SEBA/NEM Monitoring and Logging infrastructure

This is a proposal to achieve consensus on a possible design for the NEM monitoring and logging infrastructure. It is intended to be presented at a SEBA call as soon as defined.

Monitoring and logging serves the following FCAPS goals:

- Faults → Plays a central role, delivering alarms and providing troubleshooting data
- Config → Likely plays only an indirect role, for example, as part of an auto-configuration feedback loop based on collected monitoring data.
- Accounting → Monitoring data is distinct from billing data, but may include usage-related data.
- Performance → Plays a central role, collecting and archiving performance stats.
- Security → Needs to have a security story (i.e., access control on who can see what data) but is not directly part of securing the system.

Most northbound OSS systems are likely to treat SEBA's as self-contained, and it's internal components as opaque. Because of this, it's of limited utility to provide monitoring data in a way that can only be understood with in-depth knowledge of internal SEBA components. Normalization and correlation of monitoring data must be done to align messages with the NEM models that the OSS already interacts with and understands.

It should be possible, at the OSS or troubleshooting level, to pick out a specific instance of a model and find all the relevant data relative to that model per the NEM service graph.

This infrastructure will be able to answer questions such as:

- If an OLT (or other shared component) is in a fault state, which subscribers are affected?
- For a specific subscriber, find all relevant accounting and performance data even if they've replaced their ONU due to a hardware failure.
- For a specific log message, which instance in the NEM data mode/service graph does it correspond to?
- Alarm triggers should be able to perform a match on simple criteria already in a message, instead of having complex logic or external lookups on every message to know if they should trigger.

In support of this goal, infrastructure will be put in place that will:

- Collect logs from all the components
- Collect usage statistics for all the components
- Collect events from all the component
- Support operator defined alarms

- Be modular, so that each component can be replaced by an operator to fit their needs

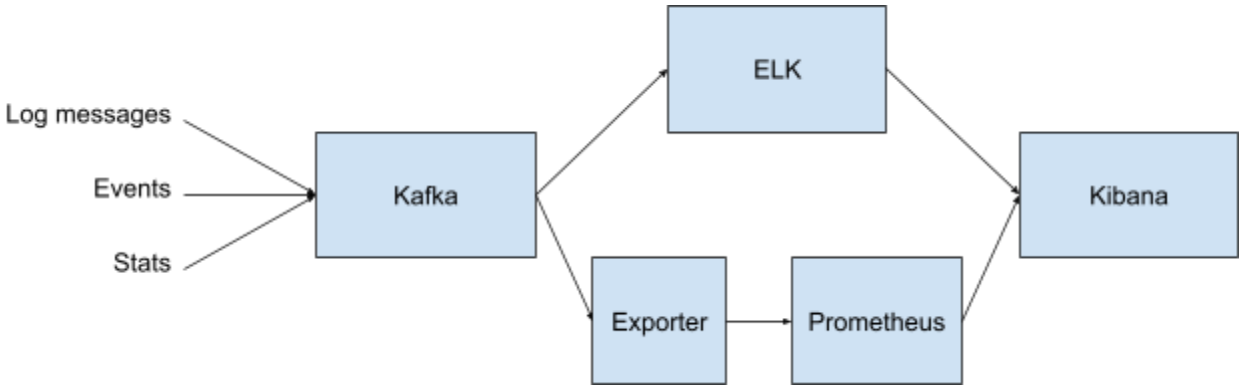
Ideally it should take advantage of already existing open-source tools as much as possible.

Architecture

At a very high level the proposed architecture is to have each component to push its logs and events into Kafka (either directly or through the usage of a sidecar container, as a longer term plan to support third party components).

Once the informations are in Kafka they'll be picked up by:

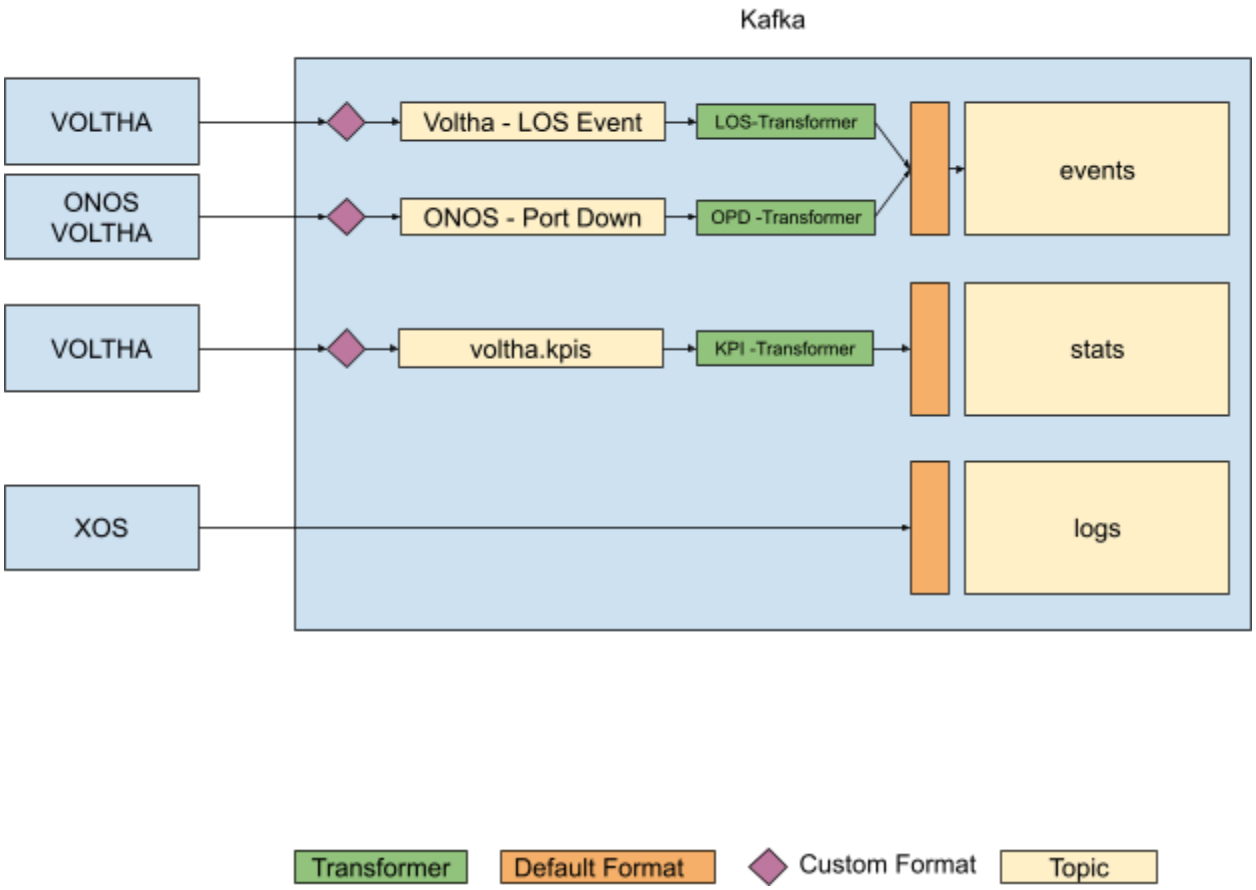
- Elk Stack: if they are logs
- Prometheus: if they are time-series kind of data or events/alarms



There is the need to have a default format for the events that comes out of Kafka, but given the variety of components we need to monitor is not realistic to imagine that all of them will adhere to that standard format. The proposed solution is to implement, where needed, kafka transformers, to format messages in the expected format.

The following path are possible for a message that reaches Kafka:

- The message comply with the default format
    - It will be directly published on the corresponding topic
- The message has a custom format
    - It will be published on a private topic
    - A transform will pick it up and format it
    - Events will be enriched with model/graph information
    - The transformed message will published on the corresponding topic



Messages Schema

The schemas for the messages has yet to be defined, but it should agnostic toward the northbound platform that is going to consume it (whether it's Prometheus, ONAP or any other OSS).

Some informations that are common to every message are:

- Component that originate the message
- Identifier (the more specific the better)
    - Central Office
    - Component originating the message
    - Rack?
    - Device Id / Serial Number?
- Timestamp

Events

Messages on the Event topic are meant to convey standalone informations. For example: PortUp/Down, LossOfSignal, …
They are not called alarms as the level at which an event is considered an Alarm is defined by the operator and that should be configured in Prometheus.

Stats

Messages on the Stats topic are meant to convey usage statistic information. For example: PacketsIn/Out, …
There are different kind of statistics that can be collected:

- Counters
- Time Series
- Gauges
- …

Logs

Messages on the Logs topic are meant to convey information about code execution to help post mortem debug. The end goal is to be able to track an action across the stack.

For example when we activate a subscriber multiple components are involved: XOS, ONOS, VOLTHA. In the log aggregator we should be able to track all the operations that led to the creation of a Subscriber in the different components.

TL;DR; Kafka messages to Prometheus metrics

Prometheus uses a Pull based mechanism to ingest data, and it does not have a direct Kafka integration.

The solution to this is to implement an Exporter. An exporter will gain information about the process/component you want to monitor and format the data into valid metric format.

By definition a metric is expressed as:

```
<metric name>{<label name>=<label value>, ...}
```

For example:

```
api_http_requests_total{method="POST", handler="/messages"}
```

Metrics are stored a flattened untyped time series. This allow us to store the smallest possible portion of data, and Labels are used to aggregate and query this informations making them meaningful.

We will need to implement a custom exporter to convert our message schema to prometheus metrics. Take a look at this blog post for an example: https://www.robustperception.io/writing-json-exporters-in-python/

For example, assume that we are receiving a kpi event from VOLTHA, defined as:

```
"prefixes":{
     "voltha.openolt.000130af0b0b2c51nni.128":{
        "metrics":{
           "rx_packets":0,
           "rx_bytes":0,
           ...
           "tx_bytes":0,
           "tx_packets":0
        }
     }
  },
  "type":"slice",
  "ts":1531266048.0
```

We need to have an exporter that listens to those messages, parses the data and updates multiple meaningful metrics:

```
rx_packets {olt="000130af0b0b2c51", port_type="nni", port_id="128"}

tx_packets {olt="000130af0b0b2c51", port_type="nni", port_id="128"}

rx_bytes {olt="000130af0b0b2c51", port_type="nni", port_id="128"}

tx_bytes {olt="000130af0b0b2c51", port_type="nni", port_id="128"}
```