NVIDIA DOCA Telemetry

Programming Guide

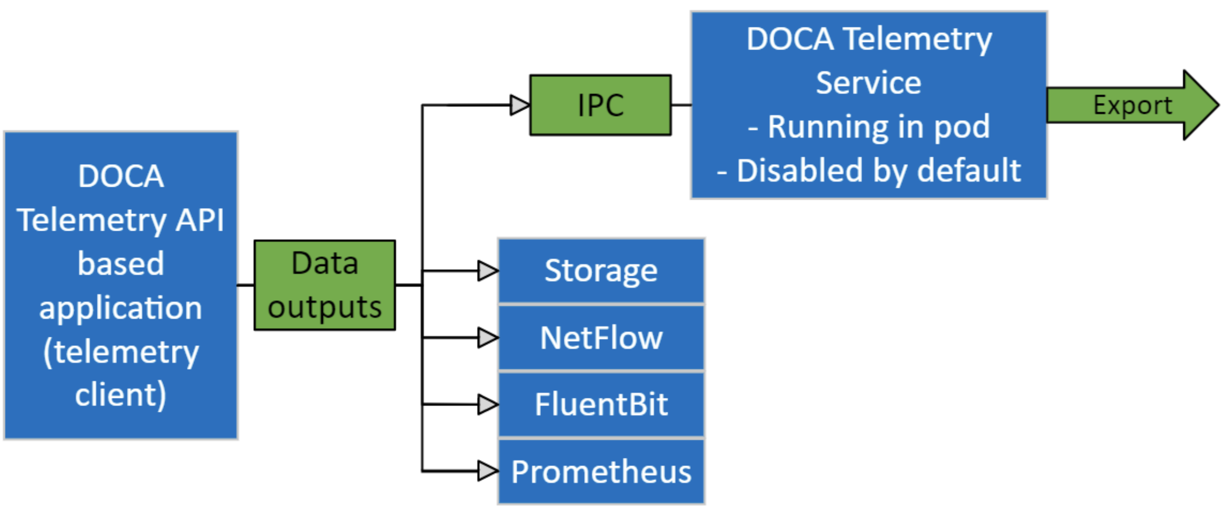
MLNX-15-060506 _v1.3 | May 2022

Table of Contents

Chapter 1. Introduction

DOCA Telemetry API offers a fast and convenient way to transfer user-defined data to DOCA Telemetry Service (DTS). In addition, the API provides several built-in outputs for user convenience, including saving data directly to storage, NetFlow, Fluent Bit forwarding, and Prometheus endpoint.

The following figure shows an overview of the telemetry API. The telemetry client side, based on the telemetry API, collects user-defined telemetry and sends it to the DTS which runs as a container on BlueField. DTS does further data routing, including export with filtering. DTS can process several user-defined telemetry clients and can collect pre-defined counters by itself. Additionally, telemetry API has built-in data outputs that can be used from telemetry client applications.



Several scenarios are available

- Send data via IPC transport to DTS. For IPC, refer to Inter-process Communication.
- Write data as binary files to storage (for debugging data format).
- Export data directly from DOCA Telemetry API application using the following options:
  - Fluent Bit exports data through forwarding
  - NetFlow exports data from NetFlow API. Available from both API and DTS. See details in Data Outputs.

- Prometheus creates Prometheus endpoint and keeps the most recent data to be scraped by Prometheus.

Users can either enable or disable any of the data outputs mentioned above. See Data Outputsto see how to enable each output.

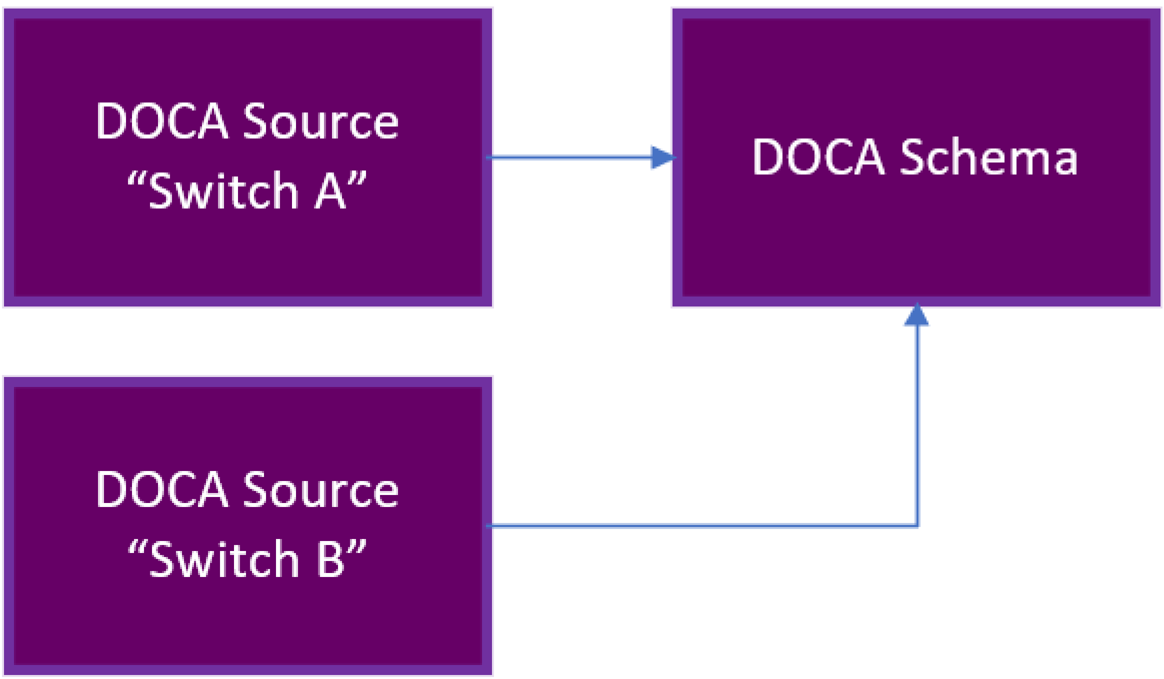
The library stores data in an internal buffer and flushes it to DTS/exporters in the following scenarios:

- Once the buffer is full. Buffer size is configurable with different attributes.
- When `doca_telemetry_source_flush(void *doca_source)` function is invoked.
- When the telemetry client terminates. If the buffer has data, it is processed before the library's context cleanup.

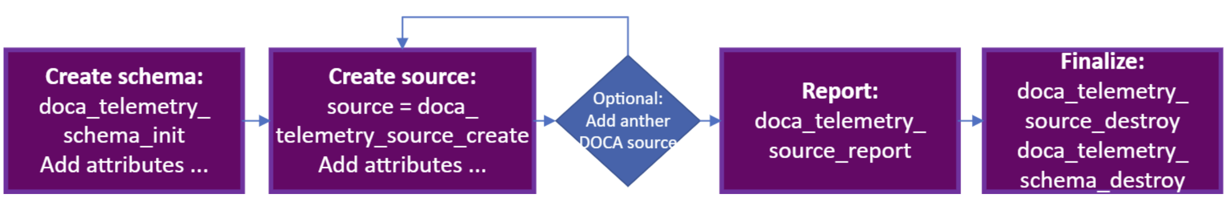
Chapter 2. Architecture

DOCA Telemetry API is fundamentally built around four major parts:

- DOCA schema – defines a reusable structure of telemetry data which can be used by multiple sources



- Source – the unique identifier of the telemetry source that periodically reports telemetry data.
- Report – exports the information to the DTS
- Finalize – releases all the resources



2.1. DOCA Telemetry API Walkthrough

Here is a basic walkthrough of the needed steps for using the DOCA Telemetry API.

1. Create `doca_schema`.
   a). Initialize an empty schema with default attributes:
   ```
   doca_telemetry_schema_init("example_doca_schema_name")
   ```
   b). Set the following attributes if needed:
   - `doca_telemetry_schema_buffer_attr_set(…)`
   - `doca_telemetry_schema_file_write_attr_set(…)`
   - `doca_telemetry_schema_ipc_attr_set(…)`
   - `doca_telemetry_ipc_timeout_attr_t(…)`

   c). Add user event types:
   ```
   doca_telemetry_schema_add_type(doca_schema, "example_event", example_fields,
    NUM_OF_DOCA_FIELDS(example_fields), &example_index);
   ```
   d). Apply attributes and types to start using:
   ```
   doca_schema doca_telemetry_schema_start(doca_schema)
   ```
2. Create `doca_source`:
   a). Initialize:
   ```
   doca_source: source = doca_telemetry_source_create(doca_schema);
   ```
   b). Set source ID and tag:
   ```
   doca_telemetry_source_name_attr_set(doca_source, &soruce_attr)
   ```
   c). Apply attributes to start using source:
   ```
   doca_telemetry_source_start(doca_source)
   ```
   You may optionally add more `doca_source`s.
3. Collect the data per source and use:
   ```
   doca_telemetry_source_report(source, event_index, &my_app_test_ev1, num_events)
   ```
4. Finalize:
   a). For every source:
   ```
   doca_telemetry_source_destroy(source)
   ```
   b). Destroy:
   ```
   doca_telemetry_schema_destroy(doca_schema)
   ```

Please find example implementation in the `telemetry_export.c` DOCA sample.

2.2. DOCA Telemetry NetFlow API Walkthrough

The DOCA telemetry API also supports NetFlow using DOCA Telemetry NetFlow API. This API is designed to allow customers to easily support the NetFlow protocol at the endpoint side. Once an endpoint produces NetFlow data the API, the corresponding exporter can be used to send the data to a NetFlow collector.

The NVIDIA DOCA Telemetry Netflow API's definitions can be found in the `doca_telemetry_netflow.h` file.

The following are the steps to use the NetFlow API:

1. Initiate the API with an appropriate source ID:
   ```
   doca_telemetry_netflow_init(source_id)
   ```
2. Set the relevant attributes:
   - `doca_telemetry_netflow_buffer_attr_set(…)`
   - `doca_telemetry_netflow_file_write_attr_set(…)`
   - `doca_telemetry_netflow_ipc_attr_set(…)`
3. Start the API with the relevant struct:
   ```
   doca_telemetry_source_name_attr_t:
             doca_telemetry_netflow_start(&source_attr)
   ```
4. Form a desired NetFlow template and the corresponding NetFlow records.
5. Collect the NetFlow data.
   ```
   doca_telemetry_netflow_send(…)
   ```
6. (Optional) Flush the NetFlow data to send data immediately instead of waiting for the buffer to fill:
   ```
   doca_telemetry_netflow_flush()
   ```
7. Clean up the API:
   ```
   doca_telemetry_netflow_destroy()
   ```

Chapter 3. API

Refer to NVIDIA DOCA Libraries API Reference Manual, for more detailed information on DOCA Telemetry API.

Note: The pkg-config (`*.pc` file) for the telemetry library is named `doca-telemetry`.

The following sections provide additional details about the library API.

3.1. doca_telemetry_buffer_attr_t

This attribute is used to set the internal buffer size used by all DOCA sources.

Configuring the attribute is optional as it is initialized with default values.

```
struct doca_telemetry_buffer_attr_t {
 uint64_t  buffer_size;
 char *data_root;
};
```

Where:

- `buffer_size` – the size of the internal buffer which accumulates the data before sending it to the outputs. Data is sent automatically once the internal buffer is full. Larger buffers mean fewer data transmissions and vice versa. The default value for this field is 60,000 (bytes).
- `data_root` – the path to where data is stored (if `file_write_enabled` is set to true)

3.2. doca_telemetry_ipc_attr_t

This attribute is used to enable/disable ipc transport. It is disabled by default.

Note: It is important to make sure that the IPC location matches the IPC location used by DTS, otherwise IPC communication fails.

```
struct doca_telemetry_ipc_attr_t {
 bool  ipc_enabled;
 char *ipc_sockets_dir;
};
```

Where:

- `ipc_enabled` – boolean flag for enabling/disabling IPC transport. Its default value is `false`.
- `ipc_sockets_dir` – a directory that contains UDS for IPC messages. Both the telemetry program and DTS must use the same folder. DTS that runs on BlueField as a container has the default folder `/opt/mellanox/doca/services/telemetry/ipc_sockets`.

3.3. doca_telemetry_source_name_attr_t

This attribute is used to create proper folder structure. All the data collected from the same host is written to the `source_id` folder under data root.

> **Note:** It is important to make sure that the IPC location matches the IPC location used by DTS, otherwise IPC communication fails.

```
struct doca_telemetry_source_name_attr_t {
 char *source_id;
 char *source_tag;
};
```

Where:

- `source_id` – describes the data's origin. It is recommended to set it to the hostname. In later dataflow steps, data is aggregated from multiple hosts/DPUs and `source_id` helps navigate in it.
- `source_tag` – a unique data identifier. It is recommended to set it to describe the data collected in the application. Several telemetry apps can be deployed on a single node (host/DPU). In that case, each telemetry data would have a unique tag and all of them would share a single `source_id`.

> **Note:** Users would not be able to start context without overwriting `source_id` and `source_tag`. The fields are mandatory to set because they are used for further data routing.

3.4. doca_telemetry_netflow_send_attr_t

DOCA Telemetry NetFlow API attribute is optional and should only be used for debugging purposes. It represents the NetFlow collector's address while working locally, effectively enabling the local NetFlow exporter. When set to NULL, no traffic is sent.

```
struct doca_telemetry_netflow_send_attr_t {
 char *netflow_collector_addr;
 uint16_t netflow_collector_port;
};
```

Where:

- `netflow_collector_addr` – NetFlow collector's address (IP or name). Its default value is `NULL`.
- `netflow_collector_port` – NetFlow collector's port. Its default value is 0.

3.5. doca_telemetry_source_report

The source report function is the heart of communication with the DTS. The report operation causes event data to be allocated to the internal buffer. Once the buffer is full, data is forwarded onward according to the set configuration.

```
int doca_telemetry_source_report(void *doca_source,
               doca_telemetry_type_index_t index,
               void *data, int count)
```

Where:

- doca_source [in] – a pointer to the doca_source which reports the event
- index [in] – the event type index received when schema was created, matching the data pointer
- data [in] – a pointer to the data buffer that needs to be sent, matching the index
- count [in] – numbers of events to be written to the internal buffer, according to the data pointer

The function returns 0 on if successful, or an internal buffer error if the buffer's size is too small. If an error occurs, try a larger buffer size that matches the event's size.

3.6. doca_telemetry_schema_add_type

This function allows adding a reusable telemetry data struct, also known as a schema. The schema allows sending a predefined data structure to the telemetry service. Note that it is mandatory to define a schema for proper functionality of the library. After adding the scehmas, one needs to invoke the schema start function.

```
int doca_telemetry_schema_add_type(void *doca_schema,
           const char *new_type_name,
           doca_telemetry_field_info_t *fields,
           int num_fields,
           doca_telemetry_type_index_t *type_index);
```

Where:

- doca_schema [in] – a pointer to the schema to which the type is added
- new_type_name [in] – name of the new type
- fields [in] – user-defined fields to be used for the schema. Multiple fields can (and should) be added.
- num_fields [in] – number of user-defined fields. Use NUM_OF_DOCA_FIELDS marco.
- type_index [out] – type index for the created type is written to this output variable

The function returns 0 on if successful, or a negative telemetry_status if an error occurs.

Chapter 4. Telemetry Data Format

The internal data format consists of 2 parts: a schema containing metadata, and the actual binary data. When data is written to storage, the data schema is written in JSON format, and the data is written as binary files. In the case of IPC transport, both schema and binary data are sent to DTS. In the case of export, data is converted to the formats required by exporter.

Adding custom event types to the schema can be done using the following API call:

```
int doca_telemetry_schema_add_type(void *doca_schema,
           const char *new_type_name,
           doca_telemetry_field_info_t *fields,
           int num_fields,
           doca_telemetry_type_index_t *type_index);
```

Where the `example_fields` variable contains the list of fields in the following format:

```
{NAME, DESCRIPTION, DOCA_TELEMETRY_FIELD_TYPE, NUM_OF_ITEMS}
```

> **Note:** See available `DOCA_TELEMETRY_FIELD_TYPE`s in `doca_telemetry.h`. See example of usage in `/opt/mellanox/doca/samples/doca_telemetry/telemetry_export/telemetry_export.c`.

> **Note:** It is highly recommended to have the `timestamp` field as the first field since it is required by most databases. To get the current timestamp in the correct format use:
>
> ```
> doca_telemetry_timestamp_t doca_telemetry_timestamp_get(void);
> ```

Chapter 5. Data Outputs

This section describes available exporters:

- IPC
- NetFlow
- Fluent Bit
- Prometheus

FluentBit and Prometheus exporters are presented in both API and DTS. Even though DTS export is preferable, the API has the same possibilities for development flexibility.

5.1. Inter-process Communication

IPC transport automatically transfers the data from the telemetry-based program to DTS service.

It is implemented as a UNIX domain socket (UDS) sockets for short messages and shared memory for data. DTS and the telemetry-based program must share the same `ipc_sockets` directory.

When IPC transport is enabled, the data is sent from the DOCA-telemetry-based application to the DTS process via shared memory.

To enable IPC, set `ipc_enabled=1` of `doca_telemetry_ipc_attr_t` to the `doca_source`.

Note that IPC transport relies on system folders. For the host usage run the DOCA-telemetry-API-based application with `sudo` to be able to use IPC with system folders.

To check the status of IPC for current context, use:

```
int status = doca_telemetry_check_ipc_status (doca_source)
```

If IPC is enabled and for some reason connection is lost, it would try to automatically reconnect on every report's function call.

5.2. NetFlow

When the NetFlow exporter is enabled (`doca_telemetry_netflow_send_attr_t` set), it sends the NetFlow data to the NetFlow collector specified by the `doca_telemetry_netflow_send_attr_t` fields: Address and port. This exporter must be used when using DOCA Telemetry Netflow API.

5.3. FluentBit

FluentBit export is based on `fluent_bit_configs` with `.exp` files for each destination. Every export file corresponds to one of FluentBit's destinations. All found and enabled `.exp` files are used as separate export destinations. Examples can be found after running DTS container under its configuration folder (`/opt/mellanox/doca/services/telemetry /config/ fluent_bit_configs/`).

All `.exp` files are documented in-place.

```
DPU# ls -l /opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/
/opt/mellanox/doca/services/telemetry/config/fluent_bit_configs/:
total 56
-rw-r--r-- 1 root root  528 Oct 11 07:52 es.exp
-rw-r--r-- 1 root root  708 Oct 11 07:52 file.exp
-rw-r--r-- 1 root root 1135 Oct 11 07:52 forward.exp
-rw-r--r-- 1 root root  719 Oct 11 07:52 influx.exp
-rw-r--r-- 1 root root  571 Oct 11 07:52 stdout.exp
-rw-r--r-- 1 root root  578 Oct 11 07:52 stdout_raw.exp
-rw-r--r-- 1 root root 2137 Oct 11 07:52 ufm_enterprise.fset
```

FluentBit `.exp` files have 2-level data routing:

- `source_tags` in `.exp` files (documented in-place)
- Token-based filtering governed by `.fset` files (documented in `ufm_enterprise.fset`)

To run with FluentBit exporter, set `enable=1` in required `.exp` files and set the environment variables before running the application:

```
export FLUENT_BIT_EXPORT_ENABLE=1
export FLUENT_BIT_CONFIG_DIR=/path/to/fluent_bit_configs
export LD_LIBRARY_PATH=/opt/mellanox/collectx/lib
```

5.4. Prometheus

Prometheus exporter sets up endpoint (HTTP server) which keeps the most recent events data as text records.

The Prometheus server can scrape the data from the endpoint while the DOCA-Telemetry-API-based application stays active.

Check the generic example of Prometheus records:

```
event_name_1{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val",
 label_4="label_4_val"} counter_value_1 timestamp_1
event_name_2{label_1="label_1_val", label_2="label_2_val", label_3="label_3_val",
 label_4="label_4_val"} counter_value_2 timestamp_2
...
```

Labels are customizable metadata which can be set from data file. Events names could be filtered by token-based name-match according to .fset files.

Set the following environment variables before running.

```
# Set the endpoint host and port to enable export.
export PROMETHEUS_ENDPOINT=http://0.0.0.0:9101
```

```
# Set indexes as a comma-separated list to keep data for every index field. In
# this example most recent data will be kept for every record with unique
# `port_num`. If not set, only one data per source will be kept as the most
# recent.
export PROMETHEUS_INDEXES=Port_num

# Set path to a file with Prometheus custom labels. Use labels to store
# information about data source and indexes. If not set, the default labels
# will be used.
export CLX_METADATA_FILE=/path/to/labels.txt

# Set the folder which contains fset-files. If set, Prometheus will scrape
# only filtered data according to fieldsets.
export PROMETHEUS_CSET_DIR=/path/to/prometheus_cset
```

Prometheus labels can be obtained from file.

Note: To scrape the data without Prometheus server use:

```
curl -s http://0.0.0.0:9101/metrics
```

Or:

```
curl -s http://0.0.0.0:9101/{fset_name}
```

Notice

Trademarks

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com