DigiTwin Messaging Service

V1.6

Contents

Introduction

DigiTwin Messaging Service aims to help IoT devices or any other private cloud endpoints to communicate with each other or the backend as required with the help of inexpensive IRC APIs. This will not only help in secure communication between the entities but also, have multiple entities as the project requires.

The DigiTwin Messaging Service (DMS), is quite simple to use. There are a few parts for this product along with a supervision Client

1. DMS-Cloud-Client
2. DMS-Edge Client
3. DMS-Server
4. DMS-Viewer
5. DMS-Listener

The DMS is distributed as a docker image. Within the docker image we have all the required parts of the required for the entire DMS topology.

The Typical setup is shown in Figure 1.

The DMS-Server is actually an IRC server that facilitates real-time text communication between users over the Internet. It operates as the central hub in the IRC network, managing connections from clients, routing messages, and hosting channels (chat rooms) for group discussions. Users connect via IRC clients using a server's IP address or domain. IRC servers are lightweight, customizable, and can support numerous simultaneous users. Popular for community discussions, gaming, and technical support, they form decentralized networks where multiple Servers link to enable larger-scale communication across distributed groups worldwide.

The DMS cloud-client and the edge-clients are clients in IRC that are applications that allow users to connect to an IRC server for real-time communication. Using the configurations to connect to the DMS-Server, these clients would be able to send as well as received messages. The messages can in-turn be used for later processing by other applications as per logic.

The figure below describes the communication between Cloud-Client and the Edge-Clients from within VPN as well as outside in the cloud.

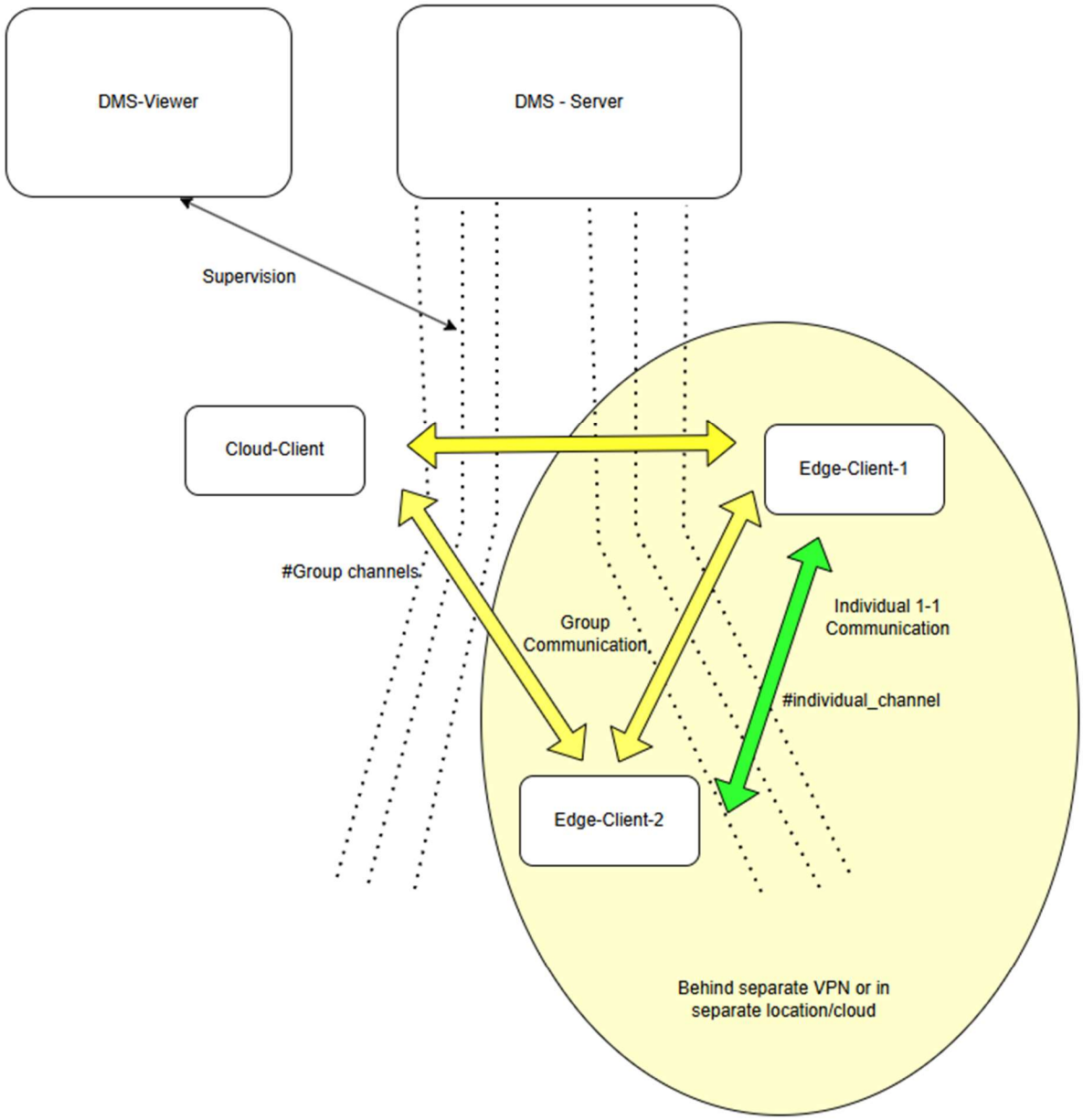

DMS-Viewer
DMS - Server
Supervision
Cloud-Client
Edge-Client-1
#Group channels
Group
Communication
Individual 1-1
Communication
#individual_channel
Edge-Client-2
Behind separate VPN or in
separate location/cloud

Figure 1

The communication occurs through server via channels. Users connect to an IRC server using a client and join channels, which are virtual chat rooms identified by names starting with `#` (e.g., `#general`). Channels host group discussions, while private messaging allows one-on-one chats. Messages sent in a channel are broadcast to all participants, fostering real-time conversations. Channels can be public, private (accessible by invitation), or password-protected. The decentralized nature allows multiple servers to link, forming networks. Several terms like clients and channels will be used later in the documentation as well.

Note: DMS is supported only on docker as of now.

Contents of the DMS docker

The DMS docker is provided on AWS as well as on our Baideac license portal. Upon downloading this docker image and logging in, the following contents can be found:

1. **DMS-Cloud-Client** - This is an executable that is Run when the docker container is running as a cloud client. This can be done based on the configurations in the .env file
2. **DMS-Edge-Client** - This is an executable that is Run when the docker container is running as an edge client. This can be done based on the configurations in the .env file
3. **DMS-Listener** - This is the folder containing a sample application that simply prints the messages routed by the DMS clients on to the console. Inside this folder there is an executable.
4. **DMS-Server** - This contains an executable that will install the DMS-Server. It also contains a .env file which can be edited by hand. This file is filled when the DMS-Server executable is Run and the prompts are filled. (more on this later)
5. **DMS-Viewer** - Contains an executable that will install the DMS-Viewer. This is a standard supervision tool that can be used by any user to see that Communications taking place between the clients
6. **channels.json** - This is a file provides a list of the channels which a client must join at startup. This file also contains routes per channel. More on this later)
7. **licenseHandler** - This is an executable that simply helps in verifying the licence keys
8. **start.sh/restart.sh** - These are scripts which help in starting a restarting the executables when a change is made in the .env file
9. **.env** - This is the configuration file in which all the settings are present. (more on this later)

As it can be seen that that all the components in a typical DMS topology is present inside the single docker, we will need to copy the different non client contents outside this docker and set up in the respective host. As a described in the Figure 1, in an ideal scenario, there would be a host each for the following

1. The DMS cloud client (Currently having multiple cloud clients in a single topology is not supported in this release)
2. Each DMS edge clients
3. DMS server

The DMS listener, if used, must be in the respective clients' host such that it is able to print out the messages on the console.

The DMS viewer can be present in a separate host(s), or can be in any of the host machines among the clients are the DMS server.

To be able to access these files, the docker image must be first run. It is assumed that the first docker container would be for the DMS-Cloud-Client. The command to run this is

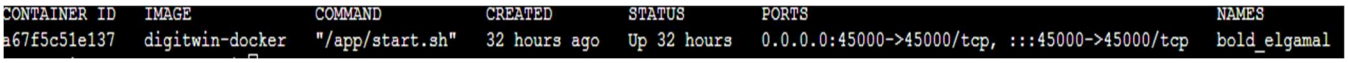
sudo docker run -d --add-host=host.docker.internal:host-gateway -p 45000:45000 digitwin-docker

The description of the attributes are as follows:

- **--add-host=host.docker.internal:host-gateway** - this provides a way to send data from within the docker to the host. This is important and should not be omitted.
- **-p 45000:45000** - this tells the port mapping of the docker to the host machine. This is present in the format of < port of the host machine which is listening for any connection request to be routed to the docker>:< the exposed port from within the docker container>
- **digitwin-docker** - this is the name of the docker image

This will start the docker. One can access the docker container ID with the following command

sudo docker ps



The above image shows the response.

The docker must be logged into such that the configurations can be made. The command to log inside the docker is as follows

sudo docker exec -it <docker container-id> bash

This will take you straight inside the docker. Basic ubuntu commands will work here. The above folders structure can be found in here.

To set up the entire topology, we must setup clients and all the other entities. Thus, as the first step we must start configuring the DMS-Server and the DMS-Viewer.

For this, the DMS-Server and the DMS-Viewer folders can be copied to the host machine from within the docker with the following command.

sudo docker cp <container-id>:/app/DMS-Server .
sudo docker cp <container-id>:/app/DMS-Viewer .

This will copy the DMS-Server and the DMS-Viewer to the host machine. This can now be copied to a different host. Let's start configuring the DMS-Server

DMS-Server Configuration

The DMS-Server Folder has 1 single executable: DMS-Server. Upon executing this application, all the necessary package including the docker is installed. Finally, it asks for 3 inputs to start the server

1. URL : This is the public available DNS or IP that is required to connect to this server. It must accessible from anywhere.
2. Port : The port on which the Server will accept connections
3. Password : The password is required to prevent unauthorized connections to the server. This Password must be used by all the clients who are connect to this Server.

Once executed, the server will start running. Now, any client with the correct credentials can connect to the DMS-Server that we set.

Note: As this is an IRC server following the protocols of IRC, a user may choose to set up another IRC server and connect the clients. While this is completely possible, Baideac Support Team will not be able to support any other IRC server.

For more help, please have check out the video on the site.

Now before connecting any client, we must check if the DMS-Server is correctly set up or not. For this, we can easily set up the DMS-Viewer.

DMS-Viewer

Many a times, one may need to supervise how the entities are communicating with each other. To supervise the communications, one can quite easily refer to any opensource IRC client. With

some easy steps, it is quite simple to connect with the servers and join any new channel to supervise.

Baideac is providing an out of the box client for anyone's usage without the hassle of downloading and installing a new client.

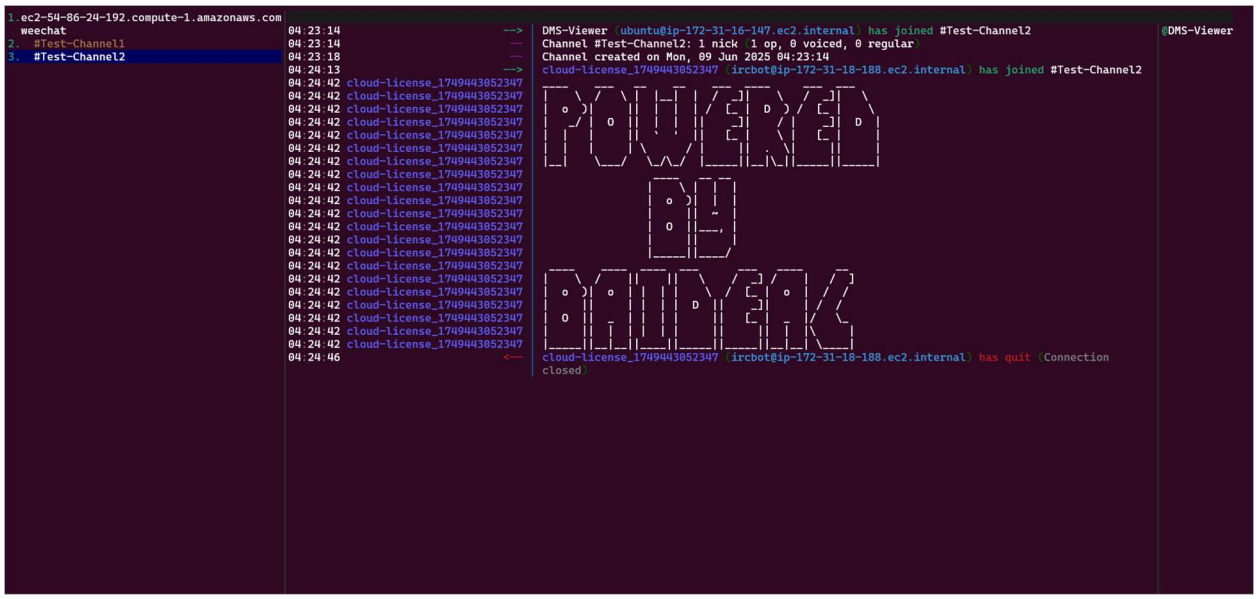
The steps to configure are quite easy. Simply copy the folder DMS-Viewer folder to the host and execute the application.

At the time of execution, all the required packages will be installed along with the required items including port, nickname, URL as well as the password. It will ask for channels as well to connect to at startup. Note, the channels cannot be blank. (Please note: the length of the nickname must no exceed 40 characters. If it is, the name would be truncated)

Once, the DMS-Viewer runs, one will see a UI loading with the previous clients also showing up.

That is all!! you would now be able to use commands "/join #channel=name" to connect to any other channel as well and supervise the communication.

At anytime, one may use "/quit" or /exit to close the application.



Now that we can be assured that the DMS-Server is running correctly, let us configure the DMS-Cloud-Client.

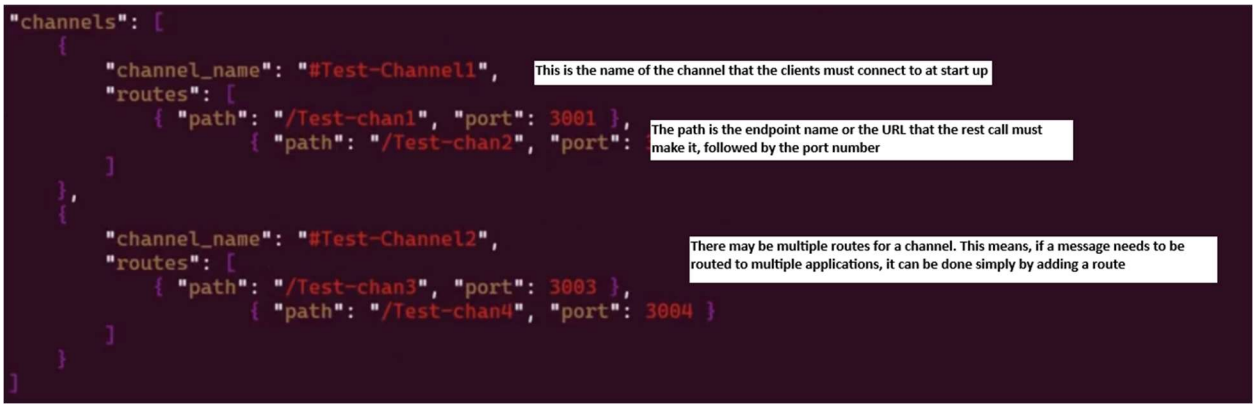
Setting up the DMS-Cloud-Client

The DMS-Cloud-Client is logically supposed be the brain in the topology. In an ideal scenario, this client would be sending commands to the edge clients. As mentioned earlier, currently having more than one cloud client in the same topology is not possible, and if set up, it may lead to unknown behaviours. The DMS-Cloud-Client can send messages it has received over a rest call. This rest call can be from outside the docker.

Before running, we must understand the different configurations. As we know, in the IRC Server, one can log in to different channel for communications, be it a one-to-one or a broadcast channel for communication with a larger set of clients, we have provided a Json file named "channel.json" to provide this list of channels which the executable must join at start up. Upon receiving a message on a certain channel, the messages can also be routed to other endpoints as required. These routes can be found in the same Json file.

We will have more explanations in the "How DMS works" section.

The json file looks like this



Under channels, we define all the channels the IRC client must join at startup. Each channel has routes with the endpoint name and the port which are added such that, any message coming on the channels will be routed automatically there.

Now we must configure the *.env* file to configure which IRC server to connect to, the port on which it must listen for the requests and others.

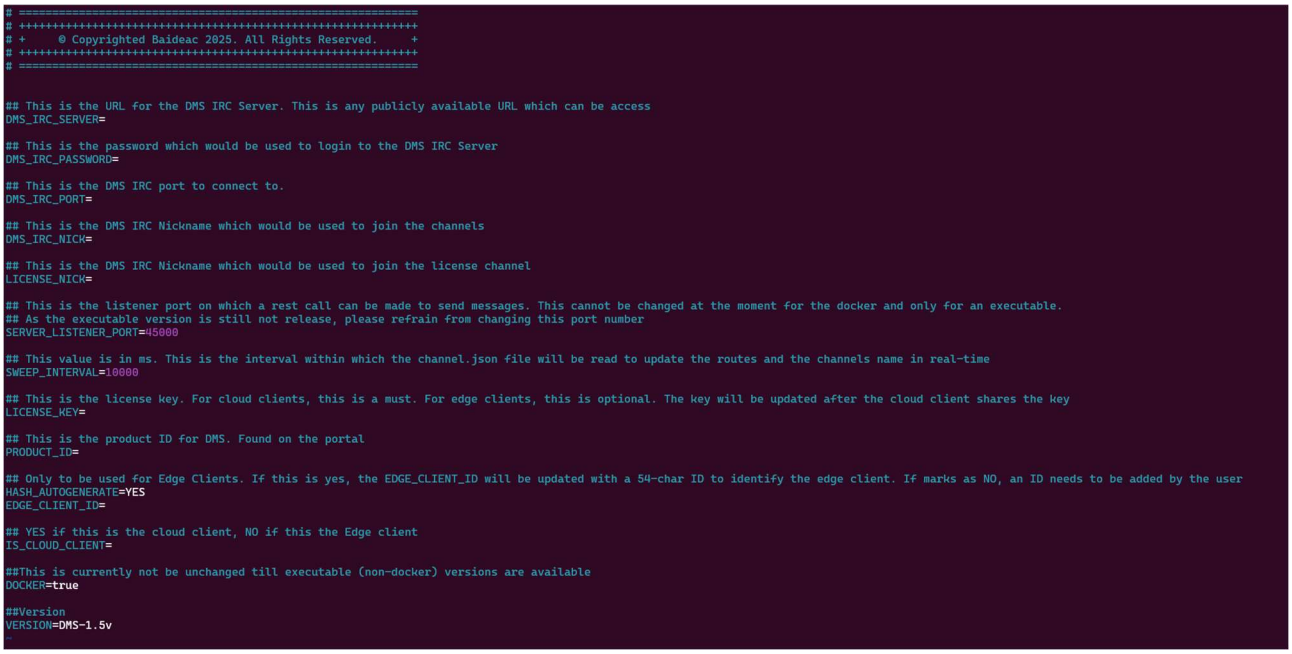
The fields to be filled for the DMS-Cloud-Client env file are:

1. **DMS_IRC_SERVER**= <the IRC Server URL>
2. **DMS_IRC_PASSWORD**=<Password if any, if no password is there, leave this blank>
3. **DMS_IRC_PORT**=<The IRC port to connect to. This is defined while setting up the DMS-Server>
4. **DMS_IRC_NICK**=<the IRC nickname that will appear in the IRC channel. It must be unique such the one can distinguish between other devices>
5. **SERVER_LISTENER_PORT**= this is the listening port where one can send their requests to send a message to the DMS-Server. (more on this in the "How DMS works" section). Currently, in the docker version, this is not possible to change. We request you to refrain from making any change here.
6. **SWEEP_INTERNAL**=Sweep interval within which the channel.json file will be parsed for updates on the new additional channels or routes (More on this in the "How DMS works Sections")
7. **LICENSE_KEY**= This is the license key that can be fetched from the Baideac license portal. For the cloud client, this is a must
8. **PRODUCT_ID**= This is the Product ID that can be fetched from the Baideac license portal. For the any client, this is a must.
9. **HASH_AUTOGENERATE**= For the cloud client, this is not required
10. **EDGE_CLIENT_ID**= For the cloud client, this is not required
11. **IS_CLOUD_CLIENT**= This is a flag that denotes whether this client is a cloud Client or an edge client. Since we are setting up a cloud client we must market as YES
12. **DOCKER**= For now, since there is no executable version, mark this as "true"
13. **VERSION=** The version number of this version of DMS

In order to check the version from outside the docker, is the following command.

sudo docker exec -it <container ID> cat .env | grep VERSION

A typical .env file looks like this



Upon saving this .env file, the cloud client will automatically join the IRC network. This will be visible in the DMS-Viewer in the different channels.

Setting up the DMS-Edge-Client

The DMS-Edge-Client is logically supposed to be the dumb endpoints in the topology. They can be imagined as slaves to the master which is the DMS-Cloud-Client. The behaviour is same as the DMS-Cloud-Client. The configuration is quite similar as well with minor differences. This client also reads the channels.json file as well as the .env file.

The fields to be filled for the DMS-Edge-Client .env file are:

1. **DMS_IRC_SERVER**= <the IRC Server URL>
2. **DMS_IRC_PASSWORD**=<Password if any, if no password is there, leave this blank>
3. **DMS_IRC_PORT**=<The IRC port to connect to. This is defined while setting up the DMS-Server>
4. **DMS_IRC_NICK**=<the IRC nickname that will appear in the IRC channel. It must be unique such the one can distinguish between other devices>
5. **SERVER_LISTENER_PORT**= this is the listening port where one can send there requests to send a message to the DMS-Server. (more on this in the “How DMS works“

section). Currently, in the docker version, this is not possible to change. We request you to refrain from making any change here.

6. **SWEEP_INTERNAL**=Sweep interval within which the channel.json file will be parsed for updates on the new additional channels or routes (More on this in the "How DMS works" Sections)
7. **LICENSE_KEY**= This is not mandatory as the key will be fetched at startup as well as every day. The cloud client will also send this key when the cloud client starts up
8. **PRODUCT_ID**= This is the Product ID that can be fetched from the Baideac license portal. For the any client, this is a must.
9. **HASH_AUTOGENERATE**= If marked as YES, a 54-character client ID will be generated and assigned to the EDGE_CLIENT_ID field. If the user wants to put in their own EDGE_CLIENT_ID, they can mark it as NO. However, in this case, the EDGE_CLIENT_ID must be filled
10. **EDGE_CLIENT_ID**= This is the ID by which logically and its client can be identified. This is done to prevent confusions between the same named edge clients
11. **IS_CLOUD_CLIENT**= This is a flag that denotes whether this client is a cloud Client or an edge client. Since we are setting up an Edge client we must market as NO
12. **DOCKER**= For now, since there is no executable version, mark this as "true"
13. **VERSION=** The version number of this version of DMS

In order to check the version from outside the docker, is the following command.

sudo docker exec -it <container ID> cat .env | grep VERSION

Upon saving this .env file, the cloud client will automatically join the IRC network. This will be visible in the DMS-Viewer in the different channels.

It must be noted that by default only 20 edge clients will be supported by the cloud client as per First Come First serve basis. Any client that joins after 20 edge clients are signed up with the cloud client, will be rejected by the cloud client. To increase this limit, the user must request the support team at support@baideac.com . The user will receive a new key with the updated limited.

The way to delete any edge client which is not required anymore manually is under process and will be present in the upcoming releases. The only way to reset count at the cloud client end is to restart the cloud client. The edge clients will automatically connect themselves to the cloud client, and edge client ID which needs to be removed will no longer get added. Still, at no point there would be more than limit set for the edge clients trying to connect to the cloud client.

Now, just set up is complete let us now exchange some messages.

How DMS works

The DMS Clients at runtime, can fetch edits as required for changing the routes or port numbers. This is done so that new routes can be added on the go without having to stop the Clients. These changes can be done at the *channel.json* file. The SWEEP_ INTERNAL field present in the .env file can be configured in milliseconds such that, within this interval the client will sweep through the channel.json file to fetch the updates. This way, at runtime, for application adding new channels or routes at runtime can be easily accommodated.

The client when running, listens on the SERVER_ LISTENER _PORT preconfigured as 45000. One can send the message via a rest call to this server port.

The format differs slightly for the different clients.

For the DMS-Cloud-Client, it is

```
curl --location 'http://localhost:45000/notify' \
--header 'Content-Type: application/json' \
--data '{
    "message": "Your Message",
    "channel":"#your-channel",
    "edgeclientId" : "the generated/assigned edge client ID"
}'
```

We need to add the edge client ID since when the cloud client is sending a message, the cloud client requires to know who to send the message to

For the DMS-Edge-Client, it is

```
curl --location 'http://localhost:45000/notify' \
--header 'Content-Type: application/json' \
--data '{
    "message": "Your Message",
    "channel":"#your-channel"
}'
```

The port 45000 is SERVER_ LISTENER _PORT.
Under the data field, we have
- Message: the message one needs to send. It can be off any size. However, tests have been done only up to 20KB of data. (sending large files is not supported in the current version)
- channel: This is the target channel where the Sender needs to send the message. The "#" is a must.
- edgeclientId – (Only for cloud client). This ID is used to identify the different edge clients

.
Now, let's walk through how can a client communicate with another client.

Let's considered that the **cloud client** wants to send a message to the **edge client**. Assuming that the clients have the same channels.json file, they will join in the same channels. For the sake of this example, we will consider the channel "#poc_test" (the '#' is a requirement always at the start of the channel name).

A rest call with any script (or Postman) can be created as below

```
curl --location 'http://localhost:45000/notify' \
--header 'Content-Type: application/json' \
--data '{
   "message": "Hello World : This is a test",
   "channel": "#Test-Channel1",
   "edgeclientId" : "34198j9813n4dnjnckxjnwksjnckjner8vjnerc"
}'
```

The rest call is done towards the DMS-Cloud-Client. The message would be forwarded to the IRC Server. (the Server details are already mentioned in the .env).

The edge client, at startup already is subscribed to the #poc_test channel as it is mentioned in the channel.json file. The DMS-Edge-Client receives the message and routes this message to any listening endpoint as per the channels.json file.

If the message needs to be routed to another services, simply add the route as shown above in the channel.json example. The message will be forwarded to that service automatically.

Now sending in the opposite way is almost similar, accept that the rest call body will not contain the edgeclientId field. It looks something like this

```
curl --location 'http://localhost:45000/notify' \
```


```
--header 'Content-Type: application/json' \
--data '{
   "message": "Hello World : This is a test",
   "channel": "# Test-Channel1”
}'
```

For more help, refer to the demo in the product page.

DMS-Listener

Baideac is providing a demo application which will double as a test endpoint. It simply takes in 2 inputs, the port and the endpoint name. Once done, this will start and listen for messages coming in from the DMS Clients.

As mentioned before, in the channel.json file, we mentioned the routes the messages will take when arriving on a channel. When a message arrives on the DMS Clients, it will route the messages to this DMS-Listener. The DMS-Listener will simply print out the message on the console.

Note: This DMS—Listener must be on the same host as the DMS Clients. The configuration of the DMS-Listener must be as per the channel.json config. For more details, please check the product video.

For any further help on any topics, please reach out to us.

How the licensing works

One of the most significant aspects of the product is the licensing. The license key works in tandem with the product ID. Both these items can be found in the Baideac licensing portal.

Both the type of clients checks the validity of the license key. However, their checks are done slightly differently.

The DMS-Cloud-Client must have the license key, otherwise it will not start. At startup, if the license key is valid, it will broadcast the license key in the #license-key channel. This is done such that, if there is a new license key, all the edge clients must be updated with it. Every day, the licence key is checked. Only if its valid, the DMS-Cloud-Client continuous to run.

The DMS-Edge-Client does not need to have the license key at startup, however, if there is no license key, the edge client will keep requesting for the license key in the #license-key channel. When the DMS-Cloud-Client receives this request, immediately it responds to that edge client with the license key. A typical exchange can be seen below.

If there is no license key at the DMS-Edge-Client end, the DMS-Edge-Client will reject all request to send messages to the IRC channels until the license key is successfully validity.

Both the clients check for the license keys validity every day.

References:

For IRC Related information https://www.geeksforgeeks.org/internet-relay-chat-irc/

For DMS-Viewer related information : https://weechat.org/