NVIDIA DOCA Telemetry

Programming Guide

MLNX-15-060506 _v1.2 | January 2022

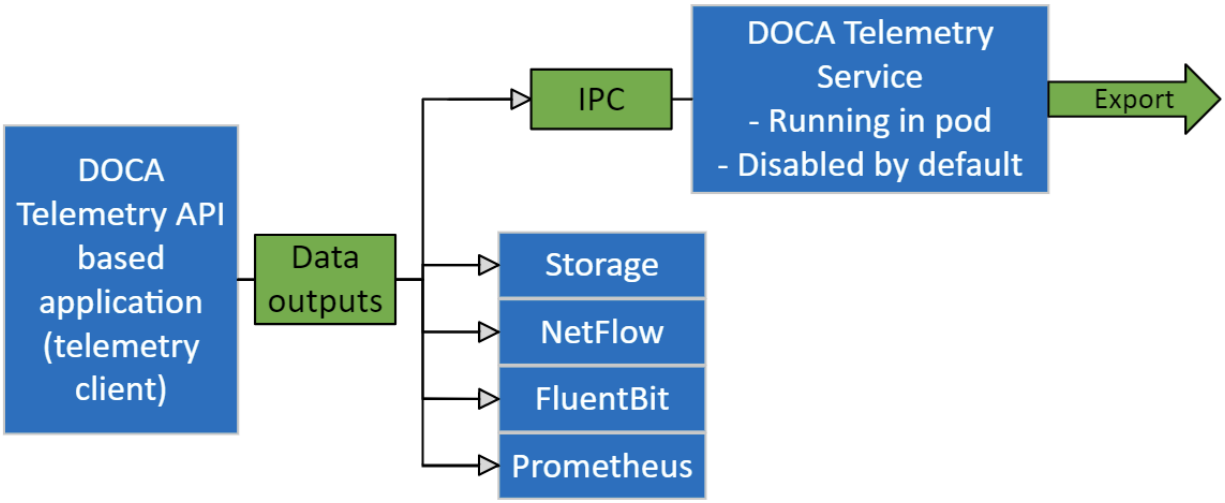
Table of Contents

Chapter 1. Introduction 1
1.1. Telemetry Data Format 2
Chapter 2. Getting Started 3
2.1. DOCA Telemetry Concepts 3
2.2. Attributes 3
2.2.1. Schema Attributes 3
2.2.2. Source Attributes 5
2.3. Logic of App Development 5
Chapter 3. DOCA Telemetry NetFlow API 7
3.1. DOCA Telemetry NetFlow Concepts 7
3.2. Attributes 7
3.3. Logic of App Development 8
Chapter 4. Data Outputs 9
4.1. Inter-Process Communication 9
4.2. NetFlow 9
4.3. FluentBit 10
4.4. Prometheus 10

Chapter 1. Introduction

DOCA telemetry API offers a fast and convenient way to collect user-defined data and transfer it 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. DOCA Telemetry API is built on the CLX_API package which is provided as part of the BlueField image installation.

The following figure shows the purpose of the telemetry API. The telemetry client, 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.

1.1. 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 `examples/telemetry/telemetry_config.h`.

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 2. Getting Started

DOCA Telemetry API is built as a shared object library, `libdoca_telemetry.so`.

All available types and functions are defined in the library's `.h` file, `doca_telemetry.h`.

2.1. DOCA Telemetry Concepts

DOCA Telemetry API is based on the following concepts:

- `doca_schema` – an abstraction that contains user types and context that is shared between several `doca_sources`. Shared context includes data writer, IPC transport and exporter routines. The `doca_schema` is configurable through schema attributes (see Schema Attributes).
- `doca_source` – the user's access point to report collected events. Each data source is created based on a `doca_schema` and must use a unique source tag identifier. All sources based on the same schema share a single source ID. See Source Attributes for the details on source ID and tag.
- `attributes` – structs containing initialization parameters for `doca_schema` and `doca_source`.
- `event_type` – a list of fields with data type, name, description, and length of array. Types must be registered in `doca_schema`. Each type has an index that the API sets when an event is registered.
- `event_buffer` – data buffer that corresponds to a `doca_schema` event type.
- `event` - actual data collected according to array type. Events are collected by the user, placed in the `event_buffer` and reported through `doca_source`.

2.2. Attributes

Users can set attributes for `doca_schema` and `doca_source`.

2.2.1. Schema Attributes

The `doca_schema` is configured with a set of default values. Modifying the initial attributes is optional.

The following attributes can be configured:

Struct Name	Field	Default Field Value	Description
doca_telemetry_buffer_attr_t	uint64_t buffer_size	60000 (bytes)	The size of the internal buffer which accumulates the data before sending it to outputs. Data is sent automatically once the internal buffer is full. Larger buffers mean less data transmissions and vice versa.
	char *data_root	"doca_telemetry_client_data" (under the current folder)	The path for where data is stored (if `file_write_enabled` is set to true)
doca_telemetry_file_writer_attr_t	bool file_write_enabled	False	The Boolean flag for enabling/disabling dumping of data to disk (under `data_root`)
	size_t max_file_size	1 * 1024 * 1024; // 1 MB	Size limit for a data file. Once a file reaches that limit, data writer switches to the next file.
	doca_telemetry_timestamp_t max_file_age	60 * 60 * 1000000L; // 1 hour	Time limit for the data file. Once a file reaches that limit, data writer switches to the next file.
doca_telemetry_ipc_attr_t	bool ipc_enabled	false	Boolean flag for enabling/disabling IPC transport
	char *ipc_sockets_dir	"/tmp/ipc_sockets"	A directory which contains UDS for IPC messages. Both client telemetry application and DTS must use the same folder. DTS that runs on BlueField as a container has the following default folder: `/opt/mellanox/doca/services/telemetry/ipc_sockets`.
doca_telemetry_ipc_timeout_t	uint32_t ipc_max_reconnect_time_msec;	500 msec	Time limit for reconnection attempts. If the limit is reached, the client is considered disconnected.
	int ipc_max_reconnect_tries;	3 times	Maximum number of reconnection attempts

Struct Name	Field	Default Field Value	Description
			during reconnection period
	uint32_t ipc_socket_timeout_msec;	3000 (3 sec)	Timeout for IPC messaging socket. If timeout is reached during `send_receive`, client is considered disconnected.

2.2.2. Source Attributes

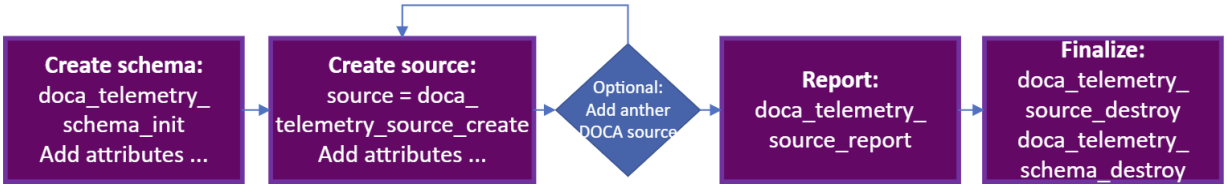
It is mandatory to set the `doca_source` attribute.

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.

Type	Field	Default Field Value	Description
struct doca_telemetry_source_name_attr_t	char *source_id	"DEFAULT_SOURCE"	`source_id` describes the origin of data. It is recommended to set it to hostname. In later dataflow steps, data is aggregated from multiple hosts/DPUs and `source_id` helps navigate in it.
	char *source_tag	"DEFAULT_TAG"	`source_tag` is the unique data identifier. It is recommended to set it to describe the data collected in the application. Several telemetry apps can be run 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`.

2.3. Logic of App Development

To summarize the implementation of telemetry in a new application, users are required to follow these steps:



1. Create `doca_schema`.

   a). Initialize 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 with:
   ```
   doca_telemetry_source_name_attr_set(doca_source, &soruce_attr)
   ```
   c). Apply attributes to start using source.
   ```
   doca_telemetry_source_start(doca_source)
   ```
3. Optionally add more `doca_sources`.
4. Collect the data per source and use.
   ```
   doca_telemetry_source_report(source, event_index, &my_app_test_ev1, num_events)
   ```
5. Finalize:

   a). For every source:
   ```
   doca_telemetry_source_destroy(source)
   ```
   b). Destroy.
   ```
   doca_telemetry_schema_destroy(doca_schema)
   ```

Please find example implementation in `telemetry_config.c`.

Chapter 3. DOCA Telemetry NetFlow API

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.

3.1. DOCA Telemetry NetFlow Concepts

DOCA Telemetry NetFlow API is based on the following concepts:

- The API operates in NetFlow-related terms (source ID, template, package record etc.)
- `attributes` are structs containing initialization parameters for the API
- In addition to the `attributes` provided by the DOCA Telemetry API, the API provides the attribute `doca_telemetry_netflow_send_attr_t` which represents the NetFlow collector's address while working locally, effectively enabling the local NetFlow Exporter (see Attributes)

3.2. Attributes

Users can set DOCA Telemetry NetfFow API attributes. The attributes are optional and should only be used for debugging purposes.

The following attributes are available:

Type	Field	Default Field Value	Description
struct doca_telemetry_netflow_send_attr	char *netflow_collector_addr	NULL	NetFlow collector's address (IP or name)
	uint16_t netflow_collector_port	0	NetFlow collector's port

3.3. Logic of App Development

To summarize the implementation of telemetry NetFlow in a new application, users are required to follow these steps:

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. Clean up the API.
   ```
   doca_telemetry_netflow_destroy()
   ```

Chapter 4. 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.

4.1. Inter-Process Communication

IPC transport automatically transfers the data from the telemetry client application to DTS service.

It is implemented as UD sockets for short messages and shared memory for data. DTS and telemetry client must share the same `ipc_sockets` directory (see Schema Attributes).

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 exploits 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.

4.2. 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.

4.3. 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-agent/config/fluent_bit_configs/`). All `.exp` files are documented in-place.

```
DPU# ls -l /opt/mellanox/doca/services/telemetry-agent/config/fluent_bit_configs/
/opt/mellanox/doca/services/telemetry-agent/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
```

4.4. 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

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation nor any of its direct or indirect subsidiaries and affiliates (collectively: "NVIDIA") make no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assume no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, and Mellanox are trademarks and/or registered trademarks of Mellanox Technologies Ltd. and/or NVIDIA Corporation in the U.S. and in other countries. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world¬wide basis. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

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