we do not need to configure yang push to **ietf-subscribed-notifications:subscriptions**. Instead, we make use of the subscription-id in every yang push message to request for subscription information. However, this means that we will have to parse every message header. And we will also have to check if this subscription has been processed or not by looking up in the mapping of subscription-id and schema-id, while in Section 1.1 it is the device that detects changes.

4.1.3 • Parse the first message to get namespaces

According to RFC8641[5], the **datastore-content** of push-update notification "constitutes a snapshot at the time of update of the set of data that has been subscribed to. The snapshot corresponds to the same snapshot that would be returned in a corresponding 'get' operation with the same selection filter parameters applied.". Normally, the get request will return all fields that have been specified in the selection filter. We could consider that what we receive in the datastore content is everything related to this subscription. Therefore, we could find out all involved models by looking for namespaces in the datastore-content in the first message we received for one YANG push subscription. In this way we even achieve a higher level of minimizing the model set.

In the example XML message in Fig.7, by looking for namespaces, we learn that the models that is subscribed to is **ietf-interfaces**. As an augmented model, we also find **ietf-ip**. Therefore, the involved models are **ietf-interfaces** and its imports and includes, and **ietf-ip**.

However, we might have some cases where a message can never have all augmenting models at the same time (for instance augment based on a choice). In that case, we would need to validate each message and restart the process to extend schema if a new message is not accepted.

4.2 Obtaining the involved models and their dependencies

```
"datastore-contents": {
    "ietf-interfaces:interfaces": [
     {
      "interface": {
       "name": "eth0",
       "type": "iana-if-type:ethernetCsmacd",
       "oper-status": "up",
       "speed": "1000000"
       "ietf-ip:ipv4": {
        "enabled": true,
        "forwarding": true
       }
      }
     }
    ]
}
```


Figure 7: An example of datastore-content in a notif message of subscription to ietf-interfaces

4.2.1 • On-demand downloading

The idea of on-demand downloading is to send get-schema request to get the YANG models based on the model name we obtain. For the main model in the subscription, we can obtain its name using method in Section 4.1. However, we need to include the dependencies of this model in its schema to maintain the completeness of model set. Especially for the reverse dependencies (deviate and augment), since they are modifying the schema of the main model but specified in the model that defines the augmentation or the deviation, special mechanism is required to obtain them.

First of all, for import and include models, since they are always specified in the YANG model content (see Fig.8), we can use libyang to parse the yang code to get their names. For each of the model name, we send a get-schema to device to request for the model.

Secondly, for deviate models, the **"list deviation"** is defined in RFC8525 in model **ietf-yang-library** in container **models-state** (see Fig.9). According to description, it is "The list of YANG deviation model names and revisions used by the server to modify the conformance of the models associated with this entry." We could do a NETCONF get to **/ietf-yang-library:models-state** at the connection setup and stores the deviations of each model in cache. Later when we obtain the main model name, we could use it to do a look up in the deviations to find the deviate model, and then send another get-schema request to get the deviate model.

Thirdly, it is not clear yet how to get the augment models. Especially if we use method 4.1.1 and 4.1.2, there is no way we can get the information about augmentation of main model when it is not specified in the model content nor in the content of **/ietf-yang-library:models-state**. However, if we use method 1.3, there is a chance that we could capture the augmentation according to the namespace appears in the datastore-contents, but since augmentation might

```
module example-system {
    yang-version 1.1;
    namespace "urn:example:system";
    prefix "sys";
    import ietf-yang-types {
     prefix "yang";
     reference "RFC 6991: Common YANG Data Types";
    }
    include example-types;
    organization "Example Inc.";
    contact
     "Joe L. User
      Example Inc.
      42 Anywhere Drive
      Nowhere, CA 95134
      USA
      Phone: +1 800 555 0100
      Email: joe@example.com";
    description
     "The module for entities implementing the Example system.";
    revision 2007-06-09 {
     description "Initial revision.";
    }
    // definitions follow...
  }
```

Figure 8: An example of include/import defined in RFC7950

not be used in the first message, we are risking of missing dependency for upcoming message. Even though we could validate each message and restart the process to extend schema later, it will be more complicated and we might need to register a schema more than once, which might introduce some unknown errors.

Last but not least, for each model we obtained through get-schema, we need to use libyang to parse the model content and check if it has dependencies. Therefore, the get-schema is a recursive process and needs to be repeated until we reach the model that has no dependency.

4.2.2 • Get-all-schemas

The main idea of get-all-schemas is to get all models, store them in the disk, and analyze their full dependencies (import, include, deviate and augment) at the beginning of connection. In this way, later when we need a model, we do not need to send another get-schema request. Instead, we could directly get them from disk by the model name, and we could also avoid the risk of missing augment dependencies.

To implement this, firstly we need to send a get request to ***ietf-netconf-monitoring:netconf-state***. For each of the schema in the schema list returned in the reply, we send a get-schema request. Since the schema list is composed of all YANG schemas supported in the device, it is guaranteed that we can get all models and their dependencies. The models obtained from

```
x--ro modules-state
  x--ro module-set-id    string
  x--ro module* [name revision]
    x--ro name             yang:yang-identifier
    x--ro revision         union
    +--ro schema?            inet:uri
    x--ro namespace          inet:uri
    x--ro feature*         yang:yang-identifier
    x--ro deviation* [name revision]
    |  x--ro name       yang:yang-identifier
    |  x--ro revision   union
    x--ro conformance-type    enumeration
    x--ro submodule* [name revision]
      x--ro name       yang:yang-identifier
      x--ro revision   union
      +--ro schema?    inet:uri
```

Figure 9: models-state in the yang tree of ietf-yang-library

get-schema reply will be store in disk. Then we need to use libyang to analyze all models' content and store their relationships in NETCONF session's context. In this way, we have the full relationship at the connection establishment. The above is default behavior when using libnetconf2 client connects to a NETCONF server.

With all models store in disk, we can use the simplest way to get the name of subscribed model. It could be either 4.1.1 or 4.1.2, depends on whether we configure yang push on ***ietf-subscribed-notification:subscriptions*** or not. With the subscribed model name, we can get its yang source and its dependencies directly from disk.

This method has several disadvantages. First of all, it takes several minutes to get all models. Moreover, when device disconnects or system reboots, its YANG models could change. When the connection and subscription resume it is no longer reliable to use the previous model set. Therefore, we need to have a mechanism to detect these events and invalidate the model set, and then do a get-all-schemas again for this device.

However, it could be easier to implement this method for obtaining models since libnetconf2 already support get-all-schema and analyzing all models' relation at the NETCONF connection establishment by default. If we want to implement Section 4.2.1, we have to talk to libnetconf2's author and ask for a config switch that could turn off get-all-schemas behavior.

4.3 Chosen Implementation

After comparing the pros and cons of each solution, we decide to chose to parse the **ietf-subscribed-notification** subscription to identify the models, and use get-all-schema to obtain the involved models and dependencies. That is because currently we focus on having a working prototype. To successfully get the involved model and parse the full dependency of this model is sufficient for a prototype demonstration. In terms of minimizing the module set or to obtain the model as it is being updated in the device, is more an optimization job.

However, another important part of libyangpush is the interaction with the schema registry. The goal is to provide a list that respect that schema registration rule of schema registry, that to register a schema, its references must be fully registered. To do that, we have use DFS to design an algorithm the traverse the YANG model dependency and to produce the list(refers to Section 4.3.1).

4.3.1 • Algorithm for parsing dependencies

Four types of dependencies exist for YANG models: *import*, *include*, *augment*, and *deviate*. When enlisting these dependencies in the schema registry, a specific sequence needs to be followed: the base model must be registered before the top level model. Achieving this can be facilitated by utilizing the tree structure of YANG model and the context capabilities of libyang(refers to Section 3.2). Within this context, the model structures establish pointers directed towards their respective dependencies.

In this scenario, the DFS(Depth-first search) is ideal for traversing the YANG tree. It enables us to search for dependency recursively and it can makes sure that all dependency models have been traversed. However, due to the semantic definition of YANG, not all four kinds of dependencies need to be traversed in each node.

According to study, only the import and include dependency need to be searched when it comes to the recursive find dependency process. For augment and deviate of a child model, they are no longer dependency of the main model. The reason is the following:

Taking the augment of augment as an example: **The augment of augment model will either be the direct augment of the parent model, or not be the dependency of the parent model**. According to section-7.17 in RFC7950[1] the augment must points to a schema node identifier. If the target augment node is a container, list, case, input, output or notification node, the "container", "leaf", "list", "leaf-list", "uses" and "choice" statements can be used within the "augment" statement. However, the augment statement does not define an absolute schema node identifier, so cannot augment container, list etc. defined in another augment state. In another words, the augment of augment is illegal.

The conclusion is the same for deviate.

To modify a schema in an augment, the absolute schema node identifier is required, but neither augments nor grouping are schema node identifier, only, list, containers, leaf and leaf-ref are node identifier. And when we use the absolute schema node identifier to define augment, it becomes a direct augment of the main model as the first model prefix in the augment path will be the prefix of the parent model.

Taking a module set of 8 modules as an example, their relationship is the following:

- **a-module**: import b-module, include c-module, augmented by d-module, deviated by a-module-deviations
- **b-module**: import e-module
- **c-module**: import e-module
- **d-module**: augment a-module, import b-module
- **a-module-deviations**: deviate a-module, import str-type, include a-deviate-include
- **e-module**
- **str-type**
- **a-deviate-include**

We have extended DFS to design an algorithm with the following rules to traverse the YANG model dependency tree:

1 for the first module, check for all four dependencies(order by import, include, augment then deviate)

2 for the dependency of module, check their import and include

The tree structure we obtain is shown in Fig.10.

The order for registering produced by the algorithm is:

1 e-module

2 b-module(references to 1)

3 c-module(references to 1)

4 a-module(with reference 123)

5 d-module(references to 24)

6 str-type

7 a-deviate-include

8 a-module-deviate(references to 467)

9 a-module(references to 2358)

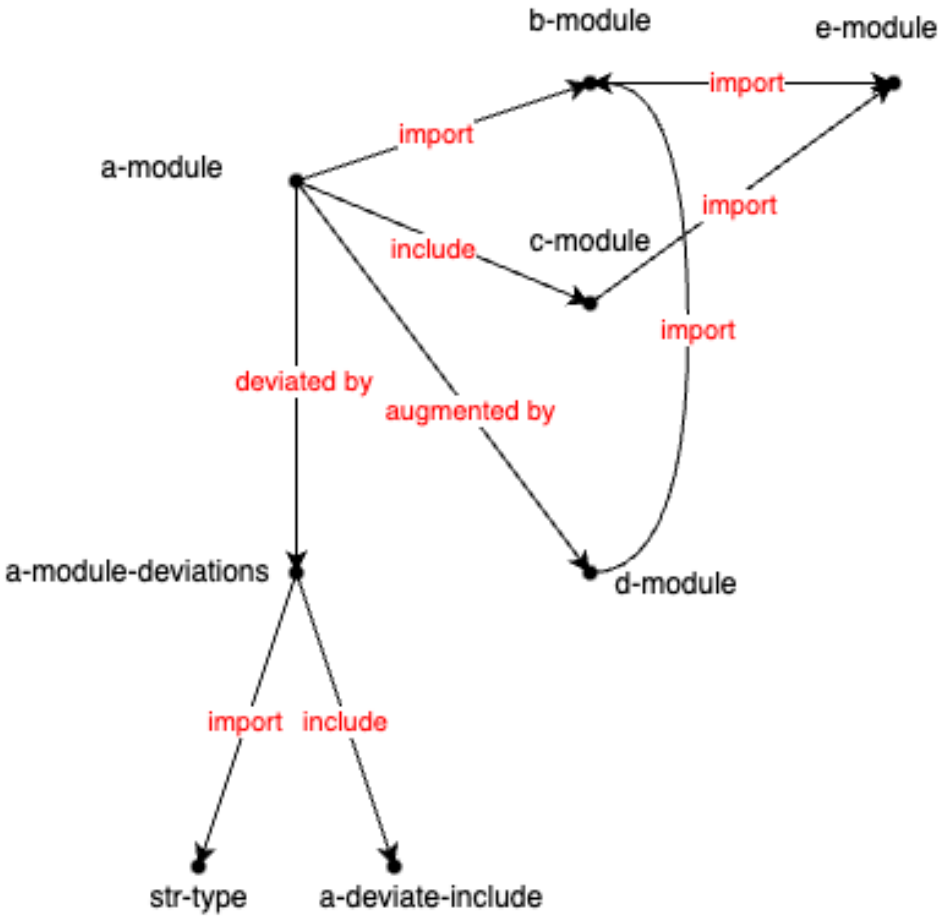

Figure 10: tree structure of module set

4.3.2 • WORKFLOW FOR GET-ALL-SCHEMAS

To sum up, the subscriber configures subscriptions to the router for it to send YANG push data to the collector(pmacct) to collect YANG data. The collector establishes subscription to ***ietf-subscribed-notification*** to get subscription update. They are respectively shown in step 5 and step 2. The process of create schema is continuous and has to be done when ever there is a push-change-update received.

The Fig.11 is the process for get-all-schemas. It is the basis for the later steps.

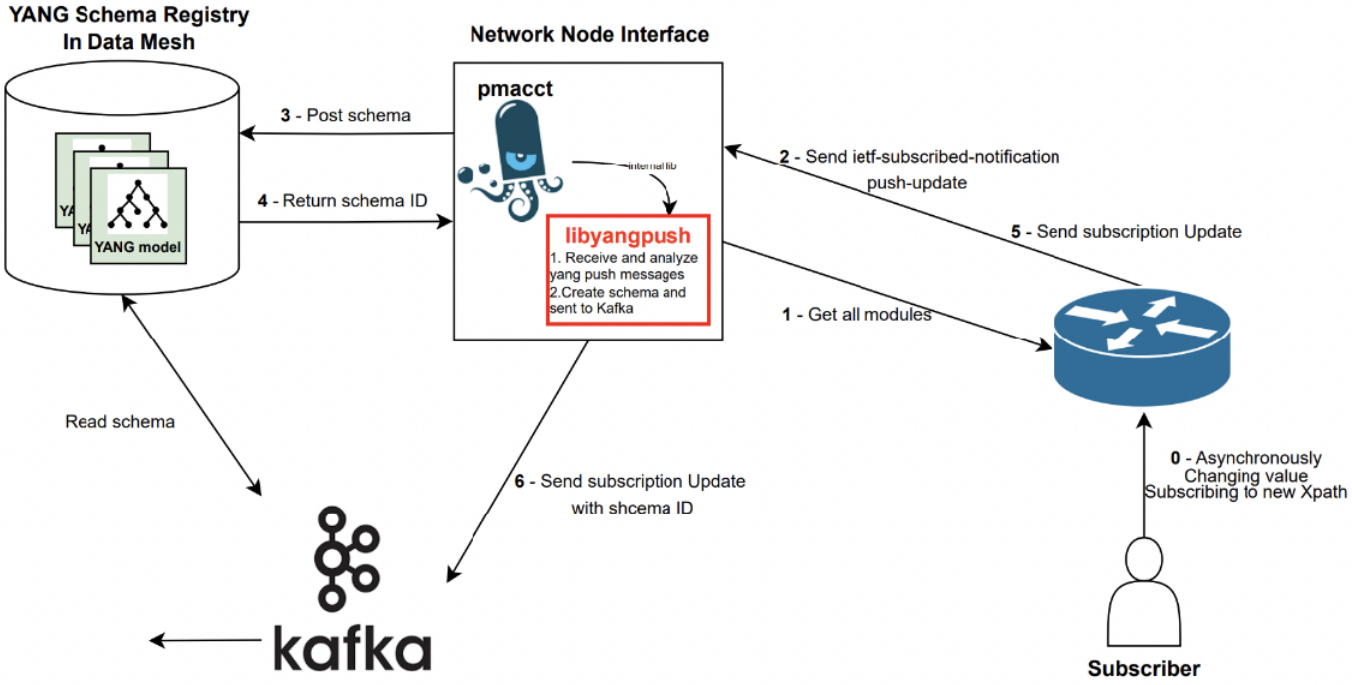

Figure 11: workflow for get-all-schemas

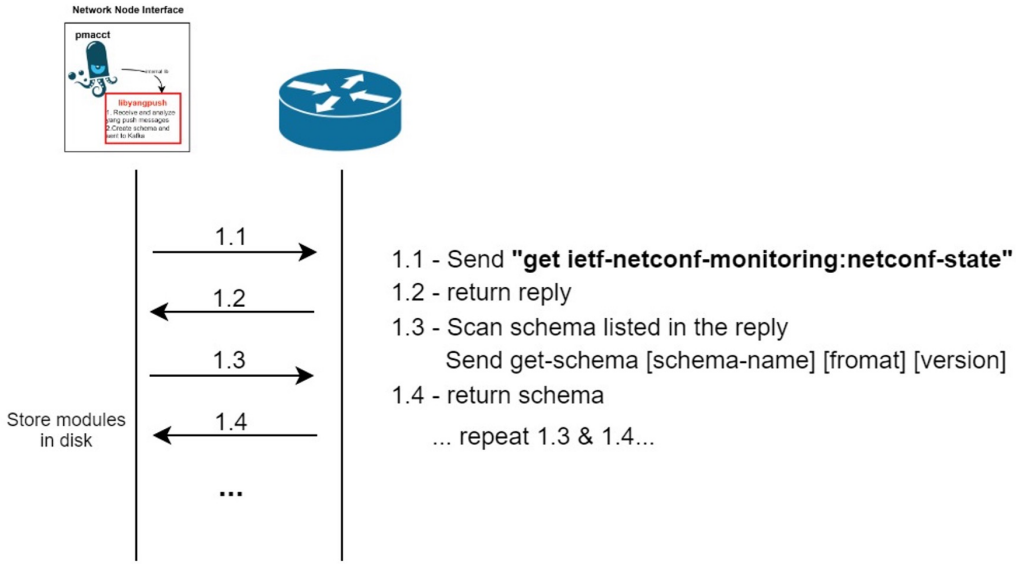

Figure 12: process for get-all-schemas

For parsing the **ietf-subscribed-notifications** push-update:

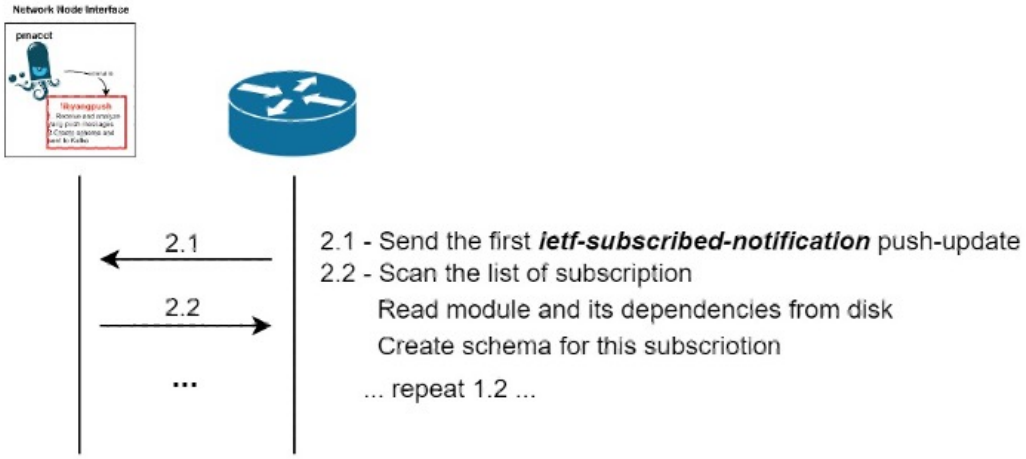

Figure 13: process for create schema

5 DEMO

The section introduced a demo that is run and tested with YANG schema registry. The schema registry is an instance running in Swisscom lab.

Overview

The goal of the demo is to test the functionality of libyangpush, check if it can parse the push-update message successfully, create schemas and register them in schema registry, and get back the schema id.

A module set contains three modules is used in the demo. In this module set, a-module is augmented by d-module, and d-module imports the e-module.

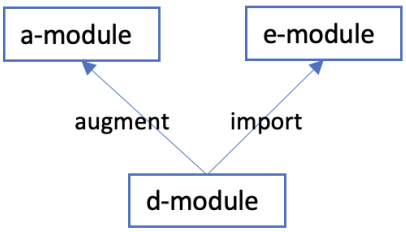

Figure 14: module set structure

The tree structure for a-module is shown in Fig.15.

```
module: a-module
 +--rw a
   +--rw a-instance* [name]
   |  +--rw name    string
   |  +--rw state?  string
   +--rw d:y
     +--rw d:y-leaf?  e:e-enum
```

Figure 15: a-module tree structure

The YANG model being subscribed to is the a-module. According to the module set structure, the order of the schema registration and their references should follows the list in Fig.16.

Receiving the YANG push Message

Due to the limitation of the mechanism of netopeer2, we can only use mock message at the moment. This is because the YANG push on-change notifications for operational datastore are

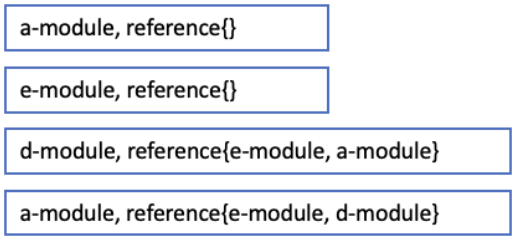

Figure 16: list of schema registration

limited to the push operational data changes, and according to what we described in Section 3.4, having a third party subscriber making subscription to the netopeer2 is not able to trigger on-change notification to the NETCONF client because this type of data is pull operational data changes and cannot be reported through on-change notification. Therefore, we use mock-up message to simulate and test the functionality of the library. The main interaction between client and netopeer2-server is the <get-schema> operation.

The message in Fig.17 is passed to libyangpush. It indicates that a new subscription with subscription id 6666 is created and that subscription targets the Xpath **/a-module:a**.

```
jean@jean-vm:~/code/libyangpush/build$ cat push-update.xml
<notification xmlns="urn:ietf:params:xml:ns:netconf:notification:1.0">
  <eventTime>2022-09-02T10:59:55.32Z</eventTime>
  <push-update xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push">
    <id>2222</id>
    <datastore-contents>
      <subscriptions xmlns="urn:ietf:params:xml:ns:yang:ietf-subscribed-notifications">
        <subscription>
          <id>6666</id>
          <datastore xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push"
            xmlns:ds="urn:ietf:params:xml:ns:yang:ietf-datastores">ds:operational</datastore>
          <datastore-xpath-filter xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push">
              /a-module:a
          </datastore-xpath-filter>
          <transport xmlns:unt="urn:ietf:params:xml:ns:yang:ietf-udp-notif-transport">unt:udp-notif</transport>
          <encoding>encode-xml</encoding>
          <receivers>
            <receiver>
              <name>subscription-specific-receiver-def</name>
              <receiver-instance-ref xmlns="urn:ietf:params:xml:ns:yang:ietf-subscribed-notif-receivers">global-udp-notif-receiver-def</receiver-instance-ref>
            </receiver>
          </receivers>
          <periodic xmlns="urn:ietf:params:xml:ns:yang:ietf-yang-push">
            <period>30000</period>
          </periodic>
        </subscription>
      </subscriptions>
    </datastore-contents>
  </push-update>
</notification>
```

Figure 17: Mock up push-update message

The folder **modules** is specified as the search path for libyang context. Before running the NETCONF client, the folder is empty. See Fig.18.

```
jean@jean-vm:~/code/libyangpush/build$ ls ../modules/
jean@jean-vm:~/code/libyangpush/build$
```

Figure 18: Search path folder content

After launching the NETCONF client, it fetches all modules from NETCONF server, parses the input messages and selects the corresponding YANG model needed to build the schema for the subscription. In Fig.19 is the first schema registered to the schema registry.

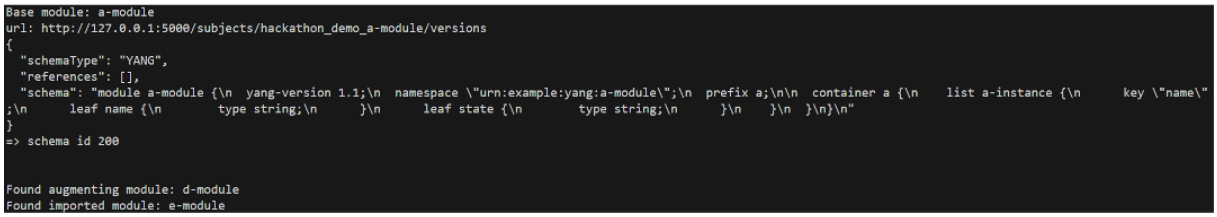

Figure 19: a-module schema with no reference

Then e-module and b-module will be registered with their references. See Fig.20.

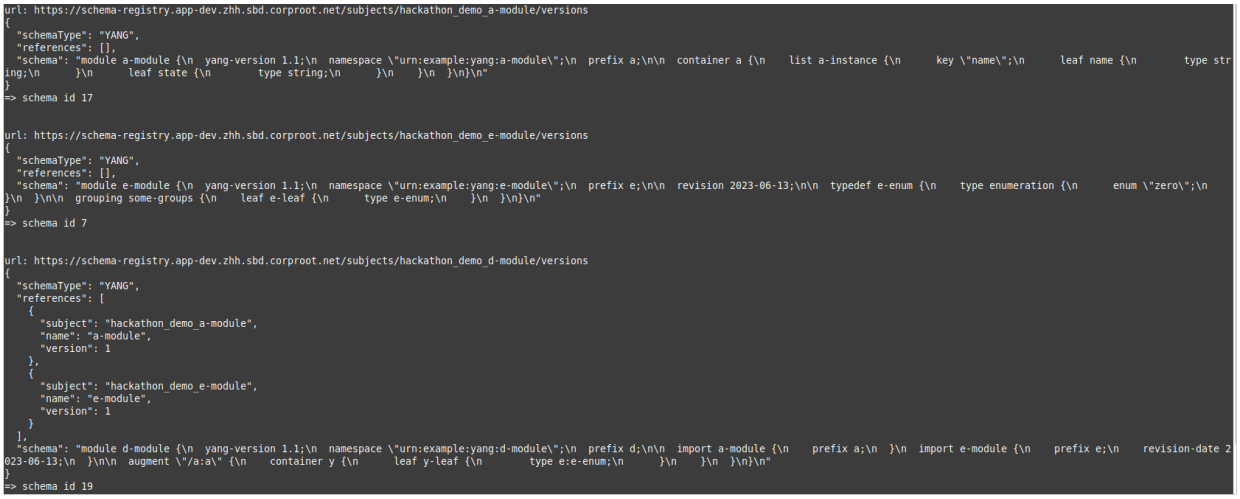

Figure 20: e-module, b-module's schema

Finally, a-module will be register again, with references to e-module and d-module.

Figure 21: Re-registration of a-module

6
SUMMARY AND FUTURE WORK

In this thesis we firstly discuss the reason why we need the integration of YANG push into Apache Kafka Data Mesh and how it is going to benefit nowadays Network Monitoring System. The operational metrics is crucial for performing accurate anomaly detection/prediction. Therefore we need YANG, the data modelling language modelling the configuration and operational data in the device to collect Management Plane data.

There are several methods we can chose to collect the Management Plane data, which brings the discussion of why using YANG push instead of other methods such as gRPC or SNMP. The advantage of using YANG and YANG push is that it adapts to the requirement of autonomous networks. It can automatically onboard on model and stream YANG data. Compared to gRPC, it reduces the human intervention by registering schema through collecting and parsing subscription information from **ietf-subscribed-notification** datastore update. Compared to SNMP, it is an active telemetry because the YANG push uses the pub/sub mechanism.

However, it is not feasible to directly apply YANG into the classical telemetry system. The data which is being sent through kafka will lose its semantics and make the job of data analysis difficult. Therefore we need the Data Mesh system proposed in this thesis to maintain the semantics in the entire data pipeline. This is achieved by adding a schema registry, which ensures that the YANG schema is accessible from both the consumer and producer by referencing with the schema id, and an extended YANG push collector, with the functionality integrated in the libyangpush to create schema for each YANG push subscription.

The most difficult part of this library is to parse the relationship and produce the registration list. There is the reverse dependency for YANG model, which is used and defined when the main model does not realize. If we do not find out those reverse dependency, there will be missing-dependency for the schema and it is not acceptable. To solve this problem, several solutions are proposed in Section 4. The get-all-schemas is selected for this project. The advantage for get-all-schemas is that it fetched all models from the beginning, and that enables us to parse all models relationship(even the reverse ones).

Except for parsing the relationship, it is also important to produce the registration list in the right order. The DFS algorithm is used to traverse through the YANG dependency tree. The node with the greater depth will be put into list first. In this way, we can make sure that the entire tree is contained in the list and the bottom level model will be registered first then the top level ones.

There is still some gaps to fill for libyangpush's functionality: it needs to create a YANG model for the notification message. Besides, it is necessary to have a debug session for the interaction between libyangpush with pmacct.

As for outcome of the thesis, the project demo has also been presented during the IETF117 Hackathon session and gained some approvals, and the library libyangpush is now a repository on github in the network-analytics organization and be public after being examined.

As for future work, the functionality of libyangpush will be put into production and deployed in Swisscom lab. And libyangpush might be delivered as a product of Huawei. There are several possible drafts that can be driven out of this project such as augmenting the **ietf-yang-library** to record the augmentation list to support the on-demand downloading.

REFERENCES

[1] M. Bjorklund, "Rfc 7950: The yang 1.1 data modeling language," Tech. Rep., 2016.

[2] I. A. Machado, C. Costa, and M. Y. Santos, "Data mesh: concepts and principles of a paradigm shift in data architectures," *Procedia Computer Science*, vol. 196, pp. 263–271, 2022.

[3] "Apache Kafka," 1 2011, [Online; accessed 2022-08-20].

[4] E. Voit, A. Clemm, A. G. Prieto, E. Nilsen-Nygaard, and A. Tripathy, "Rfc 8639: Subscription to yang notifications," Tech. Rep., 2019.

[5] A. Clemm and E. Voit, "Rfc 8641: Subscription to yang notifications for datastore updates," Tech. Rep., 2019.

[6] "Yang and netconf." [Online]. Available: https://www.liberouter.org/technologies/netconf/

[7] A. S. Foundation, "Druid | Database for modern analytics applications," 0, [Online; accessed 2022-08-20].

[8] "pmacct." [Online]. Available: http://www.pmacct.net/

[9] M. Bjorklund, "Rfc 6020: Yang-a data modeling language for the network configuration protocol (netconf)," Tech. Rep., 2010.

[10] R. Presuhn, "Rfc 3418: Management information base (mib) for the simple network management protocol (snmp)," Tech. Rep., 2002.

[11] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin, "Simple network management protocol (snmp)," Tech. Rep., 1989.

[12] G. Neufeld and S. Vuong, "An overview of asn. 1," *Computer Networks and ISDN Systems*, vol. 23, no. 5, pp. 393–415, 1992.

[13] M. Garcia-Martin and G. Camarillo, "Rfc 5364: Extensible markup language (xml) format extension for representing copy control attributes in resource lists," 2008.

[14] E. RFC 8640: Voit, A. Clemm, A. G. Prieto, E. Nilsen-Nygaard, and A. Tripathy, "Dynamic subscription to yang events and datastores over netconf," Tech. Rep., 2019.

[15] X. Wang, H. Zhao, and J. Zhu, "Grpc: A communication cooperation mechanism in distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 3, pp. 75–86, 1993.

[16] S. Stuart and R. Fernando, "Encoding rules and MIME type for Protocol Buffers," Internet Engineering Task Force, Internet-Draft draft-rfernando-protocol-buffers-00, Oct. 2012, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-rfernando-protocol-buffers/00/

[17] HuaweiDatacomm, "Huaweidatacomm/yang-transformer." [Online]. Available: https://github.com/HuaweiDatacomm/yang-transformer

[18] "Cesnet/libnetconf2: C netconf library." [Online]. Available: https://github.com/CESNET/libnetconf2

[19] T. Ylonen, "Rfc 4251: The secure shell (ssh) protocol architecture," 2006.

[20] M. Scott and M. Bjorklund, "Rfc 6022: Yang module for netconf monitoring," Tech. Rep., 2010.

[21] Cesnet, "Cesnet/libyang: Yang data modeling language library." [Online]. Available: https://github.com/CESNET/libyang

[22] "Cesnet/netopeer2: Netconf toolset." [Online]. Available: https://github.com/CESNET/netopeer2

[23] Sysrepo, "Sysrepo/sysrepo: Yang-based configuration and operational state data store for unix/linux applications." [Online]. Available: https://github.com/sysrepo/sysrepo