MEP 8 -- Add metrics for proxy

Current state: Under Discussion

ISSUE: #5955 #6249 #7091 #7114 #7139 #7161 #7162

PRs: #5987 #7113 #7137 #7157

Keywords: `metric` `Proxy`

Released:

Summary

Proxy will provide an interface for users to show some important metrics information of Milvus.

Motivation

When users use the Milvus, they may want to know exactly how Milvus is running. Now we already have integrated prometheus into our metrics system, users can use prometheus to monitor the Milvus. We also have milvus-insight project to interact with Milvus, though prometheus have already provided the metrics interface, it's not easy for `milvus-insight` to use them and visualize them (If we do so, it seems that we are going to implement a front-end of parsing prometheus metrics). So as the access layer of Milvus, Proxy should provide an interface for users to show some important metrics information of Milvus.

Public Interfaces

This MEP will add a new interface below:

```
service MilvusService {
  rpc GetMetrics(GetMetricsRequest) returns (GetMetricsResponse) {}
}

message GetMetricsRequest {
  // request is the jsonic format string, in this way, we can also extend request easier,
  string request = 1;
}

message GetMetricResponse {
  common.Status status = 1;
  // response is the jsonic format string, in this way, we can also extend response easier
  string response = 2;
}
```

I will describe how this interface should be used later in Design Details.

Design Details

As #6249 mentioned, users may want to know below metrics:

- System info

- system version
- nodes info
    - node name
    - hardware info
        - cpu info
        - memory info
        - capacity info
    - connection info (which node connect to which node)
    - created time
    - updated time
- system configurations

- System statistics

- hardware statistics(all and per node)
    - cpu info/usage
    - storage capacity/usage
    - node memory capacity/usage
- loaded collections/partitions

        - name
        - loaded time
- collections/partitions count
- indexes count
- QPS
- latency

- Logs

- search/query logs
    - ip/time/query parameters
- system Logs
    - system up/down logs

How Proxy can get these metrics?

Most of metrics listed above can be reported directly, for example, hardware information, system configurations and etc.

For some meta-related metrics, such as loaded collections/partitions, Proxy can get these information from other components also.

There are two parts we need to discuss in detail:

- Event Log Metrics
- Milvus Connection Topology

Event Log Metrics

A way to handle event log metrics such as dd/dm/dq operations is to use message stream. Other components as the producers will continously produce event logs to this stream, then Proxy as the consumer can read all event logs from stream and sort them with timestamp order. In this way, we need to consider the retention time of event log, as Milvus keeps running, event logs will become more and larger, Proxy shouldn't hold so many event logs, so a clean-up background thread is required to clean the logs peridically. Of course, we can also choose to persistent the event logs and then users can specific the timestamp section of event logs. There are many cloud-native logging systems, such as fluentd, etc.

Milvus Connection Topology

I have two ideas about implementing this.

Plan 1 : Get topology information from other components. IndexCoord, QueryCoord, DataCoord should also provide an interface for Proxy to get metrics including node connection information. Since all coordinators in Milvus can manage their all related nodes, they can easily how many nodes connected to them and their nodes' hardware information.

- **Pros**: In fact, this interface for other components is required. Some meta-related metrics can only be reported by specific components.
- **Cons**: Every components will have the same logic to handle connection topology. If we have other coordinators later, they should also implement them. It will take a lot of time to implement and test.

Plan 2: Register the topology information into etcd. When nodes connect to coordinator, register their information too. Now we have an awsome encapsulation when nodes connect to others using `Session` implemented by @qingxiang.chen. We can also encapsulate this operation to `Session`.

- **Pros**: Reduce the duplicated work that all coordinators implement the same logic.
- **Cons**:
    1. Introduce the dependency between etcd and Proxy.
    2. Not easy to get resource usage of nodes, such as memory usage, cpu usage and etc.

Detailed Implementation of Plan 1

Every component in Milvus should have an interface GetMetrics to expose their metrics. As the user access layer, Proxy connects to other coordinators directly, such as RootCoord, DataCoord, QueryCoord and IndexCoord. Taking QueryCoord as an example, how could Proxy get topology metrics between QueryCoord and Query Nodes? The answer is that Proxy get these metrics from QueryCoord. QueryCoord manages all the Query Nodes, so the metrics of Query Nodes can be easily reported to QueryCoord and then QueryCoord an summary these metrics to Proxy.

We should add GetMetrics interface to rpc service of coordinator and related node:

```
service QueryCoord {
  rpc GetMetrics(milvus.GetMetricsRequest) returns (milvus.GetMetricsResponse) {}
}

service QueryNode {
  rpc GetMetrics(milvus.GetMetricsRequest) returns (milvus.GetMetricsResponse) {}
}
```

How to use `GetMetrics` interface?

The `request` and `response` are both of jsonic format string. Below are some examples, for some special response, I will explain how they are organized.

System Info

request:

```
{
    "metric_type": "system_info"
}
```

response:

repsonse

```
{
  "nodes_info": [
    {
      "identifier": 1, // unique in the list of nodes_info
      "name": "root coordinator",
      "hardware_info": {
        "ip": "192.168.1.1",
        "cpu_core_count": 2,
        "cpu_core_usage": "10%",
        "memory": "13124124",
        "memory_usage": "234123",
        "disk": "234123",
        "disk_usage": "123123",
      },
      "system_info": {
        "system_version": "rc2 a3c662c7b",
        "deploy_mode": "cluster",
      },
      "system_configurations": {
        "maxPartitionNum": 4096,
        "timeTickInterval": 200
      },
      "created_time": "2021-04-13 08:41:34.51+00",
      "updated_time": "2021-04-13 08:41:34.51+00",
      "type": "coordinator",
      "connected": []
    },
    {
      "identifier": 2,
      "name": "data coordinator",
      "hardware_info": {
        "ip": "192.168.1.1",
        "cpu_core_count": 2,
        "cpu_core_usage": "10%",
        "memory": "13124124",
        "memory_usage": "234123",
        "disk": "234123",
        "disk_usage": "123123",
      },
      "system_info": {
        "system_version": "rc2 a3c662c7b",
```

```
    "deploy_mode": "cluster",
  },
  "system_configurations": {
    "maxPartitionNum": 4096,
    "timeTickInterval": 200
  },
  "created_time": "2021-04-13 08:41:34.51+00",
  "updated_time": "2021-04-13 08:41:34.51+00",
  "type": "coordinator",
  "connected": [
    {
      "parent": 1,
      "method": "manage"
    }
  ]
},
{
  "identifier": 3,
  "name": "proxy",
  "hardware_info": {
    "ip": "192.168.1.1",
    "cpu_core_count": 2,
    "cpu_core_usage": "10%",
    "memory": "13124124",
    "memory_usage": "234123",
    "disk": "234123",
    "disk_usage": "123123",
  },
  "system_info": {
    "system_version": "rc2 a3c662c7b",
    "deploy_mode": "cluster",
  },
  "system_configurations": {
    "maxPartitionNum": 4096,
    "timeTickInterval": 200
  },
  "created_time": "2021-04-13 08:41:34.51+00",
  "updated_time": "2021-04-13 08:41:34.51+00",
  "type": "proxy",
  "connected": [
    {
      "parent": 1,
      "method": "notification"
    },
    {
      "parent": 2,
      "method": "notification"
    }
  ]
},
{
  "identifier": 4,
  "name": "data node 1",
  "hardware_info": {
    "ip": "192.168.1.1",
    "cpu_core_count": 2,
    "cpu_core_usage": "10%",
    "memory": "13124124",
    "memory_usage": "234123",
    "disk": "234123",
    "disk_usage": "123123",
  },
  "system_info": {
    "system_version": "rc2 a3c662c7b",
    "deploy_mode": "cluster",
  },
  "system_configurations": {
    "maxPartitionNum": 4096,
    "timeTickInterval": 200
  },
  "created_time": "2021-04-13 08:41:34.51+00",
```

```
      "updated_time": "2021-04-13 08:41:34.51+00",
      "type": "data node",
      "connected": [
        {
          "parent": 2,
          "method": "manage"
        }
      ]
    },
    {
      "identifier": 5,
      "name": "data node 2",
      "hardware_info": {
        "ip": "192.168.1.1",
        "cpu_core_count": 2,
        "cpu_core_usage": "10%",
        "memory": "13124124",
        "memory_usage": "234123",
        "disk": "234123",
        "disk_usage": "123123",
      },
      "system_info": {
        "system_version": "rc2 a3c662c7b",
        "deploy_mode": "cluster",
      },
      "system_configurations": {
        "maxPartitionNum": 4096,
        "timeTickInterval": 200
      },
      "created_time": "2021-04-13 08:41:34.51+00",
      "updated_time": "2021-04-13 08:41:34.51+00",
      "type": "data node",
      "connected": [
        {
          "parent": 2,
          "method": "manage"
        }
      ]
    }
  ]
}
```

In order to show the connection topology of Milvus, we have the `nodes_info` in response. `nodes_info` is a list and every item in list indicates a node in Milvus cluster. Every item has a identifier which is unique in `nodes_info`. The identifier can be used in `connected` content, for example, proxy has connected to root coordinator and data coordinator, so the `connected` content is `[1, 2]`, 1 is the identifier of root coordinator, 2 is the identifier of data coordinator.

System Statistics

```
{
    "metric_type": "system_statistics"
}
```

response:

```
{
    "hardware_statistics": [
        {
            "identifier": 1,    // unique in the list of hardware_statistics
            "name": "root coordinator",
            "hardware_usage": {
                "cpu": {
                    "type": "Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz",
                    "usage": 6
                },
                "memory": {
                    "total": 320000,    // in mega bytes
                    "usage": 120000,    // in mega bytes
                }
            }
        }
        // ...
    ],
    "loaded_collections": [
      {
        "name": "coll1",
        "loaded_time": "2021/07/05 11:13:44.372 +08:00",
        "loaded_partitions": [
          {
            "name": "partition1",
            "loaded_time": "2021/07/05 11:13:44.372 +08:00",
          },
          // ...
        ]
      },
      // ...
    ],
    "collection_count": 3,
    "partitions_count": [
      {
        "coll1": 2,
      },
      // ...
    ],
    "indexes_count": [
      {
        "coll1": 2,
      },
      // ...
    ],
    "qps": 10096,
    "latency": 0.1
}
```

System Event Log

```
{
    "metric_type": "system_log"
}
```

response:

```
{
    "dd": [
        "create collection 1 at ts1",
        "create collection 2 at ts2"
    ],
    "dm": [
        "insert 20 records into collection 1",
        "insert 30 records into collection 2"
    ],
    "dq": [
        "search on collection 1, nq: 10, topk = 5",
        "search on collection 2, nq: 10, topk = 5"
    ]
}
```

Test Plan

test script written with `pymilvus`:

```python
#!/usr/bin/env python

import ujson

from pymilvus.grpc_gen import milvus_pb2 as milvus_types

ip = "127.0.0.1"
port = "19530"

if __name__ == "__main__":
    client = Milvus(host=ip, port=port)

    with client._connection() as handler:
        system_info_req = ujson.dumps({"metric_type": "system_info"})
        req = milvus_types.GetMetricsRequest(request=system_info_req)
        resp = handler._stub.GetMetrics(req, wait_for_ready=True, timeout=None)
        print(resp)

        system_statistics_req = ujson.dumps({"metric_type": "system_statistics"})
        req = milvus_types.GetMetricsRequest(request=system_statistics_req)
        resp = handler._stub.GetMetrics(req, wait_for_ready=True, timeout=None)
        print(resp)

        system_logs_req = ujson.dumps({"metric_type": "system_logs"})
        req = milvus_types.GetMetricsRequest(request=system_logs_req)
        resp = handler._stub.GetMetrics(req, wait_for_ready=True, timeout=None)
        print(resp)

    client.close()
```