
Fast DDS Statistics Backend Documentation

Release 0.8.0

eProsima

Oct 28, 2022

INTRODUCTION

1	Description	3
2	Linux installation from sources	5
2.1	Fast DDS Statistics Backend installation	5
3	CMake options	9
4	StatisticsBackend	11
4.1	Initialize a monitor	11
4.2	Stop and restart a monitor	13
4.3	Clearing data	13
4.4	Reset Fast DDS Statistics Backend	14
4.5	Set listeners	14
4.6	Get entities' graph	15
4.7	Get entity meta information	22
4.8	Get entities of a given kind	30
4.9	Get statistical data	32
4.10	Get entity type	36
4.11	Set entity alias	36
4.12	Check whether an entity is active	36
4.13	Check whether an entity is builtin	37
4.14	Saving and restoring the statistics data	37
5	Types	39
5.1	DataKind	39
5.2	StatisticKind	40
5.3	EntityKind	41
5.4	EntityId	41
5.5	JSON Tags	43
5.6	Database dumps	50
6	Listeners	51
6.1	DomainListener	51
6.2	PhysicalListener	52
7	Full example	53
7.1	Next steps	53
8	API Reference	55
8.1	Exception	55
8.2	Listener	58

8.3	StatisticsBackend	61
8.4	Types	70
9	Version 0.8.0	79
10	Previous versions	81
10.1	Version 0.7.1	81
10.2	Version 0.7.0	81
10.3	Version 0.6.0	81
10.4	Version 0.5.0	82
10.5	Version 0.4.0	82
10.6	Version 0.3.0	82
10.7	Version 0.2.0	83
10.8	Version 0.1.0	83
Index		85



eProsima Fast DDS Statistics Backend is a C++ library to collect data from the *Fast DDS Statistics module*, and generate statistical information to be used by applications.

Warning: This library is in the early stages of development. For this reason, the API should not be considered stable, as API breaks may occur before the first official release. Furthermore, several features may not be implemented yet.

Warning: Be aware that the statistics module is not compiled and used by default by *Fast DDS* but it has to be specifically configured to send statistical data of an specific entity. Therefore, *eProsima Fast DDS* library must be compiled with `FASTDDS_STATISTICS` CMake option enabled and use either the `FASTDDS_STATISTICS` environment variable or the `DomainParticipantQos` to activate the statistics DataWriters. For more information, please refer to the [Statistics Module documentation](#).

Warning: If *Fast DDS* has been compiled with statistics and they are explicitly enabled and statistical data are not correctly received, only few data arrive or even none, configure the *Fast DDS* endpoints publishing statistics data with a less restrictive memory constraints. Please check the following [documentation](#) for more details on how to do this.



DESCRIPTION

eProxima Fast DDS Statistics Backend is a C++ library to collect data from the *Fast DDS Statistics module*, and generate statistical information to be used by applications.

Warning: This library is in the early stages of development. For this reason, the API should not be considered stable, as API breaks may occur before the first official release. Furthermore, several features may not be implemented yet.

Warning: Be aware that the statistics module is not compiled and used by default by *Fast DDS* but it has to be specifically configured to send statistical data of a specific entity. Therefore, *eProxima Fast DDS* library must be compiled with `FASTDDS_STATISTICS` CMake option enabled and use either the `FASTDDS_STATISTICS` environment variable or the `DomainParticipantQos` to activate the statistics `DataWriters`. For more information, please refer to the [Statistics Module documentation](#).

Warning: If *Fast DDS* has been compiled with statistics and they are explicitly enabled and statistical data are not correctly received, only few data arrive or even none, configure the *Fast DDS* endpoints publishing statistics data with a less restrictive memory constraints. Please check the following [documentation](#) for more details on how to do this.

LINUX INSTALLATION FROM SOURCES

The instructions for installing the *eProxima Fast DDS Statistics Backend* from sources are provided in this page. It is organized as follows:

- *Fast DDS Statistics Backend installation*
 - *Requirements*
 - *Dependencies*
 - *Colcon installation*
 - *CMake installation*

2.1 Fast DDS Statistics Backend installation

This section describes the instructions for installing *eProxima Fast DDS Statistics Backend* in a Linux environment from sources. First of all, the *Requirements* and *Dependencies* detailed below need to be met. Afterwards, the user can choose whether to follow either the *colcon* or the *CMake* installation instructions.

2.1.1 Requirements

The installation of *eProxima Fast DDS Statistics Backend* in a Linux environment from sources requires the following tools to be installed in the system:

- *CMake*, *g++*, *pip3*, *wget* and *git*
- *Gtest* [optional]

CMake, g++, pip3, wget and git

These packages provide the tools required to install *eProxima Fast DDS Statistics Backend* and its dependencies from command line. Install *CMake*, *g++*, *pip3*, *wget* and *git* using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install cmake g++ python3-pip wget git
```

Gtest

Gtest is a unit testing library for C++. By default, *eProsima Fast DDS Statistics Backend* does not compile tests. It is possible to activate them with the opportune [CMake configuration options](#) when calling `colcon` or `CMake`. For more details, please refer to the [CMake options](#) section. For a detailed description of the Gtest installation process, please refer to the [Gtest Installation Guide](#).

Note: *eProsima Fast DDS Statistics Backend* depends on Gtest release-1.10.0 or later.

2.1.2 Dependencies

eProsima Fast DDS Statistics Backend has the following dependencies in a Linux environment:

- *eProsima Fast DDS*

eProsima Fast DDS

Please, refer to the [eProsima Fast DDS](#) installation documentation to learn the installing procedure

2.1.3 Colcon installation

`colcon` is a command line tool based on [CMake](#) aimed at building sets of software packages. This section explains how to use it to compile *eProsima Fast DDS Statistics Backend* and its dependencies.

1. Install the ROS 2 development tools (`colcon` and `vcstool`) by executing the following command:

```
pip3 install -U colcon-common-extensions vcstool
```

Note: If this fails due to an Environment Error, add the `--user` flag to the `pip3` installation command.

2. Create a `Fast-DDS-statistics-backend` directory and download the `repos` file that will be used to install *eProsima Fast DDS Statistics Backend* and its dependencies:

```
mkdir ~/Fast-DDS-statistics-backend
cd ~/Fast-DDS-statistics-backend
wget https://raw.githubusercontent.com/eProsima/Fast-DDS-statistics-backend/master/
↪fastdds_statistics_backend.repos
mkdir src
vcs import src < fastdds_statistics_backend.repos
```

3. Build the packages:

```
colcon build
```

Note: Being based on [CMake](#), it is possible to pass the CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the [CMake specific arguments](#) page of the `colcon` manual.

Instead of passing CMake configuration options on the CLI, it is also possible to use a `colcon.meta` file to set the configuration. The *eProsima Fast DDS Statistics Backend* repository already includes a `colcon.meta` file with the default configuration, which can be tuned by the user.

2.1.4 CMake installation

This section explains how to compile *eProsima Fast DDS Statistics Backend* with CMake, either *locally* or *globally*.

Local installation

1. Follow the [eProsima Fast DDS local installation guide](#) to install *eProsima Fast DDS* and all its dependencies
2. Install *eProsima Fast DDS Statistics Backend*:

```
cd ~/Fast-DDS
git clone https://github.com/eProsima/Fast-DDS-statistics-backend.git
mkdir Fast-DDS-statistics-backend/build
cd Fast-DDS-statistics-backend/build
cmake .. -DCMAKE_INSTALL_PREFIX=~/.Fast-DDS/install -DCMAKE_PREFIX_PATH=~/.Fast-DDS/
->install
sudo cmake --build . --target install
```

Note: By default, *eProsima Fast DDS Statistics Backend* does not compile tests. However, they can be activated by downloading and installing `Gtest`, and enabling *the corresponding cmake option*.

Global installation

1. Follow the [eProsima Fast DDS global installation guide](#) to install *eProsima Fast DDS* and all its dependencies
2. Install *eProsima Fast DDS Statistics Backend*:

```
cd ~/Fast-DDS
git clone https://github.com/eProsima/Fast-DDS-statistics-backend.git
mkdir Fast-DDS-statistics-backend/build
cd Fast-DDS-statistics-backend/build
cmake ..
cmake --build . --target install
```

Run an application

When running an instance of an application using *eProsima Fast DDS Statistics Backend*, it must be linked with the library where the packages have been installed, which in the case of system-wide installation is: `/usr/local/lib/` (if local installation is used, adjust for the correct directory). There are two possibilities:

- Prepare the environment locally by typing the command:

```
export LD_LIBRARY_PATH=/usr/local/lib/
```

- Add it permanently to the PATH, by typing:

```
echo 'export LD_LIBRARY_PATH=/usr/local/lib/' >> ~/.bashrc
```

CMAKE OPTIONS

eProsima Fast DDS Statistics Backend provides several CMake options for build configuration of the library.

Option	Description	Possible values	Default
BUILD_DOCUMENTATION	Build library documentation. Set to ON if BUILD_DOCUMENTATION_TESTS is set to ON.	ON OFF	OFF
BUILD_LIBRARY_TESTS	Build the library tests.	ON OFF	OFF
BUILD_DOCUMENTATION_TESTS	Build library documentation tests. Setting this ON will set BUILD_DOCUMENTATION to ON	ON OFF	OFF
BUILD_TESTS	Build the library and documentation tests. Setting this ON will set BUILD_LIBRARY_TESTS and BUILD_DOCUMENTATION_TESTS to ON	ON OFF	OFF
BUILD_SHARED_LIBS	Builds internal libraries as shared libraries, i.e. causes add_library() CMake function to create shared libraries if on. All libraries are built shared unless the library was explicitly added as a static library.	ON OFF	ON

STATISTICSBACKEND

Singleton *StatisticsBackend* is the entry point for applications that want to gather statistics information about a *Fast DDS* network using *Fast DDS* Statistics module. It provides the API necessary for starting and stopping monitorizations on a given domain or *Fast DDS* Discovery Server network, as well as the functions to extract statistics information about said monitorizations.

Fast DDS Statistics Backend can monitor several DDS domains and *Fast DDS* Discovery Server networks at the same time, notifying applications about changes in the network and arrival of new statistics data using two listeners which contain a set of callbacks that the application implements.

4.1 Initialize a monitor

Initializing a monitor on a certain Domain ID makes *eProsima Fast DDS Statistics Backend* start monitoring the statistics data and entity discoveries on that domain. No statistics data will be gathered unless there is a monitor initialized in the required domain.

StatisticsBackend provides three overloads of *init_monitor()* that can be used to start a monitorization on a DDS domain or a *Fast DDS* Discovery Server network.

```
// Init a monitor in DDS domain 0 with no listener associated.
EntityId domain_monitor_id =
    StatisticsBackend::init_monitor(0);

// Init a monitor for a Fast DDS Discovery Server network which server is located in IPv4
// address 127.0.0.1 and port 11811 using UDP as transport layer, and that uses the
↳ default GUID prefix
// eprosima::fastdds::rtps::DEFAULT_ROS2_SERVER_GUIDPREFIX.
// The monitor has no listener associated.
EntityId disc_server_monitor_id =
    StatisticsBackend::init_monitor("UDPv4:[127.0.0.1]:11811");

// Init a monitor for a Fast DDS Discovery Server network which server is located in IPv4
// address 127.0.0.1 and port 11811 using UDP as transport layer, and that uses the GUID
↳ prefix
// "44.53.01.5f.45.50.52.4f.53.49.4d.41".
// The monitor has no listener associated.
EntityId disc_server_prefix_monitor_id =
    StatisticsBackend::init_monitor("44.53.01.5f.45.50.52.4f.53.49.4d.41",
↳ "UDPv4:[localhost]:11811");
```

Furthermore, it is possible to initialize a monitor with a custom *DomainListener*. Please refer to *DomainListener* for more information about the *DomainListener* and its functionality.

```

CustomDomainListener domain_listener;

// Init a monitor in DDS domain 0 with a custom listener.
EntityId domain_monitor_id =
    StatisticsBackend::init_monitor(0, &domain_listener);

// Init a monitor for a Fast DDS Discovery Server network which server is located in IPv4
// address 127.0.0.1 and port 11811 using UDP as transport layer, and that uses the
↳default GUID prefix
// eprosima::fastdds::rtps::DEFAULT_ROS2_SERVER_GUIDPREFIX.
// The monitor uses a custom listener.
EntityId disc_server_monitor_id =
    StatisticsBackend::init_monitor("UDPv4:[127.0.0.1]:11811", &domain_listener);

// Init a monitor for a Fast DDS Discovery Server network which server is located in IPv4
// address 127.0.0.1 and port 11811 using UDP transport layer, and that uses the GUID
↳prefix
// "44.53.01.5f.45.50.52.4f.53.49.4d.41".
// The monitor uses a custom listener.
EntityId disc_server_prefix_monitor_id =
    StatisticsBackend::init_monitor("44.53.01.5f.45.50.52.4f.53.49.4d.41",
↳"UDPv4:[127.0.0.1]:11811",
    &domain_listener);

```

In addition, `init_monitor()` allows for specifying which monitorization events should be notified. This is done by setting a `CallbackMask` where the active callbacks from the listener are specified. Moreover, a mask on statistics data kind of interest can be set creating a `DataKindMask`

```

// Only get notifications when new data is available or when a new host is discovered
CallbackMask callback_mask = CallbackKind::ON_DATA_AVAILABLE | CallbackKind::ON_HOST_
↳DISCOVERY;

// Only get notificiations about network latency or subscription throughput
DataKindMask datakind_mask = DataKind::NETWORK_LATENCY | DataKind::SUBSCRIPTION_
↳THROUGHPUT;

CustomDomainListener domain_listener;

// Init a monitor in DDS domain 0 with a custom listener, a CallbackMask, and a
↳DataKindMask
EntityId domain_monitor_id =
    StatisticsBackend::init_monitor(0, &domain_listener, callback_mask, datakind_
↳mask);

// Init a monitor for a Fast DDS Discovery Server network which server is located in IPv4
// address 127.0.0.1 and port 11811 using UDP transport layer, and that uses the default
↳GUID prefix
// eprosima::fastdds::rtps::DEFAULT_ROS2_SERVER_GUIDPREFIX.
// The monitor uses a custom listener, a CallbackMask, and a DataKindMask.
EntityId disc_server_monitor_id =
    StatisticsBackend::init_monitor("UDPv4:[localhost]:11811", &domain_listener,
↳callback_mask,
    datakind_mask);

```

(continues on next page)

(continued from previous page)

```
// Init a monitor for a Fast DDS Discovery Server network which server is located in IPv4
// address 127.0.0.1 and port 11811 using UDP transport layer, and that uses the GUID_
↳prefix
// "44.53.01.5f.45.50.52.4f.53.49.4d.41".
// The monitor uses a custom listener, a CallbackMask, and a DataKindMask.
EntityId disc_server_prefix_monitor_id =
    StatisticsBackend::init_monitor("44.53.01.5f.45.50.52.4f.53.49.4d.41",
↳"UDPv4:[127.0.0.1]:11811",
        &domain_listener, callback_mask, datakind_mask);
```

`init_monitor()` throws exceptions in the following cases:

- *BadParameter* if a monitor is already created for the given DDS domain or *Fast DDS* Discovery Server network.
- *Error* if the creation of the monitor fails

4.2 Stop and restart a monitor

Warning: Restarting a stopped monitor is currently not supported. It will be implemented on a future release of *Fast DDS Statistics Backend*.

Fast DDS Statistics Backend allows for a monitorization to be stopped and restarted at any time. Stopping a monitorization merely means that the internal statistics *DataReaders* are disabled, but the already received data is still accessible to applications through the query API (see *Get statistical data*). It is important to note that:

- Calls to `stop_monitor()` on an already stopped monitor take no effect.
- Calls to `restart_monitor()` on an already running monitor take no effect.
- `stop_monitor()` must be called before calling `clear_monitor()`.
- `stop_monitor()` throws *BadParameter* if the provided monitor ID is not yet registered.

```
// Init a monitor in DDS domain 0 with no listener associated
EntityId domain_monitor_id = StatisticsBackend::init_monitor(0);
// Stop the monitor
StatisticsBackend::stop_monitor(domain_monitor_id);
// Restart the monitor
StatisticsBackend::restart_monitor(domain_monitor_id);
```

4.3 Clearing data

eProsima Fast DDS Statistics Backend monitors both the entities discovered in a certain DDS domain or *Fast DDS* Discovery Server network, and the statistic data related to these entities. *StatisticsBackend* provides several methods to clear the data contained in the internal database:

- `clear_statistics_data()` commands the deletion of all statistics data contained within the database.
- `clear_inactive_entities()` deletes from the database those *entities* that are no longer alive and communicating (see *Check whether an entity is active* for more information).

- `clear_monitor()` clears all data (entities and statistics) related to a specific monitor. To do so, the corresponding monitor needs to be stopped before the `clear_monitor()` operation can be performed.

Warning: `clear_monitor()` is currently not supported. It will be implemented on a future release of *Fast DDS Statistics Backend*.

```
// Init a monitor in DDS domain 0 with no listener associated
EntityId domain_monitor_id = StatisticsBackend::init_monitor(0);
// Clear all statistics data
StatisticsBackend::clear_statistics_data();
// Clear inactive entities
StatisticsBackend::clear_inactive_entities();
// Stop the monitor
StatisticsBackend::stop_monitor(domain_monitor_id);
// Clear all data related to the monitor
StatisticsBackend::clear_monitor(domain_monitor_id);
```

4.4 Reset Fast DDS Statistics Backend

If the user needs to restart *Fast DDS Statistics Backend* returning to the initial conditions, `reset()` is provided. Calling this method clears all the data collected since the first monitor was initialized, erases all monitors (not being available for restarting afterwards), and removes the physical listener (see *Set listeners* for more information). In order to call `reset()`, all monitors have to be stopped (inactive). Otherwise it throws *PreconditionNotMet*.

```
// Init a monitor in DDS domain 0 with no listener associated
EntityId domain_monitor_id = StatisticsBackend::init_monitor(0);
// Stop the monitor
StatisticsBackend::stop_monitor(domain_monitor_id);
// Reset Fast DDS Statistics Backend
StatisticsBackend::reset();
```

4.5 Set listeners

As explained in *Listeners*, each *Fast DDS Statistics Backend* monitor has two listeners:

- *PhysicalListener*: Registers events about changes in the physical aspects of the communication (hosts, users, processes, and locators).
- *DomainListener*: Registers events about changes in the DDS network (domain, participants, topics, data readers, and data writers).

Since the physical aspects of the communication can be shared across different DDS domains and *Fast DDS* Discovery Server networks, only one *PhysicalListener* can be set for the entire application.

Important: Even though the *PhysicalListener* can be set at any time, it is recommended to set it prior to initializing any monitoring, so that no physical events are missed.

Furthermore, it is possible to change the *DomainListener*, *CallbackMask*, and *DataKindMask* of any monitor at any time.

```
// Set a physical listener with all callbacks enabled
CustomPhysicalListener physical_listener;
StatisticsBackend::set_physical_listener(&physical_listener, CallbackMask::all());

// Init a monitor in DDS domain 0 with no listener associated
EntityId domain_monitor_id = StatisticsBackend::init_monitor(0);

// Add a domain listener to the monitor with all callbacks enabled and that does not
↳notify
// of any statistics data
CustomDomainListener domain_listener;
StatisticsBackend::set_domain_listener(
    domain_monitor_id, &domain_listener, CallbackMask::all(), DataKindMask::none());
```

`set_domain_listener()` throws *BadParameter* if the given monitor ID is not yet registered.

4.6 Get entities' graph

Warning: This feature is currently not supported. It will be implemented on a future release of *Fast DDS Statistics Backend*. Furthermore, the graph structure may change.

Fast DDS Statistics Backend allows to retrieve the entire graph of entities for which the singleton holds statistics data. The result of this query is a *Graph* tree structure that contains the *EntityId* and names of each entity. To be able to understand and interpret this tree, it is required to know about all the available entities and the inner relations between them. Following, there is a diagram of the relation between the *Fast DDS Statistics Backend* entities, and how are they divided into physical and domain related. For more information about the different *EntityKind* please refer to *EntityKind*.

Fig. 1: *Fast DDS Statistics Backend* entity relations and their division into physical and domain related.

For the following example, a simple scenario is considered, where there is one process running two participants on the same domain; one with a data reader and the other one with a data writer (both in the same topic). This means that there is only one *USER* within a single *HOST*. The application can retrieve the network graph by:

```
Graph graph = StatisticsBackend::get_graph();
```

In this example, the previous call would return a *Graph* object similar to the following:

```
{
  "hosts":
  [
    {
      "entity_id": "1",
      "name": "example_host",
      "alias": "example_host_alias",
      "alive": true,
      "users":
      [
        {
```

(continues on next page)

(continued from previous page)

```
        "entity_id": "2",
        "name": "example_user",
        "alias": "example_user_alias",
        "alive": true,
        "processes":
        [
            {
                "entity_id": "3",
                "name": "example_process_1",
                "alias": "example_process_1_alias",
                "alive": true,
                "pid": "9564",
                "participants":
                [
                    "4"
                ]
            },
            {
                "entity_id": "8",
                "name": "example_process_2",
                "alias": "example_process_2_alias",
                "alive": true,
                "pid": "9565",
                "participants":
                [
                    "9"
                ]
            }
        ]
    },
    ],
    "locators":
    [
        {
            "entity_id": "7",
            "name": "127.0.0.1:7412",
            "alias": "localhost_1",
            "alive": true,
            "datareaders":
            [
            ],
            "datawriters":
            [
                "6"
            ]
        },
        {
            "entity_id": "11",
            "name": "127.0.0.1:7414",
            "alias": "localhost_2",
```

(continues on next page)

(continued from previous page)

```

    "alive": true,
    "datareaders":
    [
        "10"
    ],
    "datawriters":
    [
    ]
}
],
"domains":
[
    {
        "entity_id": "0",
        "name": "0",
        "alias": "domain_0_alias",
        "alive": true,
        "participants":
        [
            {
                "entity_id": "4",
                "name": "participant_1",
                "alias": "participant_1_alias",
                "alive": true,
                "guid": "01.0f.22.cd.59.64.04.00.02.00.00.00|00.00.01.c1",
                "process": "3",
                "datareaders":
                [
                ],
                "datawriters":
                [
                    {
                        "entity_id": "6",
                        "name": "datawriter_1",
                        "alias": "datawriter_1_alias",
                        "alive": true,
                        "guid": "01.0f.22.cd.59.64.04.00.02.00.00.00|00.00.01.03",
                        "topic": "5",
                        "locators":
                        [
                            "7"
                        ]
                    }
                ]
            }
        ]
    },
    {
        "entity_id": "9",
        "name": "participant_2",
        "alias": "participant_2_alias",
        "alive": false,
        "guid": "01.0f.22.cd.59.64.04.00.05.00.00.00|00.00.01.c1",
        "process": "8",

```

(continues on next page)

(continued from previous page)

```

        "datareaders":
        [
            {
                "entity_id": "10",
                "name": "datareader_1",
                "alias": "datareader_1_alias",
                "alive": false,
                "guid": "01.0f.22.cd.59.64.04.00.05.00.00.00|00.00.01.04",
                "topic": "5",
                "locators":
                [
                    "11"
                ]
            }
        ],
        "datawriters":
        [
        ]
    },
    "topics":
    [
        {
            "entity_id": "5",
            "name": "example_topic",
            "alias": "example_topic_alias",
            "alive": true,
            "data_type": "example_data_type",
            "datareaders":
            [
                "10"
            ],
            "datawriters":
            [
                "6"
            ]
        }
    ]
}

```

Then, the application can extract information about the active entities from the graph as shown below:

```

// Iterate over hosts
for (const auto& host : graph["hosts"])
{
    if (host["alive"])
    {
        std::cout << "Host name: " << host["name"] << std::endl;
        std::cout << "Host alias: " << host["alias"] << std::endl;
        // Iterate over users
    }
}

```

(continues on next page)

(continued from previous page)

```

for (const auto& user : host["users"])
{
    if (user["alive"])
    {
        std::cout << "\tUser name: " << user["name"] << std::endl;
        std::cout << "\tUser alias: " << user["alias"] << std::endl;
        // Iterate over processes
        for (const auto& process : user["processes"])
        {
            if (process["alive"])
            {
                std::cout << "\t\tProcess name: " << process["name"] <<␣
↪std::endl;
                std::cout << "\t\tProcess alias: " << process["alias"] <<␣
↪std::endl;
                std::cout << "\t\tProcess PID: " << process["pid"] << std::endl;
                // Iterate over the list of participant IDs
                for (const auto& participant_id : process["participants"])
                {
                    // Look for the actual participant in the domains
                    for (const auto& domain : graph["domains"])
                    {
                        for (const auto& participant : domain["participants"])
                        {
                            // Check if the participant is the one that is being␣
↪looked for
                            if (participant["entity_id"] == participant_id &&␣
↪participant["alive"])
                            {
                                std::cout << "\t\t\tParticipant name: " <<␣
↪participant["name"] <<
                                    std::endl;
                                std::cout << "\t\t\tParticipant alias: " <<␣
↪participant["alias"] <<
                                    std::endl;
                                std::cout << "\t\t\tParticipant GUID: " <<␣
↪participant["guid"] <<
                                    std::endl;
                                // Iterate over data writers
                                for (const auto& datawriter : participant[
↪"datawriters"])
                                {
                                    if (datawriter["alive"])
                                    {
                                        std::cout << "\t\t\t\tDatawriter name: "
↪<<
                                            datawriter["name"] << std::endl;
                                        std::cout << "\t\t\t\tDatawriter alias:
↪" <<
                                            datawriter["alias"] << std::endl;
                                        std::cout << "\t\t\t\tDatawriter GUID: "
↪<<

```

(continues on next page)

(continued from previous page)

```

datawriter["guid"] << std::endl;
// Iterate over topics
for (const auto& topic : domain["topics
↳"])
{
↳that is being looked for
    if (topic["entity_id"] == datawriter[
↳"topic"])
    {
        std::cout << "\t\t\tDatawriter
↳topic name: " <<
        topic["name"] << std::endl;
        std::cout << "\t\t\tDatareader
↳topic alias: " <<
        topic["alias"] << std::endl;
        break;
    }
}
// Iterate over the list of locator IDs
for (const auto& locator_id : datawriter[
↳"locators"])
{
↳"locators")
    for (const auto& locator : graph[
↳one that is being looked for
        if (locator["entity_id"] ==
↳locator_id)
        {
            std::cout << "\t\t\t
↳tDatawriter locator name: " <<
            locator["name"] <<
↳std::endl;
            std::cout << "\t\t\t
↳tDatawriter locator alias: " <<
            locator["alias"] <<
↳std::endl;
        }
    }
}
// Iterate over data readers
for (const auto& datareader : participant[
↳"datareaders"])
{
    if (datareader["alive"])
    {
        std::cout << "\t\t\tDatareader name: "
↳<<

```

(continues on next page)

(continued from previous page)

```

}
  }
    }
      }
        }
          }
            }

```

Running the previous snippet on the given example should output:

```

Host name: "example_host"
Host alias: "example_host_alias"
  User name: "example_user"
    User name: "example_user_alias"
      Process name: "example_process_1"
        Process alias: "example_process_1_alias"
          Process PID: "9564"
            Participant name: "participant_1"
              Participant alias: "participant_1_alias"
                Participant GUID: "01.0f.22.cd.59.64.04.00.02.00.00.00|00.00.01.c1"
                  Datawriter name: "datawriter_1"
                    Datawriter alias: "datawriter_1_alias"
                      Datawriter GUID: "01.0f.22.cd.59.64.04.00.02.00.00.00|00.00.01.03"
                        Datawriter topic name: "example_topic"
                          Datawriter topic alias: "example_topic_alias"
                            Datawriter locator name: "127.0.0.1:7412"
                              Datawriter locator alias: "localhost_1"
                                Process name: "example_process_2"
                                  Process alias: "example_process_2_alias"
                                    Process PID: "9565"

```

For more information about the operations available with Graph objects, please refer to [Graph](#).

4.7 Get entity meta information

Fast DDS Statistics Backend includes the possibility of retrieving the meta information of any given entity present in the network. The returned tree always includes the basic information about the entity: kind, id, name, alias and if the entity is alive. Depending on the *EntityKind*, the returned object can contain extra information such as pid, guid, qos, locators or data_type. `get_info()` returns a *Info* object.

```

Info host_info = StatisticsBackend::get_info(host_id);
Info user_info = StatisticsBackend::get_info(user_id);
Info process_info = StatisticsBackend::get_info(process_id);
Info locator_info = StatisticsBackend::get_info(locator_id);
Info domain_info = StatisticsBackend::get_info(domain_id);
Info participant_info = StatisticsBackend::get_info(participant_id);
Info datareader_info = StatisticsBackend::get_info(datareader_id);
Info datawriter_info = StatisticsBackend::get_info(datawriter_id);
Info topic_info = StatisticsBackend::get_info(topic_id);

```

4.7.1 Host Info example

```
{
  "id": 1,
  "kind": "host",
  "name": "host_name",
  "alias": "host_alias",
  "alive": true
}
```

4.7.2 User Info example

```
{
  "id": 2,
  "kind": "user",
  "name": "user_name",
  "alias": "user_alias",
  "alive": true
}
```

4.7.3 Process Info example

```
{
  "id": 3,
  "kind": "process",
  "name": "process_name",
  "alias": "process_alias",
  "alive": true,
  "pid": "9564"
}
```

4.7.4 Locator Info example

```
{
  "id": 4,
  "kind": "locator",
  "name": "127.0.0.1:7412",
  "alias": "localhost",
  "alive": true
}
```

4.7.5 Domain Info example

```
{
  "id": 0,
  "kind": "domain",
  "name": "0",
  "alias": "domain_alias",
  "alive": true
}
```

4.7.6 Participant Info example

```
{
  "id": 5,
  "kind": "participant",
  "name": "participant_name",
  "alias": "participant_alias",
  "alive": true,
  "guid": "01.0f.22.cd.59.64.04.00.05.00.00.00|00.00.01.c1",
  "qos": {
    "available_builtin_endpoints": 3135,
    "lease_duration":
      {
        "nanoseconds": 0,
        "seconds": 3
      },
    "properties":
      [
        {
          "name": "PARTICIPANT_TYPE",
          "value": "CLIENT"
        },
        {
          "name": "DS_VERSION",
          "value": "2.0"
        }
      ],
    "user_data": "656e636c6176653d2f3b00",
    "vendor_id": "010f"
  },
  "locators":
  [
    "127.0.0.1:1234"
  ]
}
```

4.7.7 DataReader Info example

```

{
  "id": 6,
  "kind": "datareader",
  "name": "datareader_name",
  "alias": "datareader_alias",
  "alive": false,
  "guid": "01.0f.22.cd.59.64.04.00.05.00.00.00|00.00.01.04",
  "qos":
  {
    "data_sharing":
    {
      "domain_ids":
      [
        0
      ],
      "kind": "AUTO",
      "max_domains": 1,
      "shm_directory": "/dev/shm"
    },
    "deadline":
    {
      "period":
      {
        "nanoseconds": 50,
        "seconds": 10
      }
    },
    "destination_order":
    {
      "kind": "BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS"
    },
    "disable_positive_acks":
    {
      "duration":
      {
        "nanoseconds": 100,
        "seconds": 0
      },
      "enabled": true
    },
    "durability":
    {
      "kind": "VOLATILE_DURABILITY_QOS"
    },
    "durability_service":
    {
      "history_depth": 1,
      "history_kind": "KEEP_LAST_HISTORY_QOS",
      "max_instances": 30,
      "max_samples": 3000,
      "max_samples_per_instance": 100,

```

(continues on next page)

(continued from previous page)

```
    "service_cleanup_delay":
    {
        "nanoseconds": 0,
        "seconds": 5
    }
},
"group_data": "9d46781410ff",
"latency_budget":
{
    "duration":
    {
        "nanoseconds": 50,
        "seconds": 10
    }
},
"lifespan":
{
    "duration":
    {
        "nanoseconds": 0,
        "seconds": 10000
    }
},
"liveliness":
{
    "announcement_period":
    {
        "nanoseconds": 0,
        "seconds": 3
    },
    "lease_duration":
    {
        "nanoseconds": 0,
        "seconds": 10
    },
    "kind": "AUTOMATIC_LIVELINESS_QOS"
},
"ownership":
{
    "kind": "SHARED_OWNERSHIP_QOS"
},
"partition":
[
    "partition_1",
    "partition_2"
],
"presentation":
{
    "access_scope": "INSTANCE_PRESENTATION_QOS",
    "coherent_access": false,
    "ordered_access": false
},
},
```

(continues on next page)

(continued from previous page)

```

"reliability":
{
  "kind": "RELIABLE_RELIABILITY_QOS",
  "max_blocking_time":
  {
    "nanoseconds": 0,
    "seconds": 3
  }
},
"representation":
[
],
"time_based_filter":
{
  "minimum_separation":
  {
    "seconds": 12,
    "nanoseconds": 0
  }
},
"topic_data": "5b33419a",
"type_consistency":
{
  "force_type_validation": false,
  "ignore_member_names": false,
  "ignore_sequence_bounds": true,
  "ignore_string_bounds": true,
  "kind": "DISALLOW_TYPE_COERCION",
  "prevent_type_widening": false
},
"user_data": "ff00"
}
}

```

4.7.8 DataWriter Info example

```

{
  "id": 7,
  "kind": "datawriter",
  "name": "datawriter_name",
  "alias": "datawriter_alias",
  "alive": true,
  "guid": "01.0f.22.cd.59.64.04.00.02.00.00.00|00.00.01.03",
  "qos":
  {
    "data_sharing":
    {
      "domain_ids":
      [
        0
      ]
    }
  }
}

```

(continues on next page)

(continued from previous page)

```
    ],
    "kind": "AUTO",
    "max_domains": 1,
    "shm_directory": "/dev/shm"
  },
  "deadline":
  {
    "period":
    {
      "nanoseconds": 50,
      "seconds": 10
    }
  },
  "destination_order":
  {
    "kind": "BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS"
  },
  "disable_positive_acks":
  {
    "duration":
    {
      "nanoseconds": 100,
      "seconds": 0
    },
    "enabled": true
  },
  "durability":
  {
    "kind": "VOLATILE_DURABILITY_QOS"
  },
  "durability_service":
  {
    "history_depth": 1,
    "history_kind": "KEEP_LAST_HISTORY_QOS",
    "max_instances": 30,
    "max_samples": 3000,
    "max_samples_per_instance": 100,
    "service_cleanup_delay":
    {
      "nanoseconds": 0,
      "seconds": 5
    }
  },
  "group_data": "9d46781410ff",
  "latency_budget":
  {
    "duration":
    {
      "nanoseconds": 50,
      "seconds": 10
    }
  },
},
```

(continues on next page)

(continued from previous page)

```

"lifespan":
{
  "duration":
  {
    "nanoseconds": 0,
    "seconds": 10000
  }
},
"liveliness":
{
  "announcement_period":
  {
    "nanoseconds": 0,
    "seconds": 3
  },
  "lease_duration":
  {
    "nanoseconds": 0,
    "seconds": 10
  },
  "kind": "AUTOMATIC_LIVELINESS_QOS"
},
"ownership":
{
  "kind": "SHARED_OWNERSHIP_QOS"
},
"ownership_strength":
{
  "value": 5
},
"partition":
[
  "partition_1",
  "partition_2"
],
"presentation":
{
  "access_scope": "INSTANCE_PRESENTATION_QOS",
  "coherent_access": false,
  "ordered_access": false
},
"publish_mode":
{
  "kind": "ASYNCHRONOUS_PUBLISH_MODE"
},
"reliability":
{
  "kind": "RELIABLE_RELIABILITY_QOS",
  "max_blocking_time":
  {
    "nanoseconds": 0,
    "seconds": 3
  }
}

```

(continues on next page)

(continued from previous page)

```

    }
  },
  "representation":
  [
  ],
  "time_based_filter":
  {
    "minimum_separation":
    {
      "seconds": 12,
      "nanoseconds": 0
    }
  },
  "topic_data": "5b33419a",
  "user_data": "ff00"
}
}

```

4.7.9 Topic Info example

```

{
  "id": 8,
  "kind": "topic",
  "name": "topic_name",
  "alias": "topic_alias",
  "alive": true,
  "data_type": "example_data_type"
}

```

4.8 Get entities of a given kind

The *StatisticsBackend* singleton can be queried about all the entities of a given *EntityKind*. For example, *get_entities()* function can be used to retrieve all the *HOST* for which statistics are reported.

```

// Get all hosts
std::vector<EntityId> hosts = StatisticsBackend::get_entities(EntityKind::HOST);
for (EntityId host : hosts)
{
  std::cout << "Host ID: " << host << std::endl;
}

```

This call to *get_entities()* is the same as:

```

StatisticsBackend::get_entities(EntityKind::HOST, EntityId::all());

```

4.8.1 Get entities of a given kind related to another entity

The *StatisticsBackend* singleton can be queried about all the entities of a given *EntityKind* that are related to any entity. For example, *get_entities()* function can be used to retrieve all the *PARTICIPANT* running on a given *HOST*.

```
// Get all participants running in a host
std::vector<EntityId> participants = StatisticsBackend::get_
  entities(EntityKind::PARTICIPANT, host_id);
for (EntityId participant : participants)
{
    std::cout << "Participant ID: " << participant << std::endl;
}
```

get_entities() throws *BadParameter* in the following cases:

- if the *EntityKind* is *INVALID*
- if the *EntityId* does not reference a entity contained in the database or is not *EntityId::all()*.
- if the *EntityKind* of the *EntityId* is *INVALID*

This function returns the related entities according to the following table:

Table 1: Entity relations

EntityId EntityKind	Host	User	Process	Domain	Topic	Do- main- Partici- pant	DataWriter	DataReader	Locator
Host	Itself	Contains	Sub-contains	By Do- mainPar- ticipant	By Do- mainPar- ticipant	Sub-contains	Sub-contains	Sub-contains	Sub-contains
User	Con- tained	Itself	Contains	By Do- mainPar- ticipant	By Do- mainPar- ticipant	Sub-contains	Sub-contains	Sub-contains	By End- points
Process	Sub- contained	Con- tained	Itself	By Do- mainPar- ticipant	By Do- mainPar- ticipant	Contains	Sub-contains	Sub-contains	By End- points
Domain	By Do- mainPar- ticipant	By Do- mainPar- ticipant	By Do- mainPar- ticipant	Itself	Contains	Contains	Sub-contains	Sub-contains	By End- points
Topic	By Do- mainPar- ticipant	By Do- mainPar- ticipant	By Do- mainPar- ticipant	Con- tained	Itself	By End- points	Con- tains	Con- tains	By End- points
Domain- Partici- pant	Sub- contained	Sub- contained	Con- tained	Con- tained	By End- points	Itself	Con- tains	Con- tains	By End- points
DataWriter	Sub- contained	Sub- contained	Sub- contained	Sub- contained	Con- tained	Con- tained	Itself	By topic	Con- tains
DataReader	Sub- contained	Sub- contained	Sub- contained	Sub- contained	Con- tained	Con- tained	By topic	Itself	Con- tains
Locator	Sub- contained	By End- points	By End- points	By End- points	By End- points	By End- points	Con- tained	Con- tained	Itself

- **Itself**: Means that the return will only contain the entity by which the query is performed, i.e. when asking for all the *HOST* related to a given *HOST*, the return will simply be the *HOST* itself.

- **Contains:** The returned entities will be the ones that the entity by which the query is performed contains, i.e. when asking for all the *PARTICIPANT* related to a *PROCESS*, the return will be all the *PARTICIPANT* that the *PROCESS* contains.
- **Sub-contains:** The returned entities will be the ones that the entity by which the query is performed sub-contains, i.e. when asking for all the *DATAWRITER* related to a *USER*, the return will be all the *DATAWRITER* that are contained in each of the *PARTICIPANT* in each of the *PROCESS* that the *USER* contains.
- **Contained:** The returned entity will be that one in which the entity by which the query is performed is contained, i.e. when asking for all the *TOPIC* related to a *DATAREADER*, the return will be the *TOPIC* in which the *DATAREADER* is contained.
- **Sub-contained:** The returned entity will be the one in which the entity by which the query is performed is sub-contained, i.e. when asking for all the *HOST* related to a *PARTICIPANT*, the return will be the *HOST* in which the *PARTICIPANT* is sub-contained.
- **By DomainParticipant:** The returned entities will be the ones that are related to the entity by which the query is performed through the DomainParticipant, i.e. when asking for all the *HOST* related to a *DOMAIN*, the result will be all the *HOST* that have a *PARTICIPANT* running on said *DOMAIN*.
- **By Endpoints:** The returned entities will be the ones that are related to the entity by which the query is performed through the endpoints (*DATAREADER* and *DATAWRITER*), i.e. when asking for all the *LOCATOR* related to a *TOPIC*, the result will be all the *LOCATOR* that are used by all the *DATAREADER* and *DATAWRITER* present in the *TOPIC*.

4.9 Get statistical data

Fast DDS Statistics Backend provides two overloads of `get_data()` to retrieve statistical data of a given *DataKind* within a time frame (for more information about all the reported *DataKind*, please refer to *DataKind*). This time interval is evenly divided into the specified number of bins, each one with size $(t_{to} - t_{from}) / (ofbins)$. For each of these bins, a new *StatisticsData* value is calculated applying the given *StatisticKind* to all the data points in it. The result is a collection of *StatisticsData* elements with size equal to the number of specified bins.

Important: If the number of bins is set to zero, then all data points are returned and no statistic is calculated for the series.

Depending on the *DataKind*, the data is related to one or two entities, e.g. *FASTDDS_LATENCY* measures the latency between a write operation on the data writer side and the notification to the user when the data is available on reader side, whereas *HEARTBEAT_COUNT* contains the amount of sent HEARTBEATS. Because of this difference, `get_data()` can take either one or two *EntityId* related to the *DataKind* in question. The following table illustrates the expected inputs depending on the query's *DataKind* passed to `get_data()`:

<i>DataKind</i>	Source collection <i>EntityKind</i>	Target collection <i>EntityKind</i>
<i>FASTDDS_LATENCY</i>	<i>DATAWRITER</i>	<i>DATAREADER</i>
<i>NETWORK_LATENCY</i>	<i>PARTICIPANT</i>	<i>LOCATOR</i>
<i>PUBLICATION_THROUGHPUT</i>	<i>DATAWRITER</i>	Not applicable
<i>SUBSCRIPTION_THROUGHPUT</i>	<i>DATAREADER</i>	Not applicable
<i>RTPS_PACKETS_SENT</i>	<i>PARTICIPANT</i>	<i>LOCATOR</i>
<i>RTPS_BYTES_SENT</i>	<i>PARTICIPANT</i>	<i>LOCATOR</i>
<i>RTPS_PACKETS_LOST</i>	<i>PARTICIPANT</i>	<i>LOCATOR</i>
<i>RTPS_BYTES_LOST</i>	<i>PARTICIPANT</i>	<i>LOCATOR</i>
<i>RESENT_DATA</i>	<i>DATAWRITER</i>	Not applicable
<i>HEARTBEAT_COUNT</i>	<i>DATAWRITER</i>	Not applicable
<i>ACKNACK_COUNT</i>	<i>DATAREADER</i>	Not applicable
<i>NACKFRAG_COUNT</i>	<i>DATAREADER</i>	Not applicable
<i>GAP_COUNT</i>	<i>DATAWRITER</i>	Not applicable
<i>DATA_COUNT</i>	<i>DATAWRITER</i>	Not applicable
<i>PDP_PACKETS</i>	<i>PARTICIPANT</i>	Not applicable
<i>EDP_PACKETS</i>	<i>PARTICIPANT</i>	Not applicable
<i>DISCOVERY_TIME</i>	<i>PARTICIPANT</i>	Not applicable
<i>SAMPLE_DATAS</i>	<i>DATAWRITER</i>	Not applicable

`get_data()` throws *BadParameter* if the calling parameters are not consistent.

`get_data_supported_entity_kinds()` can be used to get all the *EntityKind* pairs suitable for a given *DataKind*, according to this table.

- For a *DataKind* that only relates to one Entity, the first element of the pair is the *EntityKind* of such Entity, while the second element is *INVALID*.
- For a *DataKind* that relates to two Entities, the first element of the pair is the *EntityKind* of the source Entity, while the second element is the *EntityKind* of the target Entity.

The source and target pairs returned by this method are the source and target *EntityKind* accepted by `get_data()` for the given *DataKind*. This is convenient to prepare a call to `get_data()` from an *EntityKind*. First, call `get_data_supported_entity_kinds()` with the *DataKind* to get the *EntityKind* of the related entities. Then, call `get_entities()` to get the available entities of that kind. Finally, call `get_data()` with the pairs that `get_entities()` returns.

```

/* Get all the EntityKind pairs related to DISCOVERED_ENTITY. */
std::vector<std::pair<EntityKind, EntityKind>> types_list =
    StatisticsBackend::get_data_supported_entity_kinds(DataKind::DISCOVERY_TIME);

/* Iterate over all the valid pairs composing the final result */
std::vector<StatisticsData> discovery_times;
for (std::pair<EntityKind, EntityKind> type_pair : types_list)
{
    /* Take the data for this pair and append it to the existing data */
    std::vector<StatisticsData> tmp = StatisticsBackend::get_data(
        DataKind::DISCOVERY_TIME,
        StatisticsBackend::get_entities(type_pair.first, host1_id),
        StatisticsBackend::get_entities(type_pair.second, host2_id));

    discovery_times.insert(discovery_times.end(), tmp.begin(), tmp.end());
}

```

Warning: If for a given bin, the *Fast DDS Statistics Backend* has no data, the value returned will be the one supplied by `std::numeric_limits<double>::quiet_NaN`.

4.9.1 Examples

Following, some example queries are provided to serve as inspiration for applications using *Fast DDS Statistics Backend*.

DataWriter's Fast DDS Latency median example

```

/* Get the DataReaders related to a given DataWriter */
std::vector<EntityId> datareaders = StatisticsBackend::get_
↳entities(EntityKind::DATAREADER, datawriter_id);

/* Get the current time */
std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

/*
 * Get the median of the FASTDDS_LATENCY of the last 10 minutes, divided into ten bins,
 * between a given DataWriter and its related DataReaders. After the operation,
 * latency_data.size() is 10. Each of the elements of latency_data is a StatisticsData
 * element which represents the median of the FASTDDS_LATENCY of that minute.
 */
std::vector<StatisticsData> latency_data = StatisticsBackend::get_data(
    DataKind::FASTDDS_LATENCY,           // DataKind
    std::vector<EntityId>({datawriter_id}), // Source entities
    datareaders,                         // Target entities
    10,                                  // Number of bins
    now - std::chrono::minutes(10),      // t_from
    now,                                  // t_to
    StatisticKind::MEDIAN);              // Statistic

```

Topic's Fast DDS Latency mean example

```

/* Get the DataWriters and DataReaders in a Topic */
std::vector<EntityId> topic_datawriters = StatisticsBackend::get_
↳entities(EntityKind::DATAWRITER, topic_id);
std::vector<EntityId> topic_datareaders = StatisticsBackend::get_
↳entities(EntityKind::DATAREADER, topic_id);

/* Get the current time */
std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

/*
 * Get the median of the FASTDDS_LATENCY of the last 10 minutes, divided into ten bins,
 * between the DataWriters of Host 1 and the DataReaders of Host 2. After the operation,
 * latency_data.size() is 10. Each of the elements of latency_data is a StatisticsData
 * element which represents the median of the FASTDDS_LATENCY of that minute.
 */

```

(continues on next page)

(continued from previous page)

```

std::vector<StatisticsData> latency_data = StatisticsBackend::get_data(
    DataKind::FASTDDS_LATENCY,           // DataKind
    topic_datawriters,                   // Source entities
    topic_datareaders,                   // Target entities
    10,                                   // Number of bins
    now - std::chrono::minutes(10),      // t_from
    now,                                  // t_to
    StatisticKind::MEAN);                 // Statistic

```

Topic's Heartbeat count maximum example

```

std::vector<EntityId> participant_datawriters = StatisticsBackend::get_
↳entities(EntityKind::DATAWRITER,
           participant_id);

/* Get the current time */
std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

/*
 * Get the maximum of the HEARTBEAT_COUNT of the last 10 minutes, divided into ten bins,
 * of the DataWriters of a given Participant. After the operation, heartbeat_data.size()
↳is
 * 10. Each of the elements of heartbeat_data is a StatisticsData element which
↳represents
 * the maximum of the HEARTBEAT_COUNT of that minute.
 */
std::vector<StatisticsData> heartbeat_data = StatisticsBackend::get_data(
    DataKind::HEARTBEAT_COUNT,           // DataKind
    participant_datawriters,             // Source entities
    10,                                   // Number of bins
    now - std::chrono::minutes(10),      // t_from
    now,                                  // t_to
    StatisticKind::MAX);                 // Statistic

```

Host to Host Fast DDS Latency all points example

It is also possible to retrieve all the data points of a given *DataKind* within the time frame. This is done by setting the number of bins to 0. In this case, the *StatisticKind* is ignored so it can be left to its default value.

```

std::vector<EntityId> host1_datawriters = StatisticsBackend::get_
↳entities(EntityKind::DATAWRITER, host1_id);
std::vector<EntityId> host2_datareaders = StatisticsBackend::get_
↳entities(EntityKind::DATAREADER, host2_id);

/* Get the current time */
std::chrono::system_clock::time_point now = std::chrono::system_clock::now();

/*
 * Get all the FASTDDS_LATENCY data points of the last 10 minutes between the DataWriters

```

(continues on next page)

(continued from previous page)

```

* of Host 1 and the DataReaders of Host 2. data.size() == total number of data points
* received. Since bins is 0, the statistic is left as default.
*/
std::vector<StatisticsData> data = StatisticsBackend::get_data(
    DataKind::FASTDDS_LATENCY,           // DataKind
    host1_datawriters,                   // Source entities
    host2_datareaders,                   // Target entities
    0,                                    // Number of bins
    now - std::chrono::minutes(10),      // t_from
    now);                                 // t_to

```

For more information about the available *DataKind* and *StatisticKind* please refer to *DataKind* and *StatisticKind* respectively.

4.10 Get entity type

It is also possible to retrieve the *EntityKind* of an entity given its *EntityId*:

```
EntityKind kind = StatisticsBackend::get_type(entity_id);
```

get_type() throws *BadParameter* if there is no entity with the given ID.

4.11 Set entity alias

Even though the *Fast DDS Statistics Backend* provides a name for each entity, this default name can be lengthy and not really self-explanatory and user-friendly. Therefore, *set_alias()* allows the user to apply any alias to the desired entity so it can be easily identified. If the entity provided does not exist *set_alias()* throws *BadParameter*.

```
StatisticsBackend::set_alias(entity_id, "my_alias");
```

4.12 Check whether an entity is active

Unless *clear_monitor()* is called for a monitor, *Fast DDS Statistics Backend* keeps the statistical data record of all the entities that have at some point been detected by a monitor. However, it is possible that some of these entities have already abandoned the network, thus becoming inactive. For this reason, *StatisticsBackend* exposes a *is_active()* function that returns whether an entity is active, given its *EntityId*.

```
bool active = StatisticsBackend::is_active(entity_id);
```


4.13 Check whether an entity is builtin

eProsima Fast DDS Statistics Backend discovers any DDS entity in the monitored domain or *Fast DDS* Discovery Server network, including the builtin entities used to exchange metatraffic data that allows mutual discovery. `is_metatraffic()` returns whether the entity is related to these builtin entities or not. The possible DDS builtin entities are always of *TOPIC*, *DATAWRITER*, or *DATAREADER* kind (refer to *EntityKind* for more information). This function allows the user to discriminate between the topics and endpoints exchanging metatraffic data and those that are exchanging user data.

```
bool metatraffic = StatisticsBackend::is_metatraffic(entity_id);
```

4.14 Saving and restoring the statistics data

Fast DDS Statistics Backend allows to dump the contents of the database to the file system. This may be used as a backup procedure, or as a means of analyzing the data offline later. It is also possible to load a dump previously saved, which allows for this analysis to be done with any front-end that communicates with the *Fast DDS Statistics Backend*.

- Use `dump_database()` to save the content of the Backend's database to a file.
- Use `load_database()` to loaded a saved database to the Backend.

For information about the format of the dumped data, please, refer to *Database dumps*.

Warning: Loading a saved database can only be done on an empty Backend. This means that no monitors were initialized since the Backend started, or that the Backend has been reset using `reset()`. If `load_database()` is used on a non-empty Backend, an exception will be issued.

The following snippet shows how to dump the current database contents to a file, and then load another data set that was saved previously, resetting the Backend in between.

```
// Save the database to a file
StatisticsBackend::dump_database("new_backend_dump.json", false);

// Reset the Backend to empty the current database contents
StatisticsBackend::reset();

// Load an old backup to the emptied Backend
StatisticsBackend::load_database("old_backend_dump.json");
```

The bool parameter of `dump_database()` indicates if the statistics data of all entities must be cleared after the dump.

```
// Save the database to a file, cleaning the statistics data
StatisticsBackend::dump_database("new_backend_dump.json", true);
```


5.1 DataKind

The *eProxima Fast DDS Statistics Backend* records statistics data of different nature, as provided by *eProxima Fast DDS Statistics Module*, e.g., latency or message count. We refer to the nature of these data values as their *DataKind*.

- *FASTDDS_LATENCY*: The latency between a write operation in the data writer and the moment the data is available in the data reader.
- *NETWORK_LATENCY*: The latency in the communication between two locators.
- *PUBLICATION_THROUGHPUT*: Amount of data (in Mb/s) sent by a data writer.
- *SUBSCRIPTION_THROUGHPUT*: Amount of data (in Mb/s) received by a data reader.
- *RTPS_PACKETS_SENT*: Amount of packets sent from a participant to a locator.
- *RTPS_BYTES_SENT*: Amount of bytes sent from a participant to a locator.
- *RTPS_PACKETS_LOST*: Amount of packets lost from a participant to a locator.
- *RTPS_BYTES_LOST*: Amount of bytes lost from a participant to a locator.
- *RESENT_DATA*: Amount of DATA/DATAFRAG sub-messages that had to be resent from a data writer.
- *HEARTBEAT_COUNT*: Amount of HEARTBEATs that a data writer sends.
- *ACKNACK_COUNT*: Amount of ACKNACKs that a data reader sends.
- *NACKFRAG_COUNT*: Amount of NACKFRAGs that a data reader sends.
- *GAP_COUNT*: Amount of GAPS that a data writer sends.
- *DATA_COUNT*: Amount of DATA/DATAFRAGs that a data writer sends.
- *PDP_PACKETS*: Amount of PDP packets sent by a participant.
- *EDP_PACKETS*: Amount of EDP packets sent by a participant.
- *DISCOVERY_TIME*: Time when a participant discovers another DDS entity.
- *SAMPLE_DATAS*: Amount of DATA/DATAFRAGs needed to send a single sample.

Each data kind may relate to one or two *entities* where they are measured. For example, a *FASTDDS_LATENCY* is always measured between a data data writer and a data reader, whereas *PDP_PACKETS* is always measured in a participant, with no other entity involved in the measurement. The following table describes which entity kinds are involved in the measurement of each data kind:

Signature	Source Entity	Target Entity
<i>FASTDDS_LATENCY</i>	DataWriter	DataReader
<i>NETWORK_LATENCY</i>	Locator	Locator
<i>PUBLICATION_THROUGHPUT</i>	DataWriter	•
<i>SUBSCRIPTION_THROUGHPUT</i>	DataReader	•
<i>RTPS_PACKETS_SENT</i>	DomainParticipant	Locator
<i>RTPS_BYTES_SENT</i>	DomainParticipant	Locator
<i>RTPS_PACKETS_LOST</i>	DomainParticipant	Locator
<i>RTPS_BYTES_LOST</i>	DomainParticipant	Locator
<i>RESENT_DATA</i>	DataWriter	•
<i>HEARTBEAT_COUNT</i>	DataWriter	•
<i>ACKNACK_COUNT</i>	DataReader	•
<i>NACKFRAG_COUNT</i>	DataReader	•
<i>GAP_COUNT</i>	DataWriter	•
<i>DATA_COUNT</i>	DataWriter	•
<i>PDP_PACKETS</i>	DomainParticipant	•
<i>EDP_PACKETS</i>	DomainParticipant	•
<i>DISCOVERY_TIME</i>	DomainParticipant	DDSEntity
<i>SAMPLE_DATAS</i>	DataWriter	•

5.2 StatisticKind

`get_data()` allows for retrieving data from the *eProsima Fast DDS Statistics Backend* specifying the kind of statistic we want to receive in the output. The available statistics are:

- *MEAN*: Numerical mean of values in the set.
- *STANDARD_DEVIATION*: Standard Deviation of the values in the set.
- *MAX*: Maximum value in the set.
- *MIN*: Minimum value in the set.
- *MEDIAN*: Median value of the set.
- *COUNT*: Amount of values in the set.
- *SUM*: Summation of the values in the set.

- *NONE*: Non accumulative kind. It chooses a single data point among those in the set.

5.3 EntityKind

The *eProsima Fast DDS Statistics Backend* keeps track of the entities discovered in the DDS layout. The following list shows the different entities that are tracked:

- *HOST*: The host or machine where a participant is allocated.
- *USER*: The user that has executed a participant.
- *PROCESS*: The process where the participant is running.
- *DOMAIN*: Abstract DDS network by Domain or by Discovery Server.
- *TOPIC*: DDS Topic.
- *PARTICIPANT*: DDS Domain Participant.
- *DATAWRITER*: DDS DataWriter.
- *DATAREADER*: DDS DataReader.
- *LOCATOR*: Physical locator that a communication is using.

5.4 EntityId

When monitoring a domain (see *Initialize a monitor*), *Fast DDS Statistics Backend* labels all the different discovered entities with an *EntityId* identifier that is unique in the context of the *StatisticsBackend* instance. This *EntityId* is used by the application, among other things, to query statistical data to the backend (see *Get statistical data*). To ease the use of the *Fast DDS Statistics Backend* API, *EntityId* exposes certain commonly used operations:

5.4.1 EntityId wildcard

EntityId allows for retrieving an ID that represents all the *EntityIds*:

```
EntityId all = EntityId::all();
```

5.4.2 Invalid EntityId

EntityId allows for retrieving an invalid ID:

```
EntityId invalid = EntityId::invalid();
```

5.4.3 Invalidate an EntityId

It is also possible to invalidate an *EntityId*:

```
EntityId entity_id;  
entity_id.invalidate();
```

5.4.4 Check validity of an EntityId

It can be checked whether an *EntityId* is valid:

```
EntityId entity_id;  
bool check = entity_id.is_valid();
```

5.4.5 Check EntityId represents all Entities

It can be checked whether an *EntityId* represents all the *EntityIds*:

```
EntityId entity_id;  
bool check = entity_id.is_all();
```

5.4.6 Check validity and uniqueness of an EntityId

It can be checked whether an *EntityId* is valid and unique:

```
EntityId entity_id;  
bool check = entity_id.is_valid_and_unique();
```

5.4.7 Comparison operations

EntityIds can be compared between them:

```
EntityId entity_id_1;  
EntityId entity_id_2;  
bool check = entity_id_1 < entity_id_2;  
check = entity_id_1 <= entity_id_2;  
check = entity_id_1 > entity_id_2;  
check = entity_id_1 >= entity_id_2;  
check = entity_id_1 == entity_id_2;  
check = entity_id_1 != entity_id_2;
```

5.4.8 Output to OStream

EntityIds can be output to `std::ostream`:

```
EntityId entity_id;
std::cout << "EntityId: " << entity_id << std::endl;
```

5.5 JSON Tags

The *StatisticsBackend* uses JSON format to retrieve information in many methods as *get_info()*, *get_graph()* or *dump_database()*.

5.5.1 Dump Tags Example

The following snippet shows an example of a database dump, result of calling *dump_database()* in a database with one entity of each *EntityKind*, and one data of each *DataKind*:

```
{
  "description": "DB dump with 1 entity of each EntityKind and 1 data of each DataKind
↪",
  "version": "0.0",
  "hosts":
  {
    "1":
    {
      "name": "host_0",
      "users": ["2"]
    }
  },
  "users":
  {
    "2":
    {
      "name": "user_0",
      "host": "1",
      "processes": ["3"]
    }
  },
  "processes":
  {
    "3":
    {
      "name": "process_0",
      "pid": "36000",
      "user": "2",
      "participants": ["6"]
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

    }
  },
  "domains":
  {
    "4":
    {
      "name": "domain_0",
      "participants": ["6"],
      "topics": ["5"]
    }
  },
  "topics":
  {
    "5":
    {
      "name": "topic_0",
      "data_type": "data_type",
      "domain": "4",
      "datawriters": ["7"],
      "datareaders": ["8"]
    }
  },
  "participants":
  {
    "6":
    {
      "name": "participant_0",
      "guid": "01.0f.00.00.00.00.00.00.00.00|00.00.00.00",
      "qos": {"qos": "empty"},

      "process": "3",
      "domain": "4",
      "datawriters": ["7"],
      "datareaders": ["8"],

      "data":
      {
        "discovery_time":
        {
          "6":
          [
            {
              "src_time": "1",
              "time": "0",
              "remote_id": "6",
              "discovered": true
            }
          ]
        }
      }
    }
  },

```

(continues on next page)

(continued from previous page)

```
"pdp_packets":
[
  {
    "src_time": "0",
    "count": 2
  }
],
"edp_packets":
[
  {
    "src_time": "0",
    "count": 2
  }
],
"rtps_packets_sent":
{
  "0":
  [
    {
      "src_time": "0",
      "count": 2
    }
  ]
},
"rtps_bytes_sent":
{
  "0":
  [
    {
      "src_time": "0",
      "magnitude": 0,
      "count": 2
    }
  ]
},
"rtps_packets_lost":
{
  "0":
  [
    {
      "src_time": "0",
      "count": 2
    }
  ]
},
"rtps_bytes_lost":
{
  "0":
  [
    {
      "src_time": "0",
```

(continues on next page)

(continued from previous page)

```

        "magnitude": 0,
        "count": 2
      }
    ]
  },
  "last_reported_edp_packets":{
    "count":2,
    "src_time":"0"
  },
  "last_reported_pdp_packets":{
    "count":2,
    "src_time":"0"
  },
  "last_reported_rtps_bytes_lost":{
    "0":{
      "count":2,
      "magnitude":0,
      "src_time":"0"
    }
  },
  "last_reported_rtps_bytes_sent":{
    "0":{
      "count":2,
      "magnitude":0,
      "src_time":"0"
    }
  },
  "last_reported_rtps_packets_lost":{
    "0":{
      "count":2,
      "src_time":"0"
    }
  },
  "last_reported_rtps_packets_sent":{
    "0":{
      "count":2,
      "src_time":"0"
    }
  }
}
},
"datawriters":
{
  "7":
  {
    "name": "datawriter_0",
    "guid": "01.0f.00.00.00.00.00.00.00.00|00.00.00.00",
    "qos": {"qos": "empty"},

```

(continues on next page)

(continued from previous page)

```
"participant": "6",
"topic": "5",
"locators": ["0"],

"data":
{
  "publication_throughput":
  [
    {
      "src_time": "0",
      "data": 1.1
    }
  ],
  "resent_datas":
  [
    {
      "src_time": "0",
      "count": 2
    }
  ],
  "heartbeat_count":
  [
    {
      "src_time": "0",
      "count": 2
    }
  ],
  "gap_count":
  [
    {
      "src_time": "0",
      "count": 2
    }
  ],
  "data_count":
  [
    {
      "src_time": "0",
      "count": 2
    }
  ],
  "samples_datas":
  {
    "3":
    [
      {
        "src_time": "0",
        "count": 2
      }
    ]
  },
  "history2history_latency":
```

(continues on next page)

(continued from previous page)

```

    {
      "8":
      [
        {
          "src_time": "0",
          "data": 1.1
        }
      ]
    },

    "last_reported_data_count":{
      "count":2,
      "src_time":"0"
    },
    "last_reported_gap_count":{
      "count":2,
      "src_time":"0"
    },
    "last_reported_heartbeat_count":{
      "count":2,
      "src_time":"0"
    },
    "last_reported_resent_datas":{
      "count":2,
      "src_time":"0"
    }
  }
},

"datareaders":
{
  "8":
  {
    "name": "datareader_0",
    "guid": "01.0f.00.00.00.00.00.00.00.00.00|00.00.00.00",
    "qos": {"qos": "empty"},

    "participant": "6",
    "topic": "5",
    "locators": ["0"],

    "data":
    {
      "subscription_throughput":
      [
        {
          "src_time": "0",
          "data": 1.1
        }
      ],
      "acknack_count":

```

(continues on next page)

(continued from previous page)

```

    [
      {
        "src_time": "0",
        "count": 2
      }
    ],
    "nackfrag_count":
    [
      {
        "src_time": "0",
        "count": 2
      }
    ],
    "last_reported_acknack_count":{
      "count":2,
      "src_time":"0"
    },
    "last_reported_nackfrag_count":{
      "count":2,
      "src_time":"0"
    }
  }
},
"locators":
{
  "0":
  {
    "name": "locator_0",
    "datawriters": ["7"],
    "datareaders": ["8"],

    "data":
    {
      "network_latency_per_locator":
      {
        "0":
        [
          {
            "src_time": "0",
            "data": 1.1
          }
        ]
      }
    }
  }
}
}
}

```

5.6 Database dumps

LISTENERS

Listeners allow users to define actions that will be taken in response to changes in the monitored elements, e.g., when the deployment layout changes or when new statistical data has been received.

There are two kinds of listeners:

- *DomainListener* acts upon changes in the DDS entities of the deployment or new statistical data arrives.
- *PhysicalListener* acts upon changes in the physical entities of the deployment.

6.1 DomainListener

DomainListener is an abstract class defining the callbacks that will be triggered in response to changes in the DDS network (discovery of domain, participants, topics, data readers, data writers, and arrival of new statistics data). By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

DomainListener defines the following callbacks:

- *on_data_available()*: New statistics data has been received by the backend. The arguments in the callback specifies the kind of the received data and the entity to which this data refers.
- *on_topic_discovery()*: A new topic has been discovered in the monitored domain, or an already known topic has been updated with a new QoS value. The topics are never *undiscovered*. The arguments in the callback specifies the ID of the topic and the domain to which it belongs.
- *on_participant_discovery()*: A new participant has been discovered in the monitored domain, or an already known participant has been updated with a new Quality of Service (QoS) value, or an already known participant has been removed from the network. The arguments in the callback specifies the ID of the participant and the domain to which it belongs.
- *on_datareader_discovery()*: A new data reader has been discovered in the monitored domain, or an already known data reader has been updated with a new QoS value, or an already known data reader has been removed from the network. The arguments in the callback specifies the ID of the data reader and the domain to which it belongs.
- *on_datawriter_discovery()*: A new data writer has been discovered in the monitored domain, or an already known data writer has been updated with a new QoS value, or an already known data writer has been removed from the network. The arguments in the callback specifies the ID of the data writer and the domain to which it belongs.

6.2 PhysicalListener

PhysicalListener is an abstract class defining the callbacks that will be triggered in response to changes in the physical aspects of the communication (hosts, users, processes, and locators) By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

PhysicalListener defines the following callbacks:

- *on_host_discovery()*: A new host has been discovered in the monitored network. Hosts are never *undiscovered*. The arguments in the callback specifies the ID of the participant that discovered the host.
- *on_user_discovery()*: A new user has been discovered in the monitored network. Users are never *undiscovered*. The arguments in the callback specifies the ID of the participant that discovered the user.
- *on_process_discovery()*: A new process has been discovered in the monitored network. Processes are never *undiscovered*. The arguments in the callback specifies the ID of the participant that discovered the process.
- *on_locator_discovery()*: A new locator has been discovered in the monitored network. Locators are never *undiscovered*. The arguments in the callback specifies the ID of the participant that discovered the locator.

FULL EXAMPLE

7.1 Next steps

You may find this example at the *eProxima Fast DDS Statistics Backend* Github repository, by following [this](#) link.

8.1 Exception

8.1.1 BadParameter

class `eprosima::statistics_backend::BadParameter` : public `eprosima::statistics_backend::Exception`
Exception to signal that an operation has been called with an invalid parameter.

Public Functions

BadParameter(const *BadParameter* &other) = default

Copies the *statistics_backend::BadParameter* exception into a new one.

Parameters **other** – The original exception object to copy

BadParameter &**operator=**(const *BadParameter* &other) = default

Copies the *statistics_backend::BadParameter* exception into the current one.

Parameters **other** – The original *statistics_backend::BadParameter* exception to copy

Returns the current *statistics_backend::BadParameter* exception after the copy

8.1.2 CorruptedFile

class `eprosima::statistics_backend::CorruptedFile` : public `eprosima::statistics_backend::Exception`
Exception to signal that a file with an unexpected format has been loaded.

Public Functions

CorruptedFile(const *CorruptedFile* &other) = default

Copies the *statistics_backend::CorruptedFile* exception into a new one.

Parameters **other** – The original exception object to copy

CorruptedFile &**operator=**(const *CorruptedFile* &other) = default

Copies the *statistics_backend::CorruptedFile* exception into the current one.

Parameters **other** – The original *statistics_backend::CorruptedFile* exception to copy

Returns the current *statistics_backend::CorruptedFile* exception after the copy

8.1.3 Error

class `eprosima::statistics_backend::Error` : public `eprosima::statistics_backend::Exception`
Exception to signal a generic error that falls in no other specific category.

Public Functions

Error(const *Error* &other) = default

Copies the *statistics_backend::Error* exception into a new one.

Parameters other – The original exception object to copy

Error &**operator**=(const *Error* &other) = default

Copies the *statistics_backend::Error* exception into the current one.

Parameters other – The original *statistics_backend::Error* exception to copy

Returns the current *statistics_backend::Error* exception after the copy

8.1.4 Exception

class `eprosima::statistics_backend::Exception` : public exception
Base class for all exceptions thrown by the eProsimas statistics backend library.

Subclassed by *eprosima::statistics_backend::BadParameter*, *eprosima::statistics_backend::CorruptedFile*,
eprosima::statistics_backend::Error, *eprosima::statistics_backend::Inconsistency*,
eprosima::statistics_backend::PreconditionNotMet, *eprosima::statistics_backend::Unsupported*

Public Functions

Exception(const char *message) noexcept

Construct a new *statistics_backend::Exception* object.

Parameters message – The message to be returned by *what()*

Exception(const std::string &message)

Construct a new *statistics_backend::Exception* object.

Parameters message – The message to be returned by *what()*

Exception(const *Exception* &other) = default

Copies the *statistics_backend::Exception* object into a new one.

Parameters other – The original exception object to copy

Exception &**operator**=(const *Exception* &other) = default

Copies the *statistics_backend::Exception* object into the current one.

Parameters other – The original exception object to copy

Returns the current *statistics_backend::Exception* object after the copy

virtual const char ***what**() const noexcept override

Returns the explanatory string of the exception.

Returns Null-terminated string with the explanatory information

8.1.5 Inconsistency

class `eprosima::statistics_backend::Inconsistency` : public `eprosima::statistics_backend::Exception`
Exception to signal that an inconsistency inside the database has been found.

Public Functions

Inconsistency(const *Inconsistency* &other) = default
Copies the *statistics_backend::Inconsistency* exception into a new one.

Parameters **other** – The original exception object to copy

Inconsistency &**operator**=(const *Inconsistency* &other) = default
Copies the *statistics_backend::Inconsistency* exception into the current one.

Parameters **other** – The original *statistics_backend::Inconsistency* exception to copy

Returns the current *statistics_backend::Inconsistency* exception after the copy

8.1.6 PreconditionNotMet

class `eprosima::statistics_backend::PreconditionNotMet` : public `eprosima::statistics_backend::Exception`
Exception to signal that an operation cannot be performed because the preconditions are not met.

Public Functions

PreconditionNotMet(const *PreconditionNotMet* &other) = default
Copies the *statistics_backend::PreconditionNotMet* exception into a new one.

Parameters **other** – The original exception object to copy

PreconditionNotMet &**operator**=(const *PreconditionNotMet* &other) = default
Copies the *statistics_backend::PreconditionNotMet* exception into the current one.

Parameters **other** – The original *statistics_backend::PreconditionNotMet* exception to copy

Returns the current *statistics_backend::PreconditionNotMet* exception after the copy

8.1.7 Unsupported

class `eprosima::statistics_backend::Unsupported` : public `eprosima::statistics_backend::Exception`
Exception to signal that an operation is not supported.

Public Functions

Unsupported(const *Unsupported* &other) = default

Copies the *statistics_backend::Unsupported* exception into a new one.

Parameters **other** – The original exception object to copy

Unsupported &**operator**=(const *Unsupported* &other) = default

Copies the *statistics_backend::Unsupported* exception into the current one.

Parameters **other** – The original *statistics_backend::Unsupported* exception to copy

Returns the current *statistics_backend::Unsupported* exception after the copy

8.2 Listener

8.2.1 CallbackKind

enum `eprosima::statistics_backend::CallbackKind`

Each value identifies one of the user callbacks available on the library. These values can be combined with the ‘|’ operator to form a mask and configure which events are going to be notified to the user.

See *CallbackMask*

Values:

enumerator **ON_TOPIC_DISCOVERY**

Represents the `on_topic_discovery()` callback.

enumerator **ON_PARTICIPANT_DISCOVERY**

Represents the `on_participant_discovery()` callback.

enumerator **ON_DATAWRITER_DISCOVERY**

Represents the `on_datawriter_discovery()` callback.

enumerator **ON_DATAREADER_DISCOVERY**

Represents the `on_datareader_discovery()` callback.

enumerator **ON_HOST_DISCOVERY**

Represents the `on_host_discovery()` callback.

enumerator **ON_USER_DISCOVERY**

Represents the `on_user_discovery()` callback.

enumerator **ON_PROCESS_DISCOVERY**

Represents the `on_process_discovery()` callback.

enumerator **ON_LOCATOR_DISCOVERY**

Represents the `on_locator_discovery()` callback.

enumerator **ON_DATA_AVAILABLE**

Represents the `on_data_available()` callback.

8.2.2 CallbackMask

using `eprosima::statistics_backend::CallbackMask = Bitmask<CallbackKind>`
Bitmask of callback kinds.

values of `CallbackKind` can be combined with the ‘|’ operator to build the mask:

```
CallbackMask mask = CallbackKind::ON_DATAWRITER_DISCOVERY | CallbackKind::ON_
↳DATAREADER_DISCOVERY;
```

See *Bitmask*

8.2.3 DomainListener

class `eprosima::statistics_backend::DomainListener`

Subclassed by `eprosima::statistics_backend::PhysicalListener`

Public Functions

virtual `~DomainListener()` = default
 Virtual destructor.

inline virtual void **on_topic_discovery**(*EntityId* domain_id, *EntityId* topic_id, const *Status* &status)
 This function is called when a new Topic is discovered by the library.

Parameters

- **domain_id** – Entity ID of the domain in which the topic has been discovered.
- **topic_id** – Entity ID of the discovered topic.
- **status** – The status of the discovered topic.

inline virtual void **on_participant_discovery**(*EntityId* domain_id, *EntityId* participant_id, const *Status* &status)

This function is called when a new DomainParticipant is discovered by the library, or a previously discovered DomainParticipant changes its QOS or is removed.

Parameters

- **domain_id** – Entity ID of the domain in which the DataReader has been discovered.
- **participant_id** – Entity ID of the discovered DomainParticipant.
- **status** – The status of the discovered DomainParticipants.

inline virtual void **on_datareader_discovery**(*EntityId* domain_id, *EntityId* datareader_id, const *Status* &status)

This function is called when a new DataReader is discovered by the library, or a previously discovered DataReader changes its QOS or is removed.

Parameters

- **domain_id** – Entity ID of the domain in which the DataReader has been discovered.
- **datareader_id** – Entity ID of the discovered DataReader.
- **status** – The status of the discovered DataReaders.

inline virtual void **on_datawriter_discovery**(*EntityId* domain_id, *EntityId* datawriter_id, const *Status* &status)

This function is called when a new DataWriter is discovered by the library, or a previously discovered DataWriter changes its QoS or is removed.

Parameters

- **domain_id** – Entity ID of the domain in which the DataWriter has been discovered.
- **datawriter_id** – Entity ID of the discovered DataWriter.
- **status** – The status of the discovered DataWriters.

inline virtual void **on_data_available**(*EntityId* domain_id, *EntityId* entity_id, *DataKind* data_kind)

This function is called when a new data sample is available.

Parameters

- **domain_id** – Entity ID of the domain to which the data belongs.
- **entity_id** – Entity ID of the entity to which the data refers.
- **data_kind** – Data kind of the received data.

struct **Status**

Public Members

int32_t **total_count** = 0

Total cumulative count of the entities discovered so far.

This value increases monotonically with every new discovered entity.

int32_t **total_count_change** = 0

The change in total_count since the last time the listener was called.

This value can be positive, negative or zero, depending on the entity being discovered, undiscovered or only the QoS of the entity being changed since the last time the listener was called.

int32_t **current_count** = 0

The number of currently discovered entities.

This value can only be positive or zero.

int32_t **current_count_change** = 0

The change in current_count since the last time the listener was called.

This value can be positive, negative or zero, depending on the entity being discovered, undiscovered or only the QoS of the entity being changed since the last time the listener was called.

8.2.4 PhysicalListener

```
class eprosima::statistics_backend::PhysicalListener : public
eprosima::statistics_backend::DomainListener
```

Public Functions

```
virtual ~PhysicalListener() = default
    Virtual destructor.
```

```
inline virtual void on_host_discovery(EntityId host_id, const Status &status)
    This function is called when a new Host is discovered by the library.
```

Parameters

- **host_id** – Entity ID of the discovered Host.
- **status** – The status of the discovered Host.

```
inline virtual void on_user_discovery(EntityId user_id, const Status &status)
    This function is called when a new User is discovered by the library.
```

Parameters

- **user_id** – Entity ID of the discovered User.
- **status** – The status of the discovered User.

```
inline virtual void on_process_discovery(EntityId process_id, const Status &status)
    This function is called when a new Process is discovered by the library.
```

Parameters

- **process_id** – Entity ID of the discovered Process.
- **status** – The status of the discovered Process.

```
inline virtual void on_locator_discovery(EntityId locator_id, const Status &status)
    This function is called when a new Locator is discovered by the library.
```

Parameters

- **locator_id** – Entity ID of the discovered Locator.
- **status** – The status of the discovered Locator.

8.3 StatisticsBackend

```
class eprosima::statistics_backend::StatisticsBackend
```

Public Functions

StatisticsBackend() = delete

Deleted constructor, since the whole interface is static.

Public Static Functions

```
static void set_physical_listener(PhysicalListener *listener, CallbackMask callback_mask =  
                                CallbackMask::all(), DataKindMask data_mask =  
                                DataKindMask::none())
```

Set the listener for the physical domain events.

Any physical listener already configured will be replaced by the new one. The provided pointer to the listener can be null, in which case, any physical listener already configured will be removed.

Parameters

- **listener** – The listener with the callback implementations.
- **callback_mask** – Mask of the callbacks. Only the events that have the mask bit set will be informed.
- **data_mask** – Mask of the data types that will be monitored.

```
static EntityId init_monitor(DomainId domain, DomainListener *domain_listener = nullptr, CallbackMask  
                             callback_mask = CallbackMask::all(), DataKindMask data_mask =  
                             DataKindMask::none())
```

Starts monitoring on a given domain.

This function creates a new statistics DomainParticipant that starts monitoring the requested domain ID.

Parameters

- **domain** – The domain ID of the DDS domain to monitor.
- **domain_listener** – Listener with the callback to use to inform of events.
- **callback_mask** – Mask of the callbacks. Only the events that have the mask bit set will be informed.
- **data_mask** – Mask of the data types that will be monitored.

Throws

- `eprosima::statistics_backend::BadParameter` – if a monitor is already created for the given domain.
- `eprosima::statistics_backend::Error` – if the creation of the monitor fails.

Returns The ID of the created statistics Domain.

```
static EntityId init_monitor(std::string discovery_server_locators, DomainListener *domain_listener =  
                             nullptr, CallbackMask callback_mask = CallbackMask::all(), DataKindMask  
                             data_mask = DataKindMask::none())
```

Starts monitoring the network corresponding to a server.

This function creates a new statistics DomainParticipant that starts monitoring the network of the server with the given locators. The server `GuidPrefix_t` is set to the default one: `eprosima::fastdds::rtps::DEFAULT_ROS2_SERVER_GUIDPREFIX`. If any other server is to be used, call the overload method that receives the `GuidPrefix_t` as parameter.

The format to specify a locator is: `kind:[IP]:port`, where:

- **kind** is one of { UDPv4, TCPv4, UDPv6, TCPv4 }
- **IP** is the IP address
- **port** is the IP port Note that SHM locators are not supported. For any server configured with shared memory locators, initialize the monitor using only the non shared memory locators.

Parameters

- **discovery_server_locators** – The locator list of the server whose network is to be monitored, formatted as a semicolon separated list of locators.
- **domain_listener** – Listener with the callback to use to inform of events.
- **callback_mask** – Mask of the callbacks. Only the events that have the mask bit set will be informed.
- **data_mask** – Mask of the data types that will be monitored.

Returns The ID of the created statistics Domain.

```
static EntityId init_monitor(std::string discovery_server_guid_prefix, std::string discovery_server_locators,
                             DomainListener *domain_listener = nullptr, CallbackMask callback_mask =
                             CallbackMask::all(), DataKindMask data_mask = DataKindMask::none())
```

Starts monitoring the network corresponding to a server.

This function creates a new statistics DomainParticipant that starts monitoring the network of the server with the given GuidPrefix_t and with the given locators.

The format to specify a locator is: **kind**: [IP] :port, where:

- **kind** is one of { UDPv4, TCPv4, UDPv6, TCPv4 }
- **IP** is the IP address
- **port** is the IP port Note that SHM locators are not supported. For any server configured with shared memory locators, initialize the monitor using only the non shared memory locators.

Parameters

- **discovery_server_guid_prefix** – Server GuidPrefix_t to be monitored.
- **discovery_server_locators** – The locator list of the server whose network is to be monitored, formatted as a semicolon separated list of locators.
- **domain_listener** – Listener with the callback to use to inform of events.
- **callback_mask** – Mask of the callbacks. Only the events that have the mask bit set will be informed.
- **data_mask** – Mask of the data types that will be monitored.

Returns The ID of the created statistics Domain.

```
static void restart_monitor(EntityId monitor_id)
```

Restarts a given monitor.

This function restarts a domain monitor. If the monitor is still active (meaning it has not being stopped), this function takes no effect.

Parameters **monitor_id** – The entity ID of the monitor to restart.

```
static void stop_monitor(EntityId monitor_id)
```

Stops a given monitor.

This function stops a domain monitor. After stopping, the statistical data related to the domain is still accessible.

Parameters *monitor_id* – The entity ID of the monitor to stop.

Throws `eprosima::statistics_backend::BadParameter` – if the given monitor ID is not yet registered.

```
static void clear_monitor(EntityId monitor_id)
```

Clear the data of a domain given its monitor.

This function clears all the data related to a domain given its monitor ID. If the monitor is still active (meaning it has not being stopped), this functions takes no effect. After clearing, the statistical data related to the domain is deleted and therefore no longer accessible.

Parameters *monitor_id* – The entity ID of the monitor to stop.

```
static void set_domain_listener(EntityId monitor_id, DomainListener *listener = nullptr, CallbackMask  
    callback_mask = CallbackMask::all(), DataKindMask data_mask =  
    DataKindMask::none())
```

Set the listener of a monitor for the domain events.

Any domain listener already configured will be replaced by the new one. The provided pointer to the listener can be null, in which case, any domain listener already configured will be removed.

Parameters

- **monitor_id** – The entity ID of the monitor.
- **listener** – The listener with the callback implementations.
- **callback_mask** – Mask of the callbacks. Only the events that have the mask bit set will be informed.
- **data_mask** – Mask of the data types that will be monitored.

Throws `eprosima::statistics_backend::BadParameter` – if the given monitor ID is not yet registered.

```
static std::vector<EntityId> get_entities(EntityKind entity_type, EntityId entity_id = EntityId::all())
```

Get all the entities of a given type related to another entity.

Get all the entity ids for every entity of kind *entity_type* that is connected with entity *entity_id*. Connection between entities means they are directly connected by a contained/connect relation (i.e. Host - User | Domain - Topic) or that connected entities are connected to it.

Use case: To get all host in the system, use arguments HOST and *EntityId*::all().

Use case: To get all locators from a participant with id X, use arguments LOCATOR and X, this will get all the locators that are connected with the endpoints this participant has.

In case the *entity_id* is not specified, all entities of type *entity_type* are returned.

Parameters

- **entity_type** – The type of entities for which the search is performed.
- **entity_id** – The ID of the entity to which the resulting entities are related.

Throws `eprosima::statistics_backend::BadParameter` – in the following cases:

- if the *entity_kind* is INVALID.

- if the `entity_id` does not reference a entity contained in the database or is not `EntityId::all()`.
- if the `EntityKind` of the Entity with `entity_id` is `INVALID`.

Returns All entities of type `entity_type` that are related to `entity_id`.

static bool **is_active**(*EntityId* entity_id)

Returns whether the entity is active.

For monitors, active means that no call to `stop_monitor()` has been performed since the last time the monitor was activated. For the rest of entities, active means that there is statistical data being reported within the entity.

Parameters `entity_id` – The ID of the entity whose activeness is requested.

Returns true if active, false otherwise.

static bool **is_metatraffic**(*EntityId* entity_id)

Returns whether the entity is related to a metatraffic topic.

For Topics, it is true when they are used for sharing metatraffic data. For `DDSEndpoints`, it is true when their associated to a metatraffic Topic. For the rest of entities, metatraffic is always false.

Parameters `entity_id` – The ID of the entity whose metatraffic attribute is requested.

Returns true if metatraffic, false otherwise.

static *EntityKind* **get_type**(*EntityId* entity_id)

Returns the entity kind of a given id.

Parameters `entity_id` – The ID of the entity whose type is requested.

Throws `eprosima::statistics_backend::BadParameter` – if there is no entity with the given ID.

Returns `EntityKind` of `entity_id`.

static *Info* **get_info**(*EntityId* entity_id)

Get the meta information of a given entity.

Parameters `entity_id` – The entity for which the meta information is retrieved.

Returns Info object describing the entity's meta information.

```
static std::vector<StatisticsData> get_data(DataKind data_type, const std::vector<EntityId>
&entity_ids_source, const std::vector<EntityId>
&entity_ids_target, uint16_t bins = 0, Timestamp t_from =
Timestamp(), Timestamp t_to = std::chrono::system_clock::now(),
StatisticKind statistic = StatisticKind::NONE)
```

Provides access to the data measured during the monitoring.

Use this function for data types that relate to two entities, as described in `DataType`.

For data types that relate to a single entity, use the overloaded function that takes a single entity as argument.

`t_from` and `t_to` define the time interval for which the measurements will be returned. This time interval is further divided into `bin` segments of equal length, and a measurement is returned for each segment. Consequently, `t_to` should be greater than `t_from` by at least `bin` nanoseconds.

Measurement time and intervals

If `bin` is zero, no statistic is calculated and the raw data values in the requested time interval are returned.

The kind of statistic calculated for each `bin` segment is indicated by `statistic`. In this implementation, if `statistic` is `NONE`, the first raw data point in the segment is returned.

Statistics

See *StatisticsBackend*

Parameters

- **data_type** – The type of the measurement being requested.
- **entity_ids_source** – Ids of the source entities of the requested data. These IDs must correspond to entities of specific kinds depending on the `data_type`.
- **entity_ids_target** – Ids of the target entities of the requested data. These IDs must correspond to entities of specific kinds depending on the `data_type`.
- **bins** – Number of time intervals in which the measurement time is divided.
- **t_from** – Starting time of the returned measures.
- **t_to** – Ending time of the returned measures.
- **statistic** – Statistic to calculate for each of the bins.

Throws `eprosima::statistics_backend::BadParameter` – if the above preconditions are not met.

Returns a vector of `bin` elements with the values of the requested statistic.

```
static std::vector<StatisticsData> get_data(DataKind data_type, const std::vector<EntityId> &entity_ids,
                                           uint16_t bins = 0, Timestamp t_from = Timestamp(), Timestamp
                                           t_to = std::chrono::system_clock::now(), StatisticKind statistic =
                                           StatisticKind::NONE)
```

Provides access to the data measured during the monitoring.

Use this function for data types that relate to a single entity, as described in `DataType`.

For data types that relate to two entities, use the overloaded function that takes a source and a target entity as arguments.

`t_from` and `t_to` define the time interval for which the measurements will be returned. This time interval is further divided into `bin` segments of equal length, and a measurement is returned for each segment. Consequently, `t_to` should be greater than `t_from` by at least `bin` nanoseconds.

Measurement time and intervals

If `bin` is zero, no statistic is calculated and the raw data values in the requested time interval are returned.

The kind of statistic calculated for each `bin` segment is indicated by `statistic`. In this implementation, if `statistic` is `NONE`, the first raw data point in the segment is returned.

Statistics

See *StatisticsBackend*

Parameters

- **data_type** – The type of the measurement being requested.

- **entity_ids** – Ids of the entities of the requested data. These IDs must correspond to entities of specific kinds depending on the `data_type`.
- **bins** – Number of time intervals in which the measurement time is divided.
- **t_from** – Starting time of the returned measures.
- **t_to** – Ending time of the returned measures.
- **statistic** – Statistic to calculate for each of the bins.

Throws `eprosima::statistics_backend::BadParameter` – if the above preconditions are not met.

Returns a vector of bin elements with the values of the requested statistic.

```
static std::vector<StatisticsData> get_data(DataKind data_type, const std::vector<EntityId>
                                         &entity_ids_source, const std::vector<EntityId>
                                         &entity_ids_target, uint16_t bins, StatisticKind statistic)
```

Overload of `get_data` method without time arguments.

It calls the `get_data` method with the default time arguments. It is used to set the `statistic` argument with default time values.

Parameters

- **data_type** – The type of the measurement being requested.
- **entity_ids_source** – Ids of the source entities of the requested data. These IDs must correspond to entities of specific kinds depending on the `data_type`.
- **entity_ids_target** – Ids of the target entities of the requested data. These IDs must correspond to entities of specific kinds depending on the `data_type`.
- **bins** – Number of time intervals in which the measurement time is divided.
- **statistic** – Statistic to calculate for each of the bins.

Throws `eprosima::statistics_backend::BadParameter` – if the above preconditions are not met.

Returns a vector of bin elements with the values of the requested statistic.

```
static std::vector<StatisticsData> get_data(DataKind data_type, const std::vector<EntityId> &entity_ids,
                                         uint16_t bins, StatisticKind statistic)
```

Overload of `get_data` method without time arguments.

It calls the `get_data` method with the default time arguments. It is used to set the `statistic` argument with default time values.

Parameters

- **data_type** – The type of the measurement being requested.
- **entity_ids** – Ids of the entities of the requested data. These IDs must correspond to entities of specific kinds depending on the `data_type`.
- **bins** – Number of time intervals in which the measurement time is divided.
- **statistic** – Statistic to calculate for each of the bins.

Throws `eprosima::statistics_backend::BadParameter` – if the above preconditions are not met.

Returns a vector of bin elements with the values of the requested statistic.

static *Graph* **get_graph()**

Get the topology graph.

Returns Graph object describing the complete topology of the entities.

static DatabaseDump **dump_database**(bool clear)

Get a dump of the database.

Parameters **clear** – If true, clear all the statistics data of all entities.

Returns DatabaseDump object representing the backend database.

static void **dump_database**(const std::string &filename, bool clear)

Dump Fast DDS Statistics Backend's database to a file.

Parameters

- **filename** – The name of the file where the database is dumped.
- **clear** – If true, clear all the statistics data of all entities.

static void **load_database**(const std::string &filename)

Load Fast DDS Statistics Backend's database from a file.

Parameters **filename** – The name of the file from which where the database is loaded.

Throws `eprosima::statistics_backend::BadParameter` – if the file does not exist.

Pre The Backend's database has no data. This means that no monitors were initialized since the Backend started, or that the Backend has been *reset()*.

static void **clear_statistics_data**()

Clear all statistics data of all entities.

static void **clear_inactive_entities**()

Remove all inactive entities from database.

static void **reset**()

Resets the Fast DDS Statistics Backend.

After calling this method, the Fast DDS Statistics Backend reverts to its default state, as it was freshly started:

- All the data in the database is erased.
- All monitors are removed and cannot be restarted afterwards.
- The physical listener is removed.
- The physical listener callback mask is set to `CallbackMask::none()`.
- The physical listener data mask is set to `DataMask::none()`.

Pre There are no active monitors. There can be inactive monitors.

static std::vector<std::pair<*EntityKind*, *EntityKind*>> **get_data_supported_entity_kinds**(*DataKind*
data_kind)

Return the *EntityKind* of the entities to which a *DataKind* refers.

Some *DataKind* relate to a single *Entity* of a given *EntityKind*. This is the case of `SUBSCRIPTION_THROUGHPUT`, that always relates to a `DATAREADER`. Other *DataKind* relate to two different *Entity*, each one of a given *EntityKind*. For example, `FASTDDS_LATENCY` relates to a `DATAWRITER` as source and a `DATAREADER` as target of the data flow. In the specific case of `DISCOVERY_TIME`, the *DataKind* relates to a `PARTICIPANT` as the discoverer, but can relate to a `DATAWRITER`, `DATAREADER` or another `PARTICIPANT` as the discovered entity.

Given a `DataKind`, this method provides a collection of all pairs of `EntityKind` to which this `DataKind` relates.

- For a `DataKind` that only relates to one Entity, the first element of the pair is the `EntityKind` of such Entity, while the second element is `EntityKind::INVALID`.
- For a `DataKind` that relates to two Entity, the first element of the pair is the `EntityKind` of the source Entity, while the second element is the `EntityKind` of the target Entity.

The source and target pairs returned by this method are exactly the accepted source and target `EntityKind` accepted by `get_data` for the given `DataKind`. This is convenient to prepare a call to `get_data` from an `EntityKind`. First, call `get_data_supported_entity_kinds` with the `EntityKind` to get the `EntityKinds` of the related entities. Then, call `get_entities` to get the available entities for that kind. Finally, call `get_data` with the pairs that `get_entities` returns.

i.e. Get the `DISCOVERY_TIME` of all entities on Host2 discovered by Host1:

```
// Get all the EntityKind pairs related to DISCOVERY_TIME.
std::vector<std::pair<EntityKind, EntityKind>> types_list =
    StatisticsBackend::get_data_supported_entity_kinds(DataKind::DISCOVERY_
->TIME);

// Iterate over all the valid pairs composing the final result
std::vector<StatisticsData> discovery_times;
for (std::pair<EntityKind, EntityKind> type_pair : types_list)
{
    // Take the data for this pair and append it to the existing data
    std::vector<StatisticsData> tmp = StatisticsBackend::get_data(
        DataKind::DISCOVERY_TIME,
        StatisticsBackend::get_entities(type_pair.first, host1_id),
        StatisticsBackend::get_entities(type_pair.second, host2_id));

    discovery_times.insert(discovery_times.end(), tmp.begin(), tmp.end());
}
```

See `DataKind`

See `get_data`

Parameters `data_kind` – Data kind.

Returns list of `EntityKind` pairs with the entity kinds to which a `DataKind` refers.

static void **set_alias**(*EntityId* entity_id, const std::string &alias)

Set a new alias for the entity.

Parameters

- **entity_id** – The *EntityId* of the entity.
- **alias** – New alias that will replace the old one.

Throws `eprosima::statistics_backend::BadParameter` – if there is no entity with the given ID.

8.4 Types

8.4.1 Bitmask

template<typename **E**>

class **Bitmask**

Generic bitmask for an enumerated type.

This class can be used as a companion bitmask of any enumerated type whose values have been constructed so that a single bit is set for each enum value. The enumerated values can be seen as the names of the bits in the bitmask.

Bitwise operations are defined between masks of the same type, between a mask and its companion enumeration, and between enumerated values.

```
enum my_enum
{
    RED    = 1 << 0,
    GREEN  = 1 << 1,
    BLUE   = 1 << 2
};

// Combine enumerated labels to create a mask
Bitmask<my_enum> yellow_mask = RED | GREEN;

// Combine a mask with a value to create a new mask
Bitmask<my_enum> white_mask = yellow_mask | BLUE;

// Flip all the bits in the mask
Bitmask<my_enum> black_mask = ~white_mask;

// Set a bit in the mask
black_mask.set(RED);

// Test if a bit is set in the mask
bool is_red = white_mask.is_set(RED);
```

tparam E The enumerated type for which the bitmask is constructed

8.4.2 DataKind

enum eprosimastatistics_backend::**DataKind**

Indicates the Type of Data stored by the Backend

Signature	Entities source	Entity target	No. entities
FASTDDS_LATENCY	DataWriter	DataReader	2
NETWORK_LATENCY	DomainParticipant	Locator	2

(continues on next page)

(continued from previous page)

PUBLICATION_THROUGHPUT	DataWriter		1	
SUBSCRIPTION_THROUGHPUT	DataReader		1	
RTPS_PACKETS_SENT	DomainParticipant	Locator	2	
RTPS_BYTES_SENT	DomainParticipant	Locator	2	
RTPS_PACKETS_LOST	DomainParticipant	Locator	2	
RTPS_BYTES_LOST	DomainParticipant	Locator	2	
RESENT_DATA	DataWriter		1	
HEARTBEAT_COUNT	DataWriter		1	
ACKNACK_COUNT	DataReader		1	
NACKFRAG_COUNT	DataReader		1	
GAP_COUNT	DataWriter		1	
DATA_COUNT	DataWriter		1	
PDP_PACKETS	DomainParticipant		1	
EDP_PACKETS	DomainParticipant		1	
DISCOVERY_TIME	DomainParticipant	DDSEntity	2	
SAMPLE_DATAS	DataWriter		1	

*Values:*enumerator **INVALID**

Represents no valid data kind.

enumerator **FASTDDS_LATENCY**

Latency between a write operation (writer side) and data available (notification to user in reader side)

enumerator **NETWORK_LATENCY**

Latency between Locators pair.

enumerator **PUBLICATION_THROUGHPUT**

Amount of data [Mb/s] sent by a DataWriter.

enumerator **SUBSCRIPTION_THROUGHPUT**

Amount of data [Mb/s] received by a DataReader.

enumerator **RTPS_PACKETS_SENT**

Amount of packets sent from a DDS Entity to a Locator.

enumerator **RTPS_BYTES_SENT**

Amount of bytes sent from a DDS Entity to a Locator.

enumerator **RTPS_PACKETS_LOST**

Amount of packets lost from a DDS Entity to a Locator.

enumerator **RTPS_BYTES_LOST**

Amount of bytes lost from a DDS Entity to a Locator.

enumerator **RESENT_DATA**

Amount of DATA/DATAFRAG sub-messages resent from a DataWriter/DomainParticipant.

enumerator **HEARTBEAT_COUNT**

Amount of HEARTBEATs that each non discovery DataWriter/DomainParticipant sends.

enumerator **ACKNACK_COUNT**

Amount of ACKNACKs that each non discovery DataReader/DomainParticipant sends.

enumerator **NACKFRAG_COUNT**

Amount of NACKFRAGs that each non discovery DataReader/DomainParticipant sends.

enumerator **GAP_COUNT**

Amount of GAPS sub-messages sent from a DataWriter/DomainParticipant.

enumerator **DATA_COUNT**

Amount of DATA/DATAFRAG sub-messages that each non discovery DataWriter sends.

enumerator **PDP_PACKETS**

Amount of PDP packets sent by Participant.

enumerator **EDP_PACKETS**

Amount of EDP packets sent by Participant.

enumerator **DISCOVERY_TIME**

Time when a DDS Entity discovers another DDS entity.

enumerator **SAMPLE_DATAS**

Amount of DATA/DATAFRAG sub-messages needed to send a single sample.

8.4.3 DataKindMask

using `eprosima::statistics_backend::DataKindMask = Bitmask<DataKind>`

Bitmask of data kinds.

values of DataKind can be combined with the '|' operator to build the mask:

```
DataKindMask mask = DataKind::PUBLICATION_THROUGHPUT | DataKind::SUBSCRIPTION_
↪THROUGHPUT;
```

See *Bitmask*

8.4.4 DomainId

using `eprosima::statistics_backend::DomainId = uint32_t`

Type DDS Domain IDs

8.4.5 EntityId

class eprosima::statistics_backend::EntityId

Public Functions

EntityId() noexcept

Instantiate an *EntityId*. The internal value is set to *EntityId::invalid*.

EntityId(int64_t value) noexcept

Instantiate an *EntityId* from an integer.

Parameters value – The value to use as internal value on the *EntityId*

EntityId(EntityId &&entity_id) noexcept = default

Move constructor.

Parameters entity_id – The moved *EntityId*

EntityId(const EntityId &entity_id) noexcept = default

Copy constructor.

Parameters entity_id – The copied *EntityId*

EntityId &operator=(const EntityId &entity_id) noexcept = default

Copy assignment operator.

Parameters entity_id – The assigned *EntityId*

EntityId &operator=(EntityId &&entity_id) noexcept = default

Move assignment operator.

Parameters entity_id – The assigned *EntityId*

void **invalidate()** noexcept

Invalidate an *EntityId*.

Post *is_valid()* returns false

bool **is_valid()** const noexcept

Check whether an *EntityId* is valid.

Returns True if valid, false otherwise

bool **is_all()** const noexcept

Check whether an *EntityId* is the ID representing all entities.

Returns True if is ENTITY_ID_ALL, false otherwise

bool **is_valid_and_unique()** const noexcept

Check whether an *EntityId* is an ID representing one specific entity.

Returns True if it is valid and not ENTITY_ID_ALL, false otherwise

int64_t **value()** const noexcept

Get the internal value of the *EntityId*.

Returns An int64_t with the representing internal value

Public Static Functions

static *EntityId* **all**() noexcept
Get the *EntityId* to refer all entities at once.

Returns An ID that refers all entities.

static *EntityId* **invalid**() noexcept
Get an invalid *EntityId*.

Returns An ID that is invalid

inline std::ostream &eprosima::statistics_backend::operator<<(std::ostream &output, const *EntityId* &entity_id)

Serialize an *EntityId* to std::ostream.

Parameters

- **output** – The output std::ostream
- **entity_id** – The *EntityId* to serialize

inline bool eprosima::statistics_backend::operator<(const *EntityId* &entity_id_1, const *EntityId* &entity_id_2)

Check whether an *EntityId* is smaller than another one.

Parameters

- **entity_id_1** – The left-side of the operation
- **entity_id_2** – The right-side of the operation

inline bool eprosima::statistics_backend::operator<=(const *EntityId* &entity_id_1, const *EntityId* &entity_id_2)

Check whether an *EntityId* is smaller or equal than another one.

Parameters

- **entity_id_1** – The left-side of the operation
- **entity_id_2** – The right-side of the operation

inline bool eprosima::statistics_backend::operator>(const *EntityId* &entity_id_1, const *EntityId* &entity_id_2)

Check whether an *EntityId* is greater than another one.

Parameters

- **entity_id_1** – The left-side of the operation
- **entity_id_2** – The right-side of the operation

inline bool eprosima::statistics_backend::operator>=(const *EntityId* &entity_id_1, const *EntityId* &entity_id_2)

Check whether an *EntityId* is greater or equal than another one.

Parameters

- **entity_id_1** – The left-side of the operation
- **entity_id_2** – The right-side of the operation

inline bool eprosima::statistics_backend::operator==(const *EntityId* &entity_id_1, const *EntityId* &entity_id_2)

Check whether an *EntityId* is equal to another one.

Parameters

- **entity_id_1** – The left-side of the operation
- **entity_id_2** – The right-side of the operation

```
inline bool eprosima::statistics_backend::operator!=(const EntityId &entity_id_1, const EntityId
&entity_id_2)
```

Check whether an *EntityId* is different than another one.

Parameters

- **entity_id_1** – The left-side of the operation
- **entity_id_2** – The right-side of the operation

8.4.6 EntityKind

```
enum eprosima::statistics_backend::EntityKind
```

Indicates the Type of an Entity in Statistics Backend structure

Values:

```
enumerator INVALID
```

Invalid entity kind.

```
enumerator HOST
```

Host/Machine where a participant is allocated.

```
enumerator USER
```

User that has executed a participant.

```
enumerator PROCESS
```

Process where a participant is running.

```
enumerator DOMAIN
```

Abstract DDS network by Domain or by Discovery Server.

```
enumerator TOPIC
```

DDS Topic.

```
enumerator PARTICIPANT
```

DDS Domain Participant.

```
enumerator DATAWRITER
```

DDS DataWriter.

```
enumerator DATAREADER
```

DDS DataReader.

```
enumerator LOCATOR
```

Physical locator that a communication is using (IP + port || SHM + port) Store the Locator Statistic data

8.4.7 Graph

using eprosima::statistics_backend::**Graph** = nlohmann::json

Topology graph tree structure. Please refer to <https://nlohmann.github.io/json/doxygen/index.html>

8.4.8 Info

using eprosima::statistics_backend::**Info** = nlohmann::json

Info tree structure. Please refer to <https://nlohmann.github.io/json/doxygen/index.html>

8.4.9 StatisticKind

enum eprosima::statistics_backend::**StatisticKind**

Values:

enumerator **NONE**

Non accumulative kind, it chooses a data point between the set given. Implemented to take the first data in set : [0]

enumerator **MEAN**

Numerical mean of values in the set.

enumerator **STANDARD_DEVIATION**

Standard Deviation of the values in the set.

enumerator **MAX**

Maximum value in the set.

enumerator **MIN**

Minimum value in the set.

enumerator **MEDIAN**

Median value of the set.

enumerator **COUNT**

Amount of values in the set.

enumerator **SUM**

Summation of the values in the set.

8.4.10 StatisticsData

```
using eprosima::statistics_backend::StatisticsData = std::pair<Timestamp, double>
```

Type of the data returned by the backend.

The first field represents the time at which the data was recorded. This can be the time of the raw data point if no bins are being used, or the starting time of the bin (see `get_data()`).

The second field represents the data value itself. This will be the value of the calculated statistic, or the raw data if no statistic has been requested (see `get_data()`).

See `get_data()`

8.4.11 Timestamp

```
using eprosima::statistics_backend::Timestamp = std::chrono::time_point<std::chrono::system_clock>
```

Type used to represent time points

8.4.12 JSON Tags

```
constexpr const char *eprosima::statistics_backend::ACTUAL_DUMP_VERSION = "0.0"
```

Actual version of the Database Dump.

VERSION 0.8.0

This release includes the following **features**:

1. New API to clear statistic data and remove inactive entities from database.

This release includes the following **improvements**:

1. CI improvements:

1. Include address-sanitizer job.
2. Flaky tests are run in a specific job.

1. Internal implementation improvements:

1. Remove database unused collections.
2. Smart pointers refactor using unique instead of shared pointers.

1. Example:

1. Improve example including new API.

This release includes the following **bugfixes**:

1. Memory leaks fixes reported by address-sanitizer.

PREVIOUS VERSIONS

10.1 Version 0.7.1

This release adds the following **improvements**:

- Update python dependencies for building the documentation
- Re-generate Fast DDS-Gen generated TypeSupport with Fast DDS-Gen v2.2.0
- Example to export ROS 2 statistics to Prometheus

10.2 Version 0.7.0

This release adds the following **feature**:

- Possibility of loading Fast DDS Statistics Backend's `DomainParticipant` from Fast DDS XML configuration files.

This release adds the following **improvements**:

- Warn the user the Fast DDS Statistics module needs to be enabled.
- Update documentation regarding `locator` specification.
- Fix destruction error in the Database's queue thread.

10.3 Version 0.6.0

This release adds the following **improvement**:

- Update statistics type support including physical data in `DISCOVERY_TOPIC`

10.4 Version 0.5.0

This release adds the following **improvements**:

- Improvements on Windows CI
- Refactor on processing queues to avoid data races with entity creation order
- Update Statistics Module type support

10.5 Version 0.4.0

This release has the following **API extensions**:

- *StatisticsBackend::is_metattraffic* allows the user to know if a specific topic or endpoint is related to metattraffic data.

This release adds the following **features**:

- Add HelloWorld Example.

This release includes the following **bug fixes and improvements**:

- Return the end of the time interval as the data point's timestamps instead of the initial one.
- Avoid adding the same locator twice to the database.
- Fix deadlock when accessing the database within a callback implementation.
- Avoid using deprecated namespace.
- Improvements on CI.

10.6 Version 0.3.0

This release has the following **API breaks**:

- *StatisticsBackend::dump_database* methods now have an additional argument *clear*.

This release adds the following **features**:

- Dumping the database can optionally delete the traffic data from the internal data structures after the dump is completed, in order to reduce memory footprint.
- Support to create monitors on discovery server networks.
- Statistics data related to meta-traffic are now collected under the builtin meta-traffic endpoint created on each participant.
- Entities removed from the network now have a *non-active* status.
- Transitions between active and non-active status are notified to the user listeners.
- It is possible to change the domain listener and mask after the monitor is created.

This release includes the following **bug fixes and improvements**:

- By default, statistics data is received using UDP transport, shared memory is disabled.
- Network latency data now relates to a source participant and a destination locator (previously a source locator and a destination locator).

- Statistics data can now trigger the discovery of a new locator.
- Improved entity names and aliases to be more user friendly.
- Solved an issue that may cause the internal database to freeze.
- Allow for topics with the same name to be on different domains.

10.7 Version 0.2.0

This minor release is API compatible with the previous minor release, but introduces **ABI breaks**:

- Methods and attributes have been added on several classes, so indexes of symbols on dynamic libraries may have changed.

This release adds the following **features**:

- Support for Windows platforms
- Dumped data can now be loaded to the backend
- Backend can now be reset to a clean state (deleting all the data and monitors)
- `StatisticsBackend::get_data` implementation for `SAMPLE_DATAS` and `DISCOVERY_TIME` sample types
- `StatisticsBackend::get_data` implementation for zero bins

It also includes the following **improvements**:

- The participant info now contains all the locators of the participant
- Entities have an alias that can be set by the user to facilitate identification

Some important **bugfixes** are also included:

- Fixed errors when the same topic name is used on different domains
- Fixed crashes on database queues when database operation fails

10.8 Version 0.1.0

This first release includes the following features:

- Starting and stopping monitoring a DDS domain ID
- Keeping track of discovered entities (hosts, users, processes, participants, topics, data readers, data writers, and locators).
- Listening and recording statistics data related to the discovered entities, as reported by the *Fast DDS* statistics module.
- Retrieving the recorded statistics data, preprocessed with the requested statistic (mean, standard deviation, maximum, minimum, median)
- Persisting the recorded statistics to a file.
- Loading the statistics from a persisted file.

INDEX

E

(C++ enumerator), 71
 eprosima::statistics_backend::ACTUAL_DUMP_VERSION (C++ member), 77
 eprosima::statistics_backend::BadParameter (C++ class), 55
 eprosima::statistics_backend::BadParameter::BadParameter (C++ function), 55
 eprosima::statistics_backend::BadParameter::operator= (C++ function), 55
 eprosima::statistics_backend::Bitmask (C++ class), 70
 eprosima::statistics_backend::CallbackKind (C++ enum), 58
 eprosima::statistics_backend::CallbackKind::ON_DATA_AVAILABLE (C++ enumerator), 58
 eprosima::statistics_backend::CallbackKind::ON_DATAREADER_DISCOVERY (C++ enumerator), 58
 eprosima::statistics_backend::CallbackKind::ON_DATAWRITER_DISCOVERY (C++ enumerator), 58
 eprosima::statistics_backend::CallbackKind::ON_HOST_DISCOVERY (C++ enumerator), 58
 eprosima::statistics_backend::CallbackKind::ON_LOCATOR_DISCOVERY (C++ enumerator), 58
 eprosima::statistics_backend::CallbackKind::ON_PARTICIPANT_DISCOVERY (C++ enumerator), 58
 eprosima::statistics_backend::CallbackKind::ON_PROCESS_DISCOVERY (C++ enumerator), 58
 eprosima::statistics_backend::CallbackKind::ON_TOPIC_DISCOVERY (C++ enumerator), 58
 eprosima::statistics_backend::CallbackKind::ON_USER_DISCOVERY (C++ enumerator), 58
 eprosima::statistics_backend::CallbackMask (C++ type), 59
 eprosima::statistics_backend::CorruptedFile (C++ class), 55
 eprosima::statistics_backend::CorruptedFile::CorruptedFile (C++ function), 55
 eprosima::statistics_backend::CorruptedFile::operator= (C++ function), 55
 eprosima::statistics_backend::DataKind (C++ enum), 70
 eprosima::statistics_backend::DataKind::ACKNACK_COUNT (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::DATA_COUNT (C++ enumerator), 72
 eprosima::statistics_backend::DataKind::DISCOVERY_TIME (C++ enumerator), 72
 eprosima::statistics_backend::DataKind::EDP_PACKETS (C++ enumerator), 72
 eprosima::statistics_backend::DataKind::FASTDDS_LATENCY (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::GAP_COUNT (C++ enumerator), 72
 eprosima::statistics_backend::DataKind::HEARTBEAT_COUNT (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::INVALID (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::NACKFRAG_COUNT (C++ enumerator), 72
 eprosima::statistics_backend::DataKind::NETWORK_LATENCY (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::PDP_PACKETS (C++ enumerator), 72
 eprosima::statistics_backend::DataKind::PUBLICATION_THROUGHPUT (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::RESET_DATA (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::RTPS_BYTES_LOST (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::RTPS_BYTES_SENT (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::RTPS_PACKETS_LOST (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::RTPS_PACKETS_SENT (C++ enumerator), 71
 eprosima::statistics_backend::DataKind::SAMPLE_DATAS (C++ enumerator), 72
 eprosima::statistics_backend::DataKind::SUBSCRIPTION_THROUGHPUT (C++ enumerator), 71
 eprosima::statistics_backend::DataKindMask (C++ type), 72
 eprosima::statistics_backend::DomainId (C++ type), 72
 eprosima::statistics_backend::DomainListener

(C++ function), 61
 eprosima::statistics_backend::PhysicalListenerDiscovery eprosima::statistics_backend::StatisticsBackend::is_active
 (C++ function), 61 (C++ function), 65
 eprosima::statistics_backend::PhysicalListenerDiscovery eprosima::statistics_backend::StatisticsBackend::is_metadata
 (C++ function), 61 (C++ function), 65
 eprosima::statistics_backend::PhysicalListenerDiscovery eprosima::statistics_backend::StatisticsBackend::load_data
 (C++ function), 61 (C++ function), 68
 eprosima::statistics_backend::PreconditionNotMet eprosima::statistics_backend::StatisticsBackend::reset
 (C++ class), 57 (C++ function), 68
 eprosima::statistics_backend::PreconditionNotMet eprosima::statistics_backend::StatisticsBackend::restart_monitoring
 (C++ function), 57 (C++ function), 63
 eprosima::statistics_backend::PreconditionNotMet eprosima::statistics_backend::StatisticsBackend::set_aliases
 (C++ function), 57 (C++ function), 69
 eprosima::statistics_backend::StatisticKind eprosima::statistics_backend::StatisticsBackend::set_domain
 (C++ enum), 76 (C++ function), 64
 eprosima::statistics_backend::StatisticKind::COUNT eprosima::statistics_backend::StatisticsBackend::set_physical
 (C++ enumerator), 76 (C++ function), 62
 eprosima::statistics_backend::StatisticKind::MAX eprosima::statistics_backend::StatisticsBackend::StatisticKind
 (C++ enumerator), 76 (C++ function), 62
 eprosima::statistics_backend::StatisticKind::MEAN eprosima::statistics_backend::StatisticsBackend::stop_monitoring
 (C++ enumerator), 76 (C++ function), 64
 eprosima::statistics_backend::StatisticKind::MEDIAN eprosima::statistics_backend::StatisticsData
 (C++ enumerator), 76 (C++ type), 77
 eprosima::statistics_backend::StatisticKind::MIN eprosima::statistics_backend::Timestamp
 (C++ enumerator), 76 (C++ type), 77
 eprosima::statistics_backend::StatisticKind::NONE eprosima::statistics_backend::Unsupported
 (C++ enumerator), 76 (C++ class), 57
 eprosima::statistics_backend::StatisticKind::STANDARD_DEVIATION eprosima::statistics_backend::Unsupported::operator=
 (C++ enumerator), 76 (C++ function), 58
 eprosima::statistics_backend::StatisticKind::SUM eprosima::statistics_backend::Unsupported::Unsupported
 (C++ enumerator), 76 (C++ function), 58
 eprosima::statistics_backend::StatisticsBackend
 (C++ class), 61
 eprosima::statistics_backend::StatisticsBackend::clear_inactive_entities
 (C++ function), 68
 eprosima::statistics_backend::StatisticsBackend::clear_monitor
 (C++ function), 64
 eprosima::statistics_backend::StatisticsBackend::clear_statistics_data
 (C++ function), 68
 eprosima::statistics_backend::StatisticsBackend::dump_database
 (C++ function), 68
 eprosima::statistics_backend::StatisticsBackend::get_data
 (C++ function), 65–67
 eprosima::statistics_backend::StatisticsBackend::get_data_supported_entity_kinds
 (C++ function), 68
 eprosima::statistics_backend::StatisticsBackend::get_entities
 (C++ function), 64
 eprosima::statistics_backend::StatisticsBackend::get_graph
 (C++ function), 68
 eprosima::statistics_backend::StatisticsBackend::get_info
 (C++ function), 65
 eprosima::statistics_backend::StatisticsBackend::get_type
 (C++ function), 65
 eprosima::statistics_backend::StatisticsBackend::init_monitor