

MEP 2 -- Component startup and connection

Current state: Accepted

ISSUE: design issue [#6350](#). Fix [#5976](#) [#6098](#) [#6110](#) [#6236](#) .

PRs:

Keywords: Connection Manager, Component startup

Released:

Summary

Add a ConnectionManager to manage and monitor grpc Connection client. And with ConnectionManager, we will unified component startup process.

Motivation

We have faced a lot of problems caused by grpc connection. The current version of the grpc connection layer exposes fewer external parameters and is difficult to be controlled by the outer layer. And there is no reasonable way to handle the online and offline information of the session. This will cause a lot of problems listed as follows.

Problem:

1. The startup logic of each component is not uniform.
2. The method call of the grpc client cannot be stopped by client.stop(), and the total connection time is too long due to the execution of the serial grpc connection.
3. In the method recall process, the server can not know other servers' unregistration, leading to the low latency before a method fails.

Public Interfaces

```
type ConnectionManager struct {
```

```
    session *Session
```

```
    rootCoord  RootCoord
```

```
    queryCoord queryCoord
```

```
    dataCoord  DataCoord
```

```
    indexCoord IndexCoord
```

```
    queryNodes []QueryNode
```

```
    dataNodes  []DataNode
```

```
    indexNodes []IndexNode
```

```
}
```

```
AddDependency(serverName string) error
```

```
GetClients(serverName string) []interface
```

```
WatchClients(serverName string) []interface
```

Design Details

Goal:

1. The startup logic of each component is clear and unified, and the connection of the dependent service becomes necessary when calling the method instead of the startup process.
2. If the kv registered by the node fails on etcd, the related grpc request should fail quickly and inform the SDK or the initiator of the request;
3. After the failed node is restarted, the grpc link should be restored automatically. The automatic restoration process should be able to be appreciable and be non-blocking;

A new component ClientManager is responsible for monitoring the grpc client with other servers.

ClientManager:

1. Responsible for managing the establishment and update of all grpc Client connections.
2. ClientManager can obtain the current online session of etcd by adding dependencies, and monitor the changes of related keys to increase or decrease.
3. When an Add event is received, a goroutine is started to connect to grpc, and the local client list is updated after the connection. Upon receiving the Del event, delete the local existing client, and terminate the connection establishment operation and grpc call related to this Client.
4. ClientManager has an interface to return whether the object is online based on the NodeID or Coordinator service name.
5. A watch interface will be provided to notify a client is established or offline for future load balance.

The ClientManager is designed to be an interface and the standalone milvus can be implemented as calling methods directly instead of through grpc.

Grpc service check and timeout with ClientManager:

1. If there is a grpc call that needs to rely on a server, first get whether the grpc client connection of the server is available according to the ClientManager, and fail quickly if it is not available.
2. If it is available, for the grpc request, retry 3 times, each timeout period is 10 seconds, if it still times out after 3 times, a timeout error will be returned to the grpc.
3. If the server is offline in the process of grpc request, the ClientManager will stop the client for abort follow-up operations, the request will return an unavailable error to the caller.

Through this mechanism, we can achieve goals 2 and 3.

Compatibility, Deprecation, and Migration Plan

And with ClientManager, the startup process will be standardized as the following process.

Node startup process (distributed layer)

1. Initialize its own related components, such as creating the inner Node object, starting the grpc service, opening opentracing, etc.
2. Initialize the inner Node object

When creating a node object(internal layer), we need to initialize the state code of the node object for avoiding illegal access to state code and grpc calls.

Node startup process (internal layer)

For nodes, the services that need to be connected are coordinators. In the current version, there is only one coordinator with the same role.

1. Initialize state code when NewServer().
2. Use ClientManager to add Coord service dependency. Taking ProxyNode as an example, adding a dependency on rootCoordinator, ClientManager will obtain the required rootCoordinator service address from etcd. If the rootCoordinator service has been registered and the address is successfully obtained, a grpc connection will be established. If the grpc connection can not be established, then enter the retry logic, retry 3 times, each 30 seconds, and still fail after 3 retry attempts, then give up retrying, but do not panic. Mark rootCoordinator offline, and expect to re-trigger the link to rootCoordinator through etcd's Add event.
3. Register its own service address with etcd from its own session
4. Set statecode to Healthy

Coordinator startup process (distributed layer)

1. Initialize its own related components, such as creating the inner Coordinator object, starting the grpc service, opening opentracing, etc.
2. Initialize the inner Coordinator object

When creating a Coordinator object(internal layer), we need to initialize the state code of the node object for avoiding illegal access to state code and grpc calls.

Coordinator startup process (internal layer)

For the Coordinator node, it is necessary to connect to the services of multiple nodes, and these nodes may go online and offline frequently, and at the same time, avoid connecting inactive Nodes. At the same time, the Coordinator node may also depend on other coordinators.

1. Initialize state code when NewServer().
2. ClientManager adds Coordinator service dependency. Take QueryCoord as an example to add a dependency on RootCoordinator. ClientManager obtains the required RootCoordinator service address from etcd. If the service has been registered and the address is successfully obtained, the grpc connection is established. If the grpc connection can not be established, then enter the retry logic, retry 3 times, each 30 seconds, and still fail after 3 retries, then give up and retry, but do not panic, mark RootCoordinator offline, and expect to re-trigger the link to RootCoordinator through etcd Add event in the future.
3. The ClientManager processes the services of the Node that already exists in the meta. Taking QueryCoord as an example, ClientManager obtains the service address of the existing QueryNode from etcd, and opens a separate coroutine for each QueryNode to process it. The

processing logic is as follows: Get the address, try to establish a grpc link, if the grpc connection can not be established, retry 3 times, each 30 seconds, if the connection fails, give up processing, if the connection is successful, create the corresponding Client, and add the Client to the ClientManager.

4. Register its own service address with etcd from its own session
5. Set statecode to Healthy

Test Plan

The unittest will be added to test the ClientManager to test whether the connection can be reacted quickly.
And the recovery test will be tested with existing problem issue [#5976](#) [#6098](#) [#6110](#) [#6236](#).