KubeEdge Documentation

Release 0.1

KubeEdge

Mar 25, 2019

Contents

KubeEdge is an open source system for extending native containerized application orchestration capabilities to hosts at Edge.

CHAPTER 1

Welcome to KubeEdge

KubeEdge is an open source system for extending native containerized application orchestration capabilities to hosts at Edge.

1.1 Why KubeEdge?

Learn about KubeEdge and the KubeEdge Mission here

1.2 First Steps

To get the most out of KubeEdge, start by reviewing a few introductory topics:

- Setup - Install KubeEdge
- Integrate with IEF - Integrate with the Intelligent Edge Fabric cloud
- Contributing - Contribute to KubeEdge
- Troubleshooting - Troubleshoot commonly occurring issues. GitHub issues are here

CHAPTER 2

How to contribute

Kubeedge is Apache 2.0 licensed and accepts contributions via GitHub pull requests. This document outlines some of the conventions on commit message formatting, contact points for developers, and other resources to help get contributions into kubeedge.

2.1 Email and chat

- Email: kubeedge
- Slack: kubeedge

2.2 Getting started

- Fork the repository on GitHub
- Read the setup for build instructions

2.3 Reporting bugs and creating issues

Reporting bugs is one of the best ways to contribute. However, a good bug report has some very specific qualities, so please read over our short document on reporting bugs before submitting a bug report. This document might contain links to known issues, another good reason to take a look there before reporting a bug.

2.3.1 Contribution flow

This is a rough outline of what a contributor's workflow looks like:

- Create a topic branch from where to base the contribution. This is usually master.

- Make commits of logical units.
- Make sure commit messages are in the proper format (see below).
- Push changes in a topic branch to a personal fork of the repository.
- Submit a pull request to kubeedge/kubeedge.
- The PR must receive an approval from two maintainers.

Thanks for contributing!

2.3.2 Code style

The coding style suggested by the Golang community is used in kubeedge. See the style doc for details.

Please follow this style to make kubeedge easy to review, maintain and develop.

2.3.3 Format of the commit message

We follow a rough convention for commit messages that is designed to answer two questions: what changed and why. The subject line should feature the what and the body of the commit should describe the why.

```
scripts: add test codes for metamanager

this add some unit test codes to imporve code coverage for metamanager

Fixes #12
```

The format can be described more formally as follows:

```
<subsystem>: <what changed>
<BLANK LINE>
<why this change was made>
<BLANK LINE>
<footer>
```

The first line is the subject and should be no longer than 70 characters, the second line is always blank, and other lines should be wrapped at 80 characters. This allows the message to be easier to read on GitHub as well as in various git tools.

CHAPTER 3

Roadmap

3.1 Release 1.0

KubeEdge will provide the fundamental infrastructure and basic functionalities for IOT/Edge workload. This includes:

- K8s Application deployment through kubectl from Cloud to Edge node(s)
- K8s configmap, secret deployment through kubectl from Cloud to Edge node(s) and their applications in Pod
- Bi-directional and multiplex network communication between Cloud and edge nodes
- K8s Pod and Node status querying with kubectl at Cloud with data collected/reported from Edge
- Edge node autonomy when its getting offline and recover post reconnection to Cloud
- Device twin and MQTT protocol for IOT devices talking to Edge node

3.2 Release 2.0 and Future

- Build service mesh with KubeEdge and Istio
- Enable function as a service at Edge
- Support more types of device protocols to Edge node such as AMQP, BlueTooth, ZigBee, etc.
- Evaluate and enable super large scale of Edge clusters with thousands of Edge nodes and millions of devices
- Enable intelligent scheduling of apps. to large scale of Edge nodes
- etc.

CHAPTER 4

Support

If you need support, start with the troubleshooting guide, and work your way through the process that we've outlined.

4.1 Community

Slack channel:

We use Slack for public discussions. To chat with us or the rest of the community, join us in the KubeEdge Slack team channel #general. To sign up, use our Slack inviter link here.

Mailing List

Please sign up on our mailing list

CHAPTER 5

What is KubeEdge

KubeEdge is an open source system extending native containerized application orchestration and device management to hosts at the Edge. It is built upon Kubernetes and provides core infrastructure support for networking, application deployment and metadata synchronization between cloud and edge. It also supports MQTT and allows developers to author custom logic and enable resource constrained device communication at the Edge. Kubeedge consists of a cloud part and an edge part. Both edge and cloud parts are now opensourced.

5.1 Advantages

The advantages of Kubeedge include mainly:

- **Edge Computing**

  With business logic running at the Edge, much larger volumes of data can be secured & processed locally where the data is produced. This reduces the network bandwidth requirements and consumption between Edge and Cloud. This increases responsiveness, decreases costs, and protects customers' data privacy.

- **Simplified development**

  Developers can write regular http or mqtt based applications, containerize these, and run them anywhere - either at the Edge or in the Cloud - whichever is more appropriate.

- **Kubernetes-native support**

  With KubeEdge, users can orchestrate apps, manage devices and monitor app and device status on Edge nodes just like a traditional Kubernetes cluster in the Cloud

- **Abundant applications**

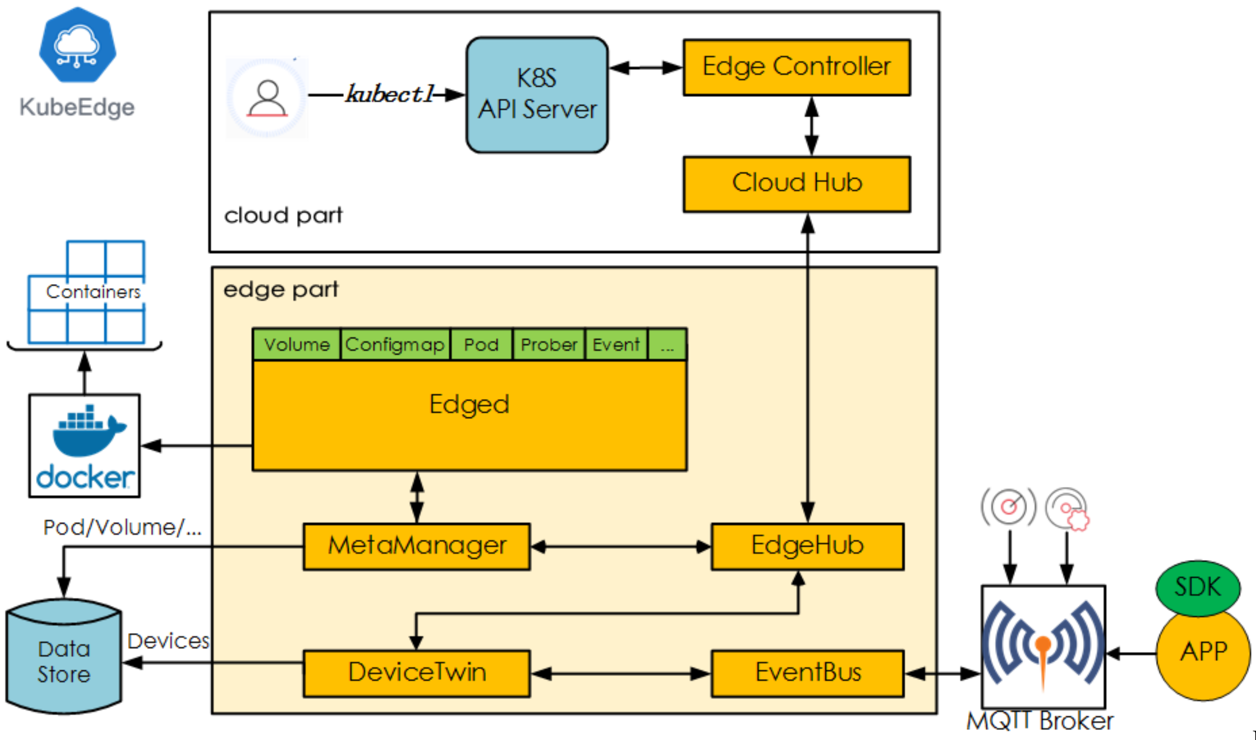
  It is easy to get and deploy existing complicated machine learning, image recognition, event processing and other high level applications to the Edge.

5.2 Components

KubeEdge is composed of these components:

- **Edged:** an agent that runs on edge nodes and manages containerized applications.
- **EdgeHub:** a web socket client responsible for interacting with Cloud Service for edge computing (like Edge Controller as in the KubeEdge Architecture). This includes syncing cloud-side resource updates to the edge and reporting edge-side host and device status changes to the cloud.
- **CloudHub:**: A web socket server responsible for watching changes at the cloud side, caching and sending messages to EdgeHub.
- **EdgeController**: an extended kubernetes controller which manages edge nodes and pods metadata so that the data can be targeted to a specific edge node.
- **EventBus:** an MQTT client to interact with MQTT servers (mosquitto), offering publish and subscribe capabilities to other components.
- **DeviceTwin:** responsible for storing device status and syncing device status to the cloud. It also provides query interfaces for applications.
- **MetaManager:** the message processor between edged and edgehub. It is also responsible for storing/retrieving metadata to/from a lightweight database (SQLite).

5.3 Architecture



KubeEdge

Architecture

5.4 Getting involved

There are many ways to contribute to Kubeedge, and we welcome contributions!

Read the contributor's guide to get started on the code.

CHAPTER 6

Beehive

6.1 Beehive Overview

Beehive is a messaging framework based on go-channels for communication between modules of KubeEdge. A module registered with beehive can communicate with other beehive modules if the name with which other beehive module is registered or the name of the group of the module is known. Beehive supports following module operations:

1. Add Module
2. Add Module to a group
3. CleanUp (remove a module from beehive core and all groups)

Beehive supports following message operations:

1. Send to a module/group
2. Receive by a module
3. Send Sync to a module/group
4. Send Response to a sync message

6.2 Message Format

Message has 3 parts

1. Header:
    1. ID: message ID (string)
    2. ParentID: if it is a response to a sync message then parentID exists (string)
    3. TimeStamp: time when message was generated (int)
    4. Sync: flag to indicate if message is of type sync (bool)
2. Route:

   1. Source: origin of message (string)
   2. Group: the group to which the message has to be broadcasted (string)
   3. Operation: what's the operation on the resource (string)
   4. Resource: the resource to operate on (string)

3. Content: content of the message (interface{})

6.3 Register Module

1. On starting edge_core, each module tries to register itself with the beehive core.
2. Beehive core maintains a map named modules which has module name as key and implementation of module interface as value.
3. When a module tries to register itself with beehive core, beehive core checks from already loaded modules.yaml config file to check if the module is enabled. If it is enabled, it is added in the modules map or else it is added in the disabled modules map.

6.4 Channel Context Structure Fields

6.4.1 (*Important for understanding beehive operations*)

1. **channels:** channels is a map of string(key) which is name of module and chan(value) of message which will used to send message to the respective module.
2. **chsLock:** lock for channels map
3. **typeChannels:** typeChannels is is map of string(key)which is group name and (map of string(key) to chan(value) of message) (value) which is map of name of each module in the group to the channels of corresponding module.
4. **typeChsLock:** lock for typeChannels map
5. **anonChannels:** anonChannels is a map of string(parentid) to chan(value) of message which will be used for sending response for a sync message.
6. **anonChsLock:** lock for anonChannels map

6.5 Module Operations

6.5.1 Add Module

1. Add module operation first creates a new channel of message type.
2. Then the module name(key) and its channel(value) is added in the channels map of channel context structure.
3. Eg: add edged module

```
coreContext.Addmodule("edged")
```

6.5.2 Add Module to Group

1. addModuleGroup first gets the channel of a module from the channels map.
2. Then the module and its channel is added in the typeChannels map where key is the group and in the value is a map in which (key is module name and value is the channel).
3. Eg: add edged in edged group. Here 1st edged is module name and 2nd edged is the group name.

```
coreContext.AddModuleGroup("edged","edged")
```

6.5.3 CleanUp

1. CleanUp deletes the module from channels map and deletes the module from all groups(typeChannels map).
2. Then the channel associated with the module is closed.
3. Eg: CleanUp edged module

```
coreContext.CleanUp("edged")
```

6.6 Message Operations

6.6.1 Send to a Module

1. Send gets the channel of a module from channels map.
2. Then the message is put on the channel.
3. Eg: send message to edged.

```
coreContext.Send("edged",message)
```

6.6.2 Send to a Group

1. Send2Group gets all modules(map) from the typeChannels map.
2. Then it iterates over the map and sends the message on the channels of all modules in the map.
3. Eg: message to be sent to all modules in edged group.

```
coreContext.Send2Group("edged",message) message will be sent to all modules in edged␣
↪group.
```

6.6.3 Receive by a Module

1. Receive gets the channel of a module from channels map.
2. Then it waits for a message to arrive on that channel and returns the message. Error is returned if there is any.
3. Eg: receive message for edged module

```
msg, err := coreContext.Receive("edged")
```

6.6.4 SendSync to a Module

1. SendSync takes 3 parameters, (module, message and timeout duration)
2. SendSync first gets the channel of the module from the channels map.
3. Then the message is put on the channel.
4. Then a new channel of message is created and is added in anonChannels map where key is the messageID.
5. Then it waits for the message(response) to be received on the anonChannel it created till timeout.
6. If message is received before timeout, message is returned with nil error or else timeout error is returned.
7. Eg: send sync to edged with timeout duration 60 seconds

```
response, err := coreContext.SendSync("edged",message,60*time.Second)
```

6.6.5 SendSync to a Group

1. Get the list of modules from typeChannels map for the group.
2. Create a channel of message with size equal to the number of modules in that group and put in anonChannels map as value with key as messageID.
3. Send the message on channels of all the modules.
4. Wait till timeout. If the length of anonChannel = no of modules in that group, check if all the messages in the channel have parentID = messageID. If no return error else return nil error.
5. If timeout is reached,return timeout error.
6. Eg: send sync message to edged group with timeout duration 60 seconds

```
err := coreContext.Send2GroupSync("edged",message,60*time.Second)
```

6.6.6 SendResp to a sync message

1. SendResp is used to send response for a sync message.
2. The messageID for which response is sent needs to be in the parentID of the response message.
3. When SendResp is called, it checks if for the parentID of response message , there exists a channel is anonChannels.
4. If channel exists, message(response) is sent on that channel.
5. Or else error is logged.

```
coreContext.SendResp(respMessage)
```

CHAPTER 7

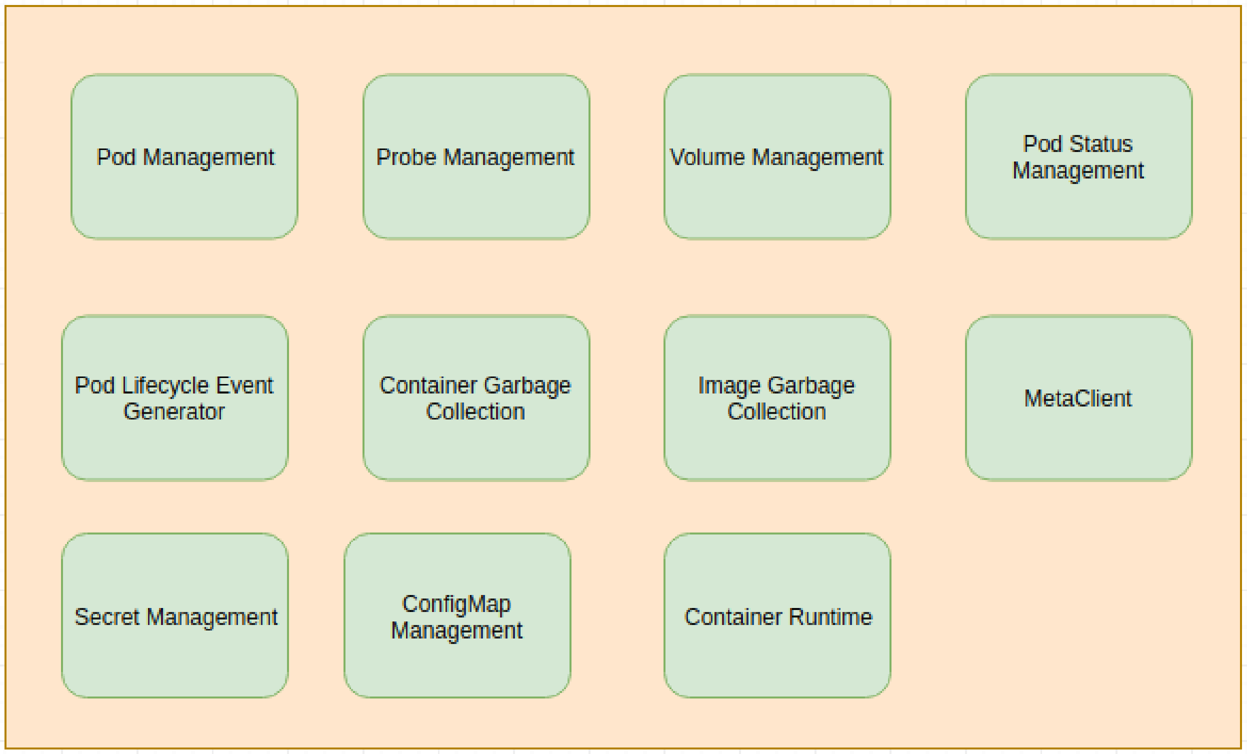
EdgeD

7.1 Overview

EdgeD is an edge node module which manages pod lifecycle. It helps user to deploy containerized workloads or applications at the edge node. Those workloads could perform any operation from simple telemetry data manipulation to analytics or ML inference and so on. Using `kubectl` command line interface at the cloud side, user can issue commands to launch the workloads.

Docker container runtime is currently supported for container and image management. In future other runtime support shall be added, like containerd etc.,

There are many modules which work in tandom to achive edged's functionalities.



EdgeD

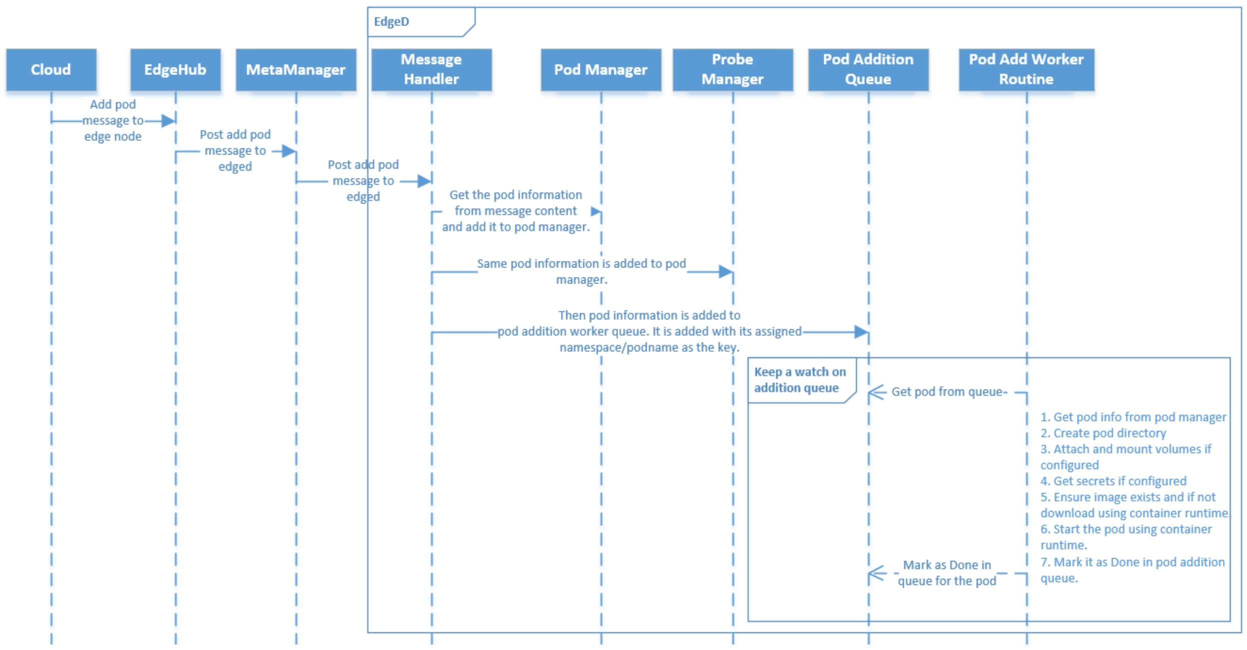
OverAll

Fig 1: EdgeD Functionalities

7.2 Pod Management

It is handles for pod addition, deletion and modification. It also tracks the health of the pods using pod status manager and pleg. Its primary jobs are as follows:

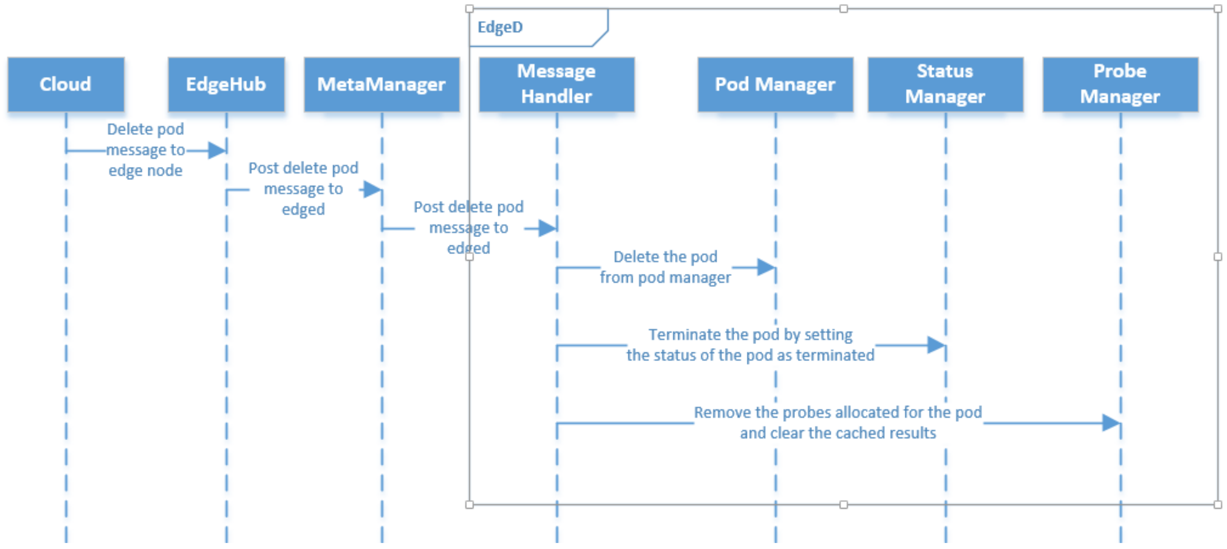
- Receives and handles pod addition/deletion/modification messages from metamanager.
- Handles separate worker queues for pod addition and deletion.
- Handles worker routines to check worker queues to do pod operations.
- Keeps separate cache for config map and secrets respectively.
- Regular cleanup of orphaned pods


EdgeD
Cloud
EdgeHub
MetaManager
Message Handler
Pod Manager
Probe Manager
Pod Addition Queue
Pod Add Worker Routine
Add pod message to edge node
Post add pod message to edged
Post add pod message to edged
Get the pod information from message content and add it to pod manager.
Same pod information is added to pod manager.
Then pod information is added to pod addition worker queue. It is added with its assigned namespace/podname as the key.
Keep a watch on addition queue
Get pod from queue
1. Get pod info from pod manager
2. Create pod directory
3. Attach and mount volumes if configured
4. Get secrets if configured
5. Ensure image exists and if not download using container runtime
6. Start the pod using container runtime.
7. Mark it as Done in pod addition queue.
Mark as Done in queue for the pod

Pod

Addition Flow

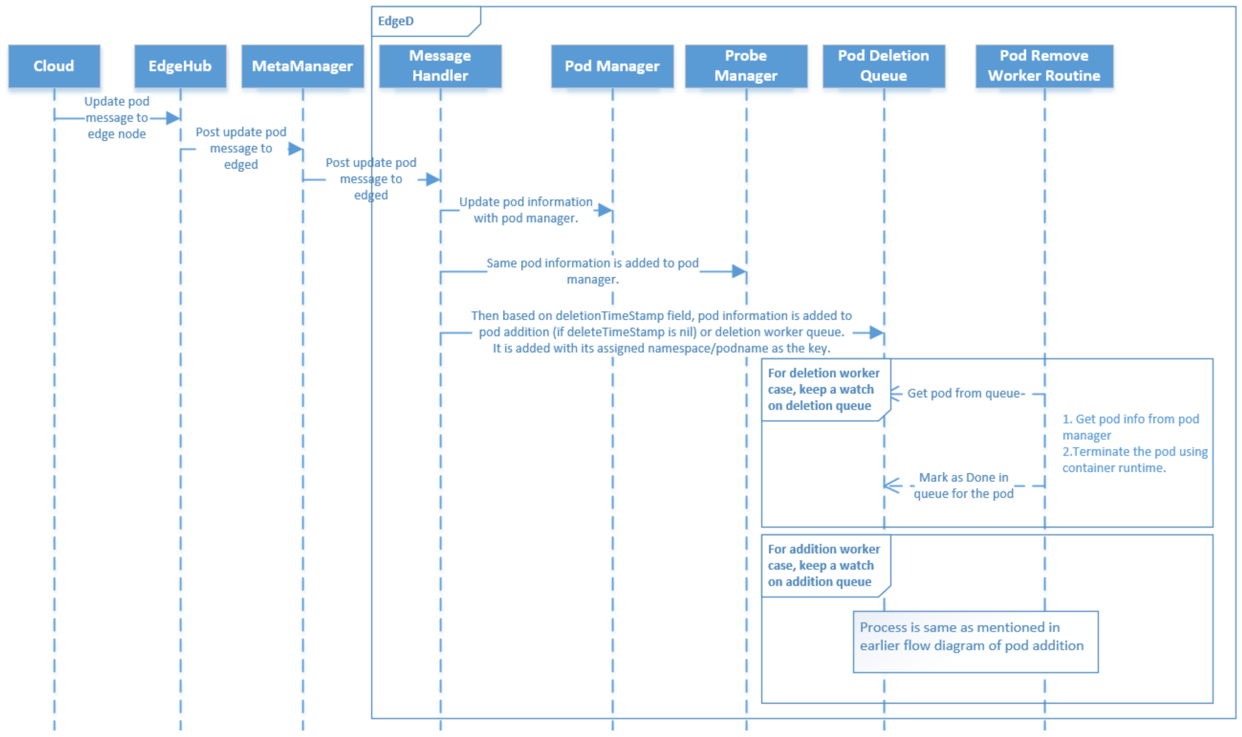
Fig 2: Pod Addition Flow


EdgeD
Cloud
EdgeHub
MetaManager
Message Handler
Pod Manager
Status Manager
Probe Manager
Delete pod message to edge node
Post delete pod message to edged
Post delete pod message to edged
Delete the pod from pod manager
Terminate the pod by setting the status of the pod as terminated
Remove the probes allocated for the pod and clear the cached results

Pod

Deletion Flow

Fig 3: Pod Deletion Flow

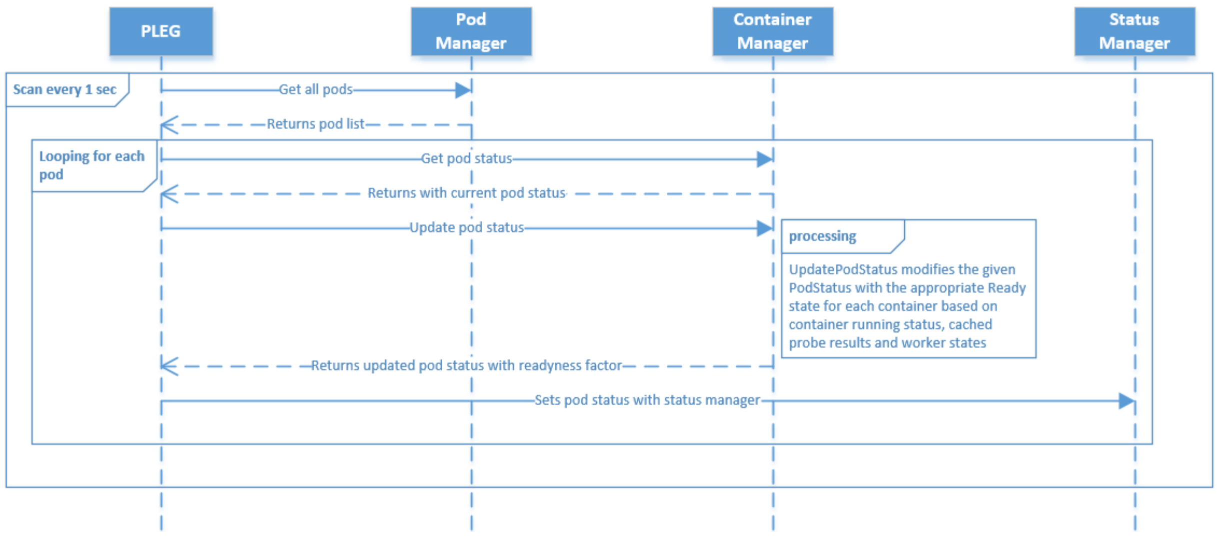

EdgeD
Cloud
EdgeHub
MetaManager
Message Handler
Pod Manager
Probe Manager
Pod Deletion Queue
Pod Remove Worker Routine
Update pod message to edge node
Post update pod message to edged
Post update pod message to edged
Update pod information with pod manager.
Same pod information is added to pod manager.
Then based on deletionTimeStamp field, pod information is added to pod addition (if deleteTimeStamp is nil) or deletion worker queue. It is added with its assigned namespace/podname as the key.
For deletion worker case, keep a watch on deletion queue
Get pod from queue
1. Get pod info from pod manager
2.Terminate the pod using container runtime.
Mark as Done in queue for the pod
For addition worker case, keep a watch on addition queue
Process is same as mentioned in earlier flow diagram of pod addition

Pod Updation Flow

Fig 4: Pod Updation Flow

7.3 Pod Lifecycle Event Generator

This module helps in monitoring pod status for edged. Every second, using probe's for liveliness and readyness, it updates the information with pod status manager for every pod.

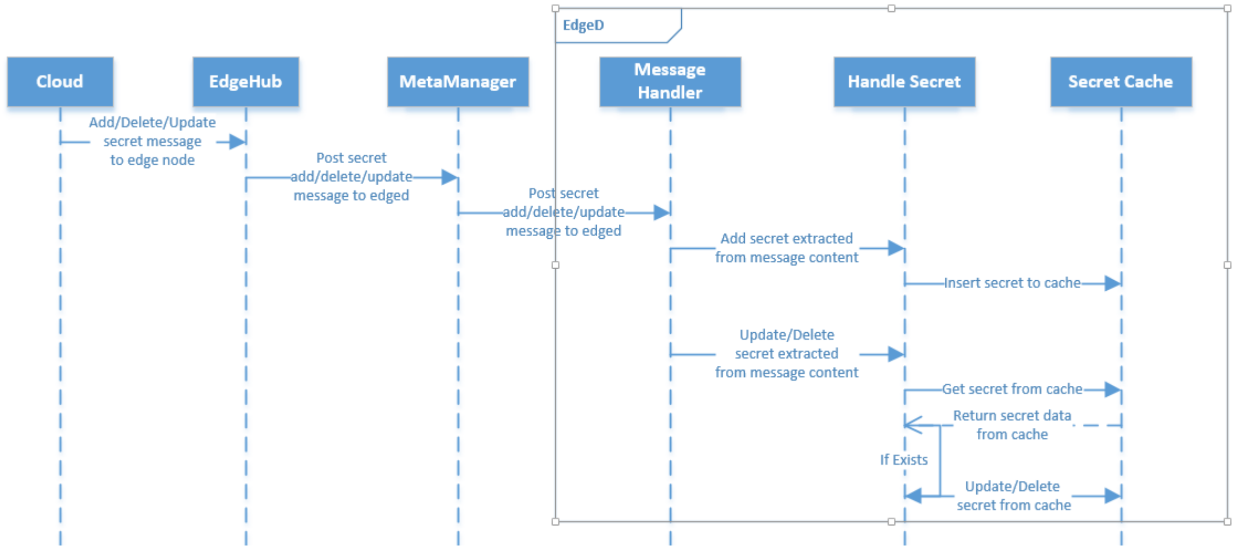

PLEG
Pod Manager
Container Manager
Status Manager
Scan every 1 sec
Get all pods
Returns pod list
Looping for each pod
Get pod status
Returns with current pod status
Update pod status
processing
UpdatePodStatus modifies the given PodStatus with the appropriate Ready state for each container based on container running status, cached probe results and worker states
Returns updated pod status with readyness factor
Sets pod status with status manager

PLEG Design

Fig 5: PLEG at EdgeD

7.4 Secret Management

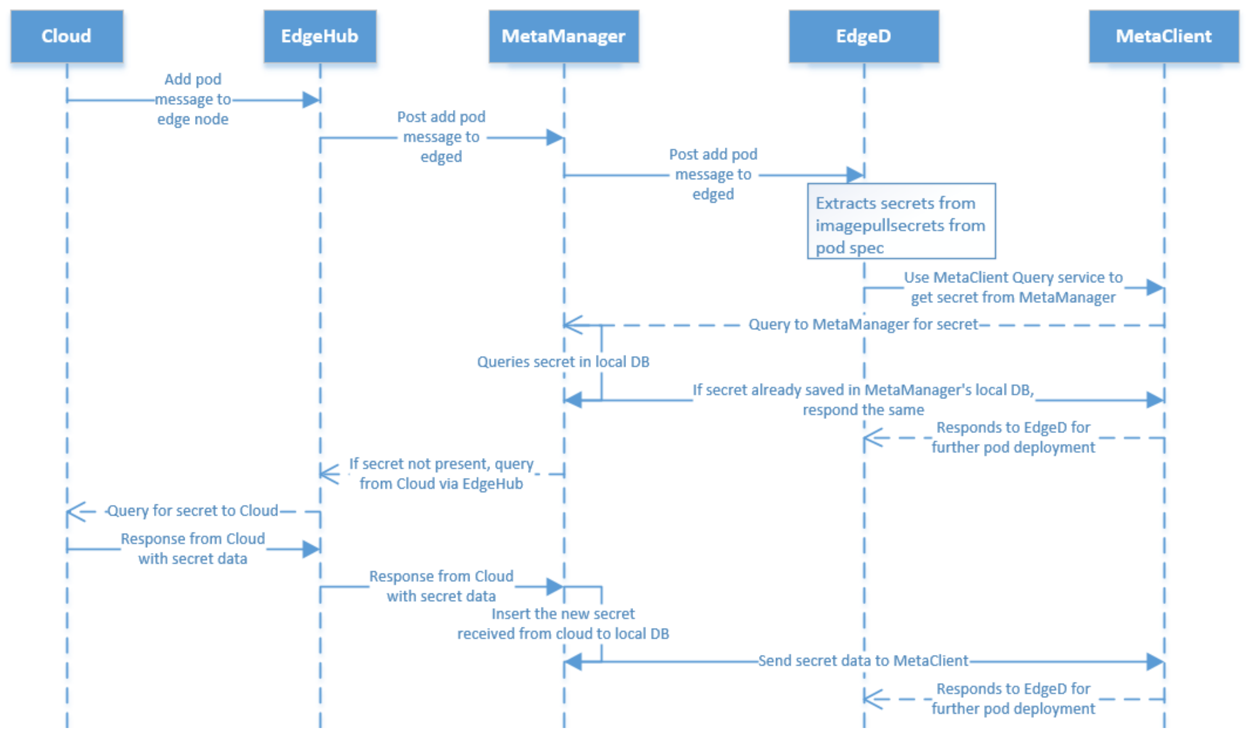
At edged, Secrets are handled separately. For its operations like addition, deletion and modifications; there are separate set of config messages or interfaces. Using these interfaces, secrets are updated in cache store. Below flow diagram explains the message flow.


EdgeD
Cloud
EdgeHub
MetaManager
Message Handler
Handle Secret
Secret Cache
Add/Delete/Update secret message to edge node
Post secret add/delete/update message to edged
Post secret add/delete/update message to edged
Add secret extracted from message content
Insert secret to cache
Update/Delete secret extracted from message content
Get secret from cache
Return secret data from cache
If Exists
Update/Delete secret from cache

Secret Message Handling

Fig 6: Secret Message Handling at EdgeD

Also edged uses MetaClient module to fetch secret from Metamanager (if available with it) else cloud. Whenever edged queries for a new secret which Metamanager doesn't has, the request is forwarded to cloud. Before sending the response containing the secret, it stores a copy of it and send it to edged. Hence the subsequent query for same secret key will be responded by Metamanger only, hence reducing the response delay. Below flow diagram shows, how secret is fetched from metamanager and cloud. The flow of how secret is saved in metamanager.


Cloud
EdgeHub
MetaManager
EdgeD
MetaClient
Add pod message to edge node
Post add pod message to edged
Post add pod message to edged
Extracts secrets from imagepullsecrets from pod spec
Use MetaClient Query service to get secret from MetaManager
Query to MetaManager for secret
Queries secret in local DB
If secret already saved in MetaManager's local DB, respond the same
Responds to EdgeD for further pod deployment
If secret not present, query from Cloud via EdgeHub
Query for secret to Cloud
Response from Cloud with secret data
Response from Cloud with secret data
Insert the new secret received from cloud to local DB
Send secret data to MetaClient
Responds to EdgeD for further pod deployment

Query Secret

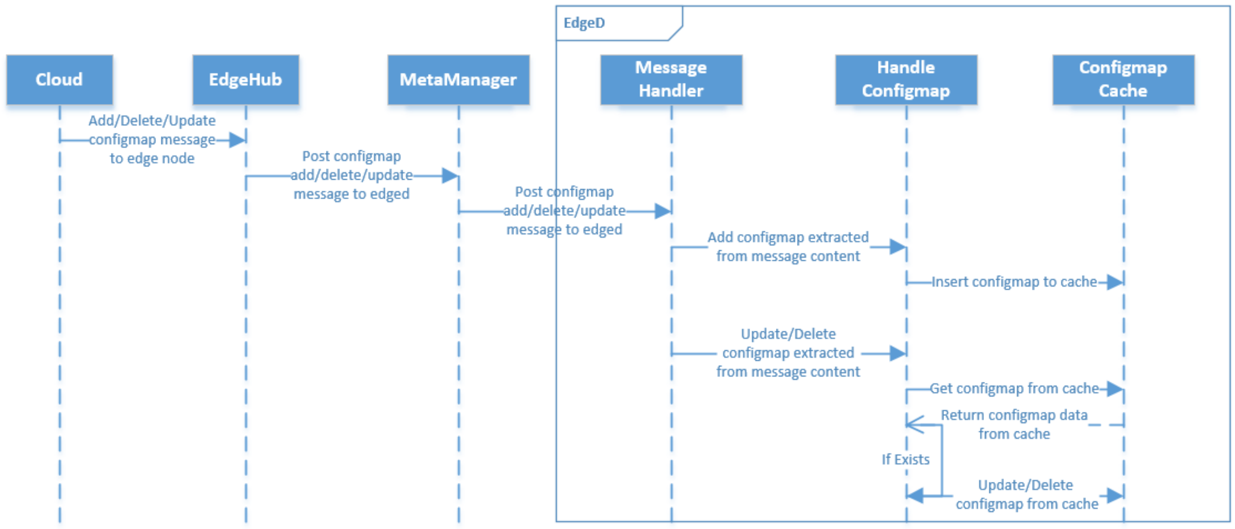
Fig 7: Query Secret by EdgeD

7.5 Probe Management

Probe management creates to probes for readiness and liveliness respectively for pods to monitor the containers. Readiness probe helps by monitoring when the pod has reached to running state. Liveliness probe helps in monitoring the health of pods, if they are up or down. As explained earlier PLEG module uses its services.

7.6 ConfigMap Management

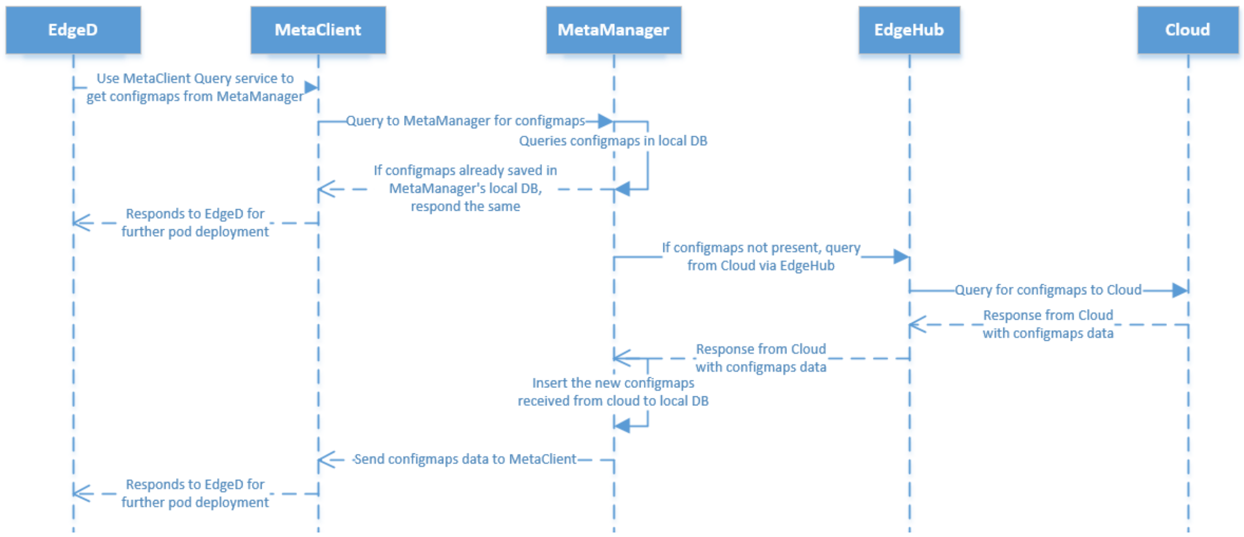
At edged, ConfigMap are also handled separately. For its operations like addition, deletion and modifications; there are separate set of config messages or interfaces. Using these interfaces, configMaps are updated in cache store. Below flow diagram explains the message flow.


EdgeD
Cloud
EdgeHub
MetaManager
Message Handler
Handle Configmap
Configmap Cache
Add/Delete/Update configmap message to edge node
Post configmap add/delete/update message to edged
Post configmap add/delete/update message to edged
Add configmap extracted from message content
Insert configmap to cache
Update/Delete configmap extracted from message content
Get configmap from cache
Return configmap data from cache
If Exists
Update/Delete configmap from cache

ConfigMap Message Handling

Fig 8: ConfigMap Message Handling at EdgeD

Also edged uses MetaClient module to fetch configmap from Metamanager (if available with it) else cloud. Whenever edged queries for a new configmaps which Metamanager doesn't has, the request is forwarded to cloud. Before sending the response containing the configmaps, it stores a copy of it and send it to edged. Hence the subsequent query for same configmaps key will be responded by Metamanger only, hence reducing the response delay. Below flow diagram shows, how configmaps is fetched from metamanager and cloud. The flow of how configmaps is saved in metamanager.


EdgeD
MetaClient
MetaManager
EdgeHub
Cloud
Use MetaClient Query service to get configmaps from MetaManager
Query to MetaManager for configmaps
Queries configmaps in local DB
If configmaps already saved in MetaManager's local DB, respond the same
Responds to EdgeD for further pod deployment
If configmaps not present, query from Cloud via EdgeHub
Query for configmaps to Cloud
Response from Cloud with configmaps data
Response from Cloud with configmaps data
Insert the new configmaps received from cloud to local DB
Send configmaps data to MetaClient
Responds to EdgeD for further pod deployment

Query Configmaps

Fig 9: Query Configmaps by EdgeD

7.7 Container GC

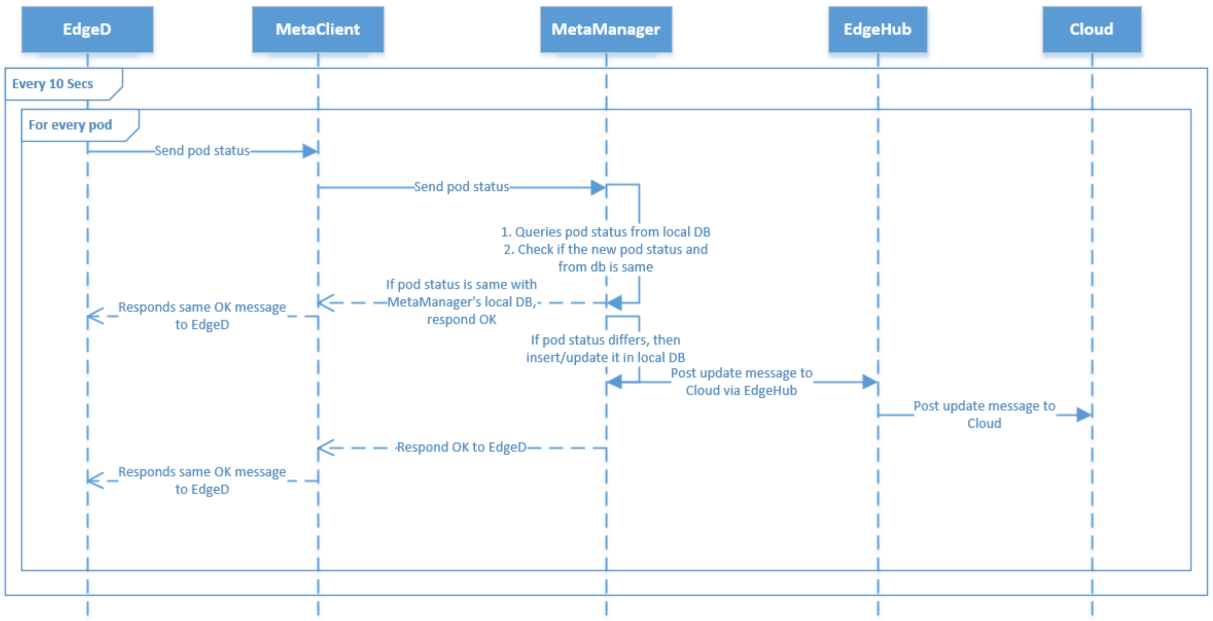
Container garbage collector is an edged routine which wakes up every minute, collecting and removing dead containers using the specified container gc policy The policy for garbage collecting containers we apply takes on three variables, which can be user-defined. MinAge is the minimum age at which a container can be garbage collected, zero for no limit. MaxPerPodContainer is the max number of dead containers any single pod (UID, container name) pair is allowed to have, less than zero for no limit. MaxContainers is the max number of total dead containers, less than zero for no limit as well. Gernerally the oldest containers are removed first.

7.8 Image GC

Image garbage collector is an edged routine which wakes up every 5 secs, collects information about disk usage based on the policy used. The policy for garbage collecting images we apply takes two factors into consideration, HighThresholdPercent and LowThresholdPercent. Disk usage above the high threshold will trigger garbage collection, which attempts to delete unused images until the low threshold is met. Least recently used images are deleted first.

7.9 Status Manager

Status manager is as an independent edge routine, which collects pods statuses every 10 seconds and forwards this information with cloud using metaclient interface to the cloud.


EdgeD
MetaClient
MetaManager
EdgeHub
Cloud
Every 10 Secs
For every pod
Send pod status
Send pod status
1. Queries pod status from local DB
2. Check if the new pod status and from db is same
If pod status is same with MetaManager's local DB, respond OK
Responds same OK message to EdgeD
If pod status differs, then insert/update it in local DB
Post update message to Cloud via EdgeHub
Post update message to Cloud
Respond OK to EdgeD
Responds same OK message to EdgeD

Status Manager Flow

Fig 10: Status Manager Flow

7.10 Volume Management

Volume manager runs as an edge routine which brings out the information of which volume(s) are to be attached/mounted/unmounted/detached based on pods scheduled on the edge node.

Before starting the pod, all the specified volumes referenced in pod specs are attached and mounted, Till then the flow is blocked and with it other operations.

7.11 MetaClient

Metaclient is an interface of Metamanger for edged. It helps edge to get configmap and secret details from metamanager or cloud. It also sends sync messages, node status and pod status towards metamanger to cloud.

CHAPTER 8

EventBus

8.1 Overview

Eventbus acts as an interface for sending/receiving messages on mqtt topics.

It supports 3 kinds of mode:

- internalMqttMode
- externalMqttMode
- bothMqttMode

8.2 Topic

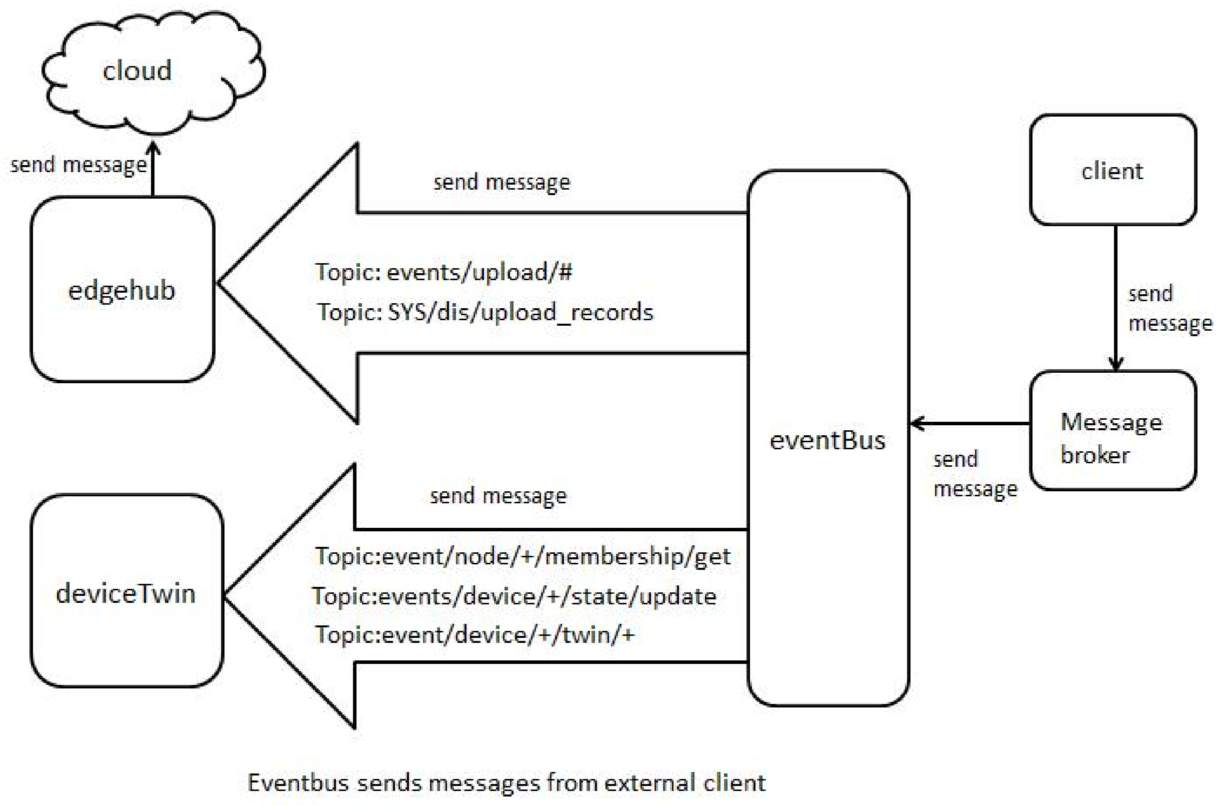
eventbus subscribes to the following topics:

```
- $hw/events/upload/#
- SYS/dis/upload_records
- SYS/dis/upload_records/+
- $hw/event/node/+/membership/get
- $hw/event/node/+/membership/get/+
- $hw/events/device/+/state/update
- $hw/events/device/+/state/update/+
- $hw/event/device/+/twin/+
```

Note: topic wildcards

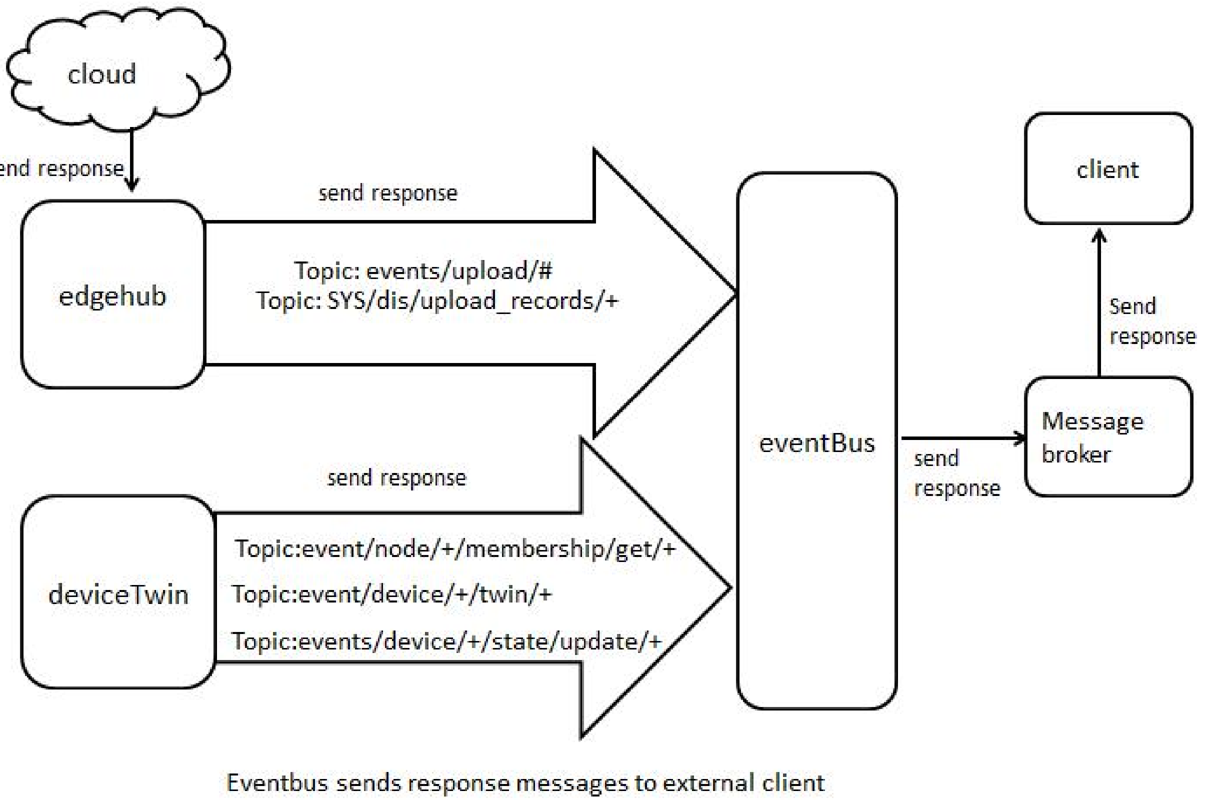
8.3 Flow chart

8.3.1 1. eventbus sends messages from external client


cloud
send message
edgehub
send message
Topic: events/upload/#
Topic: SYS/dis/upload_records
eventBus
client
send message
Message broker
send message
deviceTwin
send message
Topic:event/node/+/membership/get
Topic:events/device/+/state/update
Topic:event/device/+/twin/+
Eventbus sends messages from external client

eventbus sends messages from external client

8.3.2 2. eventbus sends response messages to external client


cloud
send response
edgehub
send response
Topic: events/upload/#
Topic: SYS/dis/upload_records/+
deviceTwin
send response
Topic:event/node/+/membership/get/+
Topic:event/device/+/twin/+
Topic:events/device/+/state/update/+
eventBus
send response
Message broker
Send response
client
Eventbus sends response messages to external client

eventbus sends response messages to external client

CHAPTER 9

MetaManager

9.1 Overview

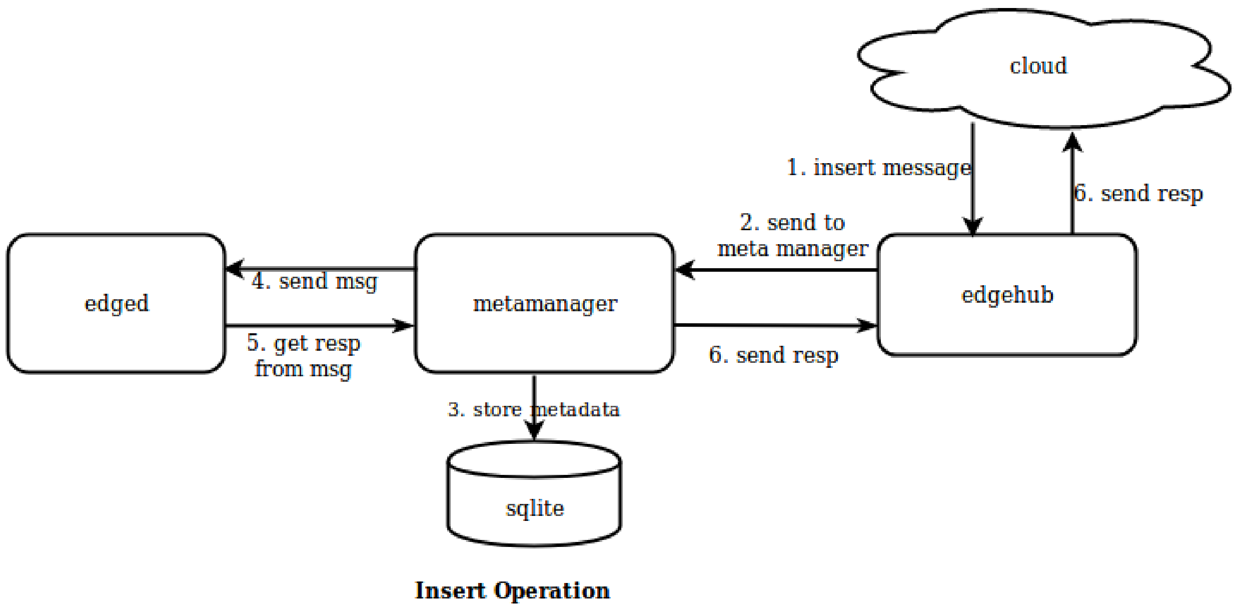
MetaManager is the message processor between edged and edgehub. It's also responsible for storing/retrieving metadata to/from a lightweight database(SQLite).

Metamanager receives different types of messages based on the operations listed below :

- Insert
- Update
- Delete
- Query
- Response
- NodeConnection
- MetaSync

9.2 Insert Operation

`Insert` operation messages are received via the cloud when new objects are created. An example could be a new user application pod created/deployed through the cloud.



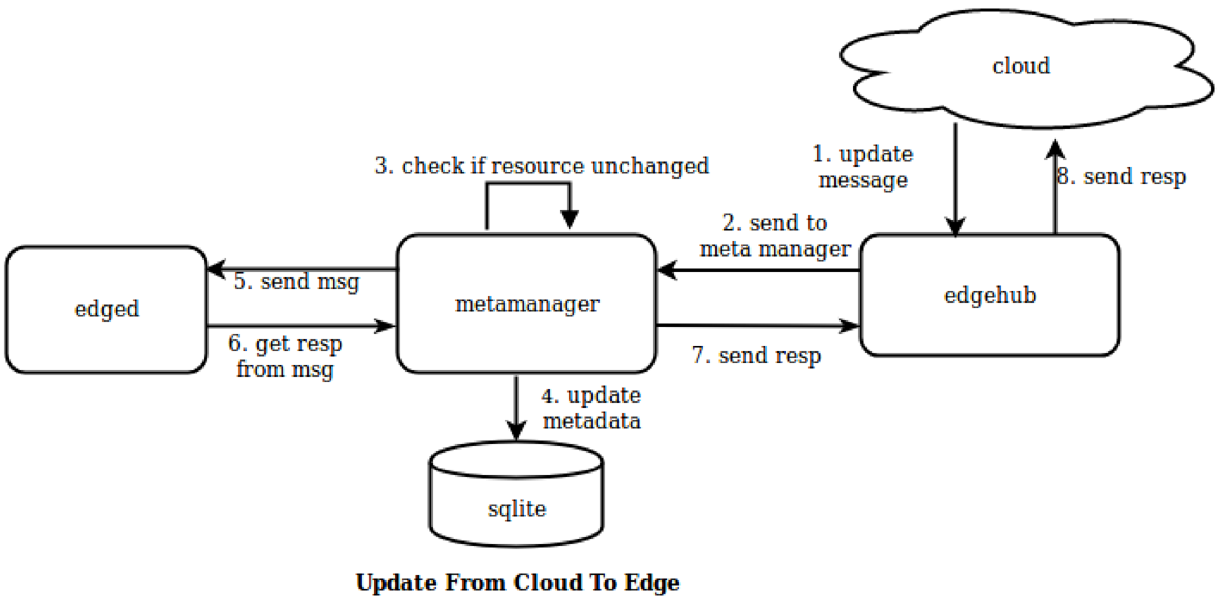
Insert Operation

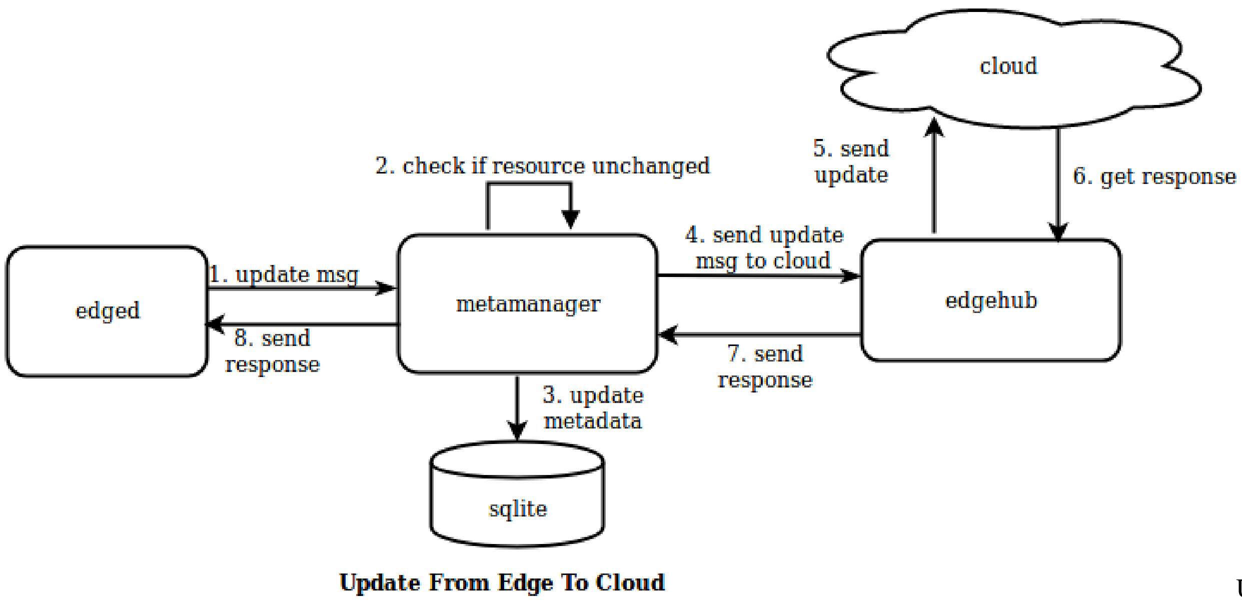
The insert operation request is received via the cloud by edgehub. It dispatches the request to the metamanager which saves this message in the local database. metamanager then sends an asynchronous message to edged. edged processes the insert request e,g. by starting the pod and populates the response in the message. metamanager inspects the message, extracts the response and sends it back to edged which sends it back to the cloud.

9.3 Update Operation

`Update` operations can happen on objects at the cloud/edge.

The update message flow is similar to an insert operation. Additionally, metamanager checks if the resource being updated has changed locally. If there is a delta, only then the update is stored locally and the message is passed to edged and response is sent back to the cloud.

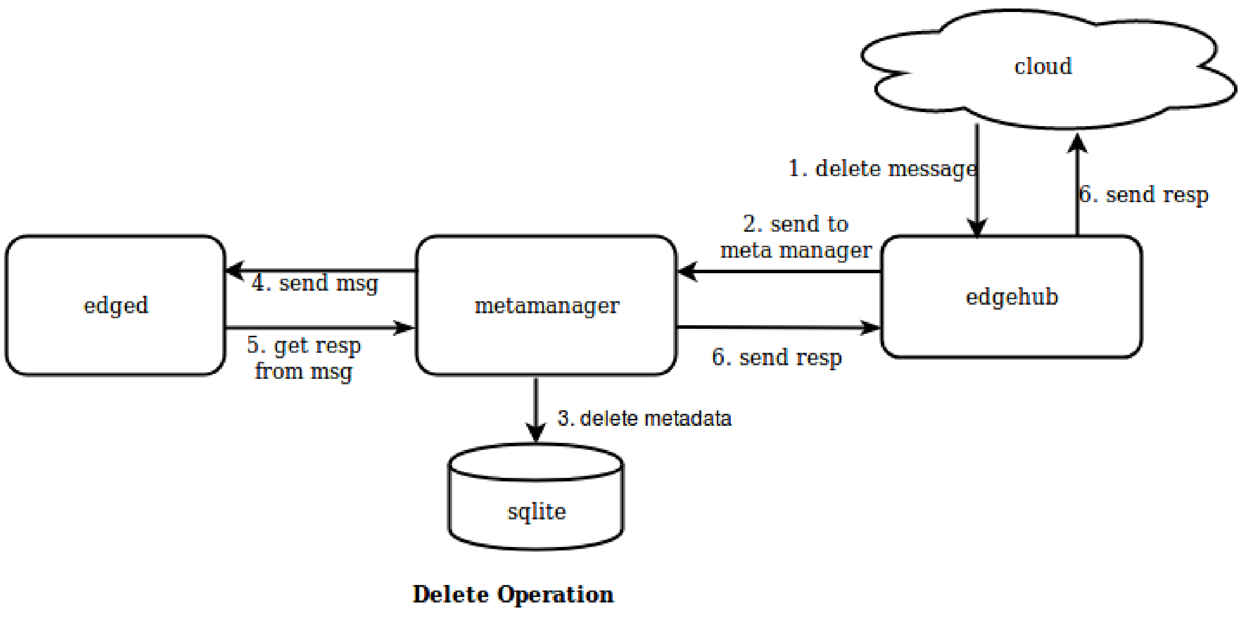




Update Operation

9.4 Delete Operation

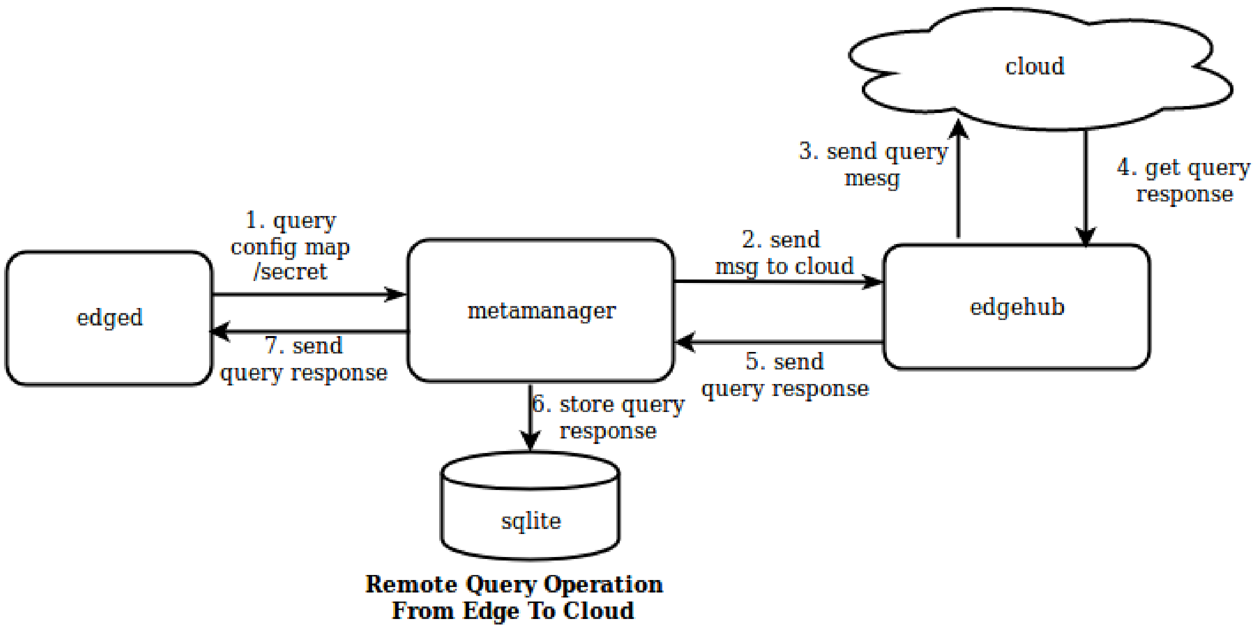
`Delete` operations are triggered when objects like pods are deleted from the cloud.

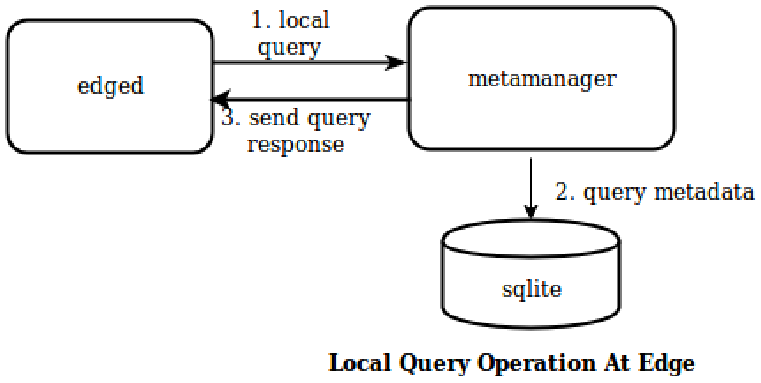


Delete Operation

9.5 Query Operation

`Query` operations let you query for metadata either locally at the edge or for some remote resources like config maps/secrets from the cloud. edged queries this metadata from metamanager which further handles local/remote query processing and returns the response back to edged. A Message resource can be broken into 3 parts (resKey,resType,resId) based on separator '/'.





Query Operation

9.6 Response Operation

`Responses` are returned for any operations performed at the cloud/edge. Previous operations showed the response flow either from the cloud or locally at the edge.

9.7 NodeConnection Operation

`NodeConnection` operation messages are received from edgeHub to give information about the cloud connection status. metamanager tracks this state in-memory and uses it in certain operations like remote query to the cloud.

9.8 MetaSync Operation

MetaSync operation messages are periodically sent by metamanager to sync the status of the pods running on the edge node. The sync interval is configurable in conf\edge.yaml (defaults to 60 seconds).

```yaml
meta:
    sync:
        podstatus:
            interval: 60 #seconds
```

CHAPTER 10

Edgehub

10.1 Overview

Edge hub is a web socket client, which is responsible for interacting with Huawei Cloud IEF service. It supports functions like sync cloud side resources update, report edged side host and device status changes.

It acts as the communication link between the edge and the cloud. It forwards the messages received from the cloud to the corresponding module at the edge and vice-versa.

The main functions performed by edgehub are :-

- Get CloudHub URL
- Keep Alive
- Publish Client Info
- Route to Cloud
- Route to Edge

10.2 Get CloudHub URL

The main responsibility of get cloudHub URL is to contact the placement server and get the URL of cloudHub.

1. A HTTPS client is created using the certificates provided
2. A get request is sent to the placement URL
3. ProjectID and NodeID are added to the body of the response received from the placement URL to form the cloudHub URL.

```
bodyBytes, _ := ioutil.ReadAll(resp.Body)
url := fmt.Sprintf("%s/%s/%s/events", string(bodyBytes), ehc.config.ProjectID, ehc.
↪config.NodeID)
```

10.3 Keep Alive

A keep-alive message or heartbeat is sent to cloudHub after every heartbeatPeriod.

10.4 Publish Client Info

- The main responsibility of publish client info is to inform the other groups or modules regarding the status of connection to the cloud.
- It sends a beehive message to all groups (namely metaGroup, twinGroup and busGroup), informing them whether cloud is connected or disconnected.

10.5 Route To Cloud

The main responsibility of route to cloud is to receive from the other modules (through beehive framework), all the messages that are to be sent to the cloud, and send them to cloudHub through the websocket connection.

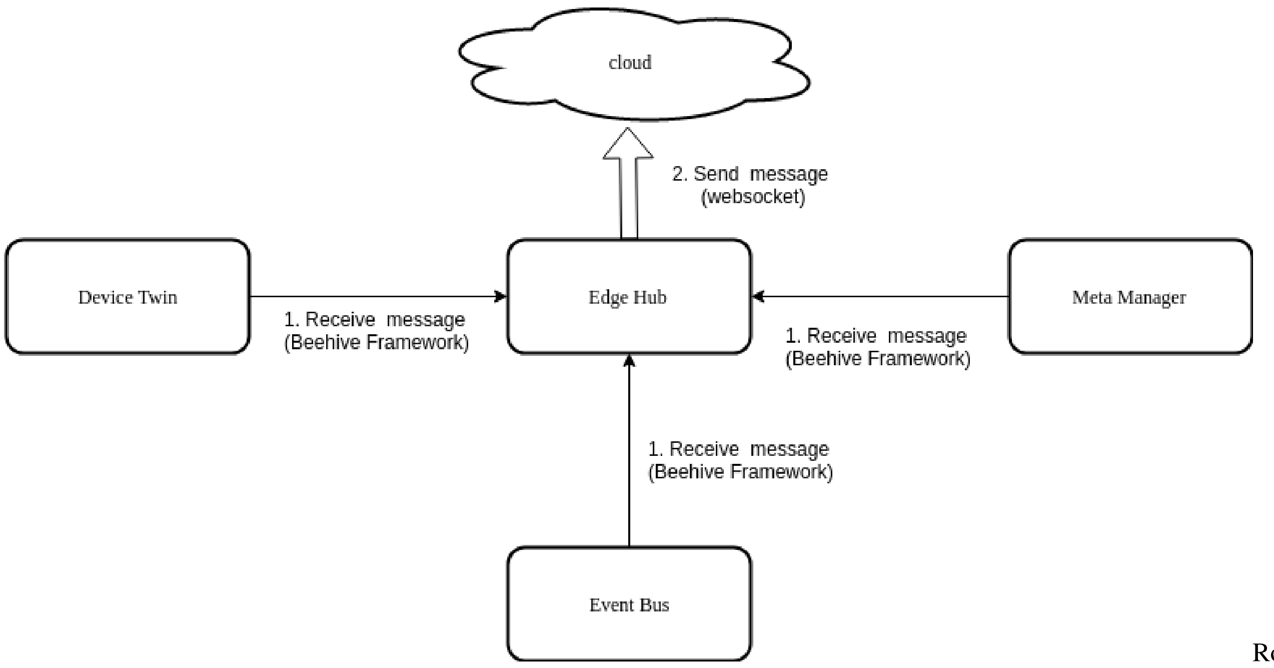
The major steps involved in this process are as follows :-

1. Continuously receive messages from beehive Context
2. Send that message to cloudHub
3. If the message received is a sync message then :

   3.1 If response is received on syncChannel then it creates a map[string] chan containing the messageID of the message as key

   3.2 It waits for one heartbeat period to receive a response on the channel created, if it does not receive any response on the channel within the specified time then it times out.

   3.3 The response received on the channel is sent back to the module using the SendResponse() function.


cloud
2. Send message (websocket)
Device Twin
1. Receive message (Beehive Framework)
Edge Hub
1. Receive message (Beehive Framework)
Meta Manager
1. Receive message (Beehive Framework)
Event Bus

Route

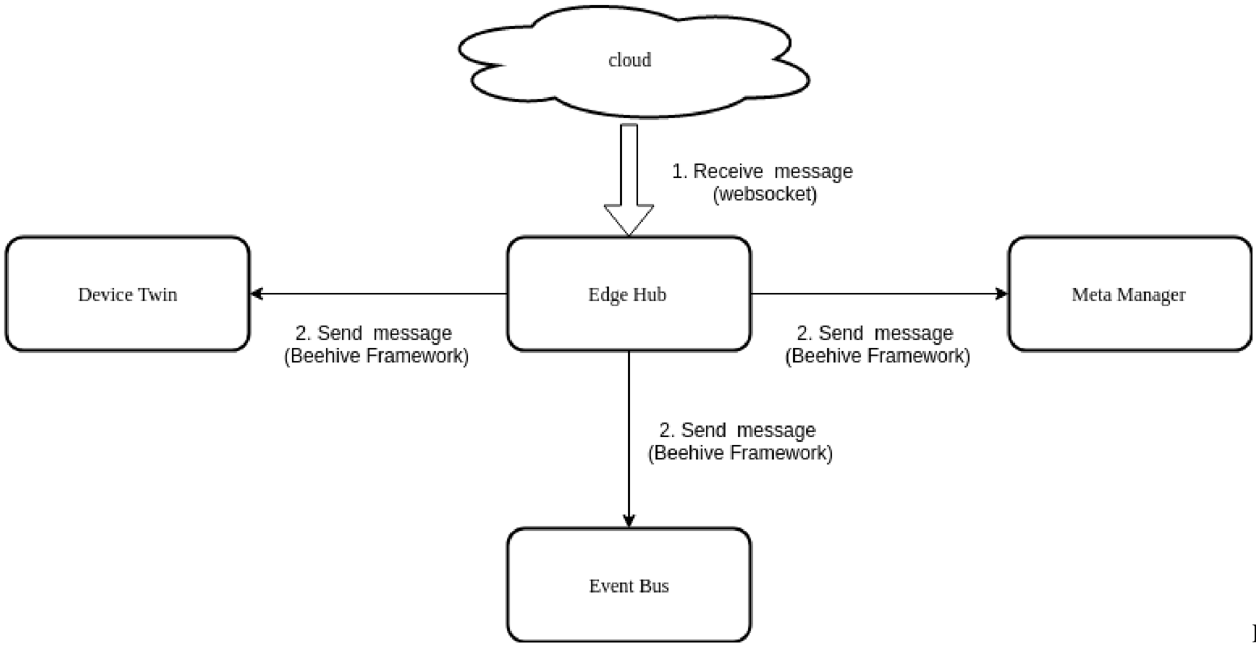
to Cloud

10.6 Route To Edge

The main responsibility of route to edge is to receive messages from the cloud (through the websocket connection) and send them to the required groups through the beehive framework.

The major steps involved in this process are as follows :-

- Receive message from cloudHub
- Check whether the route group of the message is found.
- Check if it is a response to a SendSync() function.
- If it is not a response message then the message is sent to the required group
- If it is a response message then the message is sent to the syncKeep channel


cloud
1. Receive message (websocket)
Device Twin
Edge Hub
Meta Manager
2. Send message (Beehive Framework)
2. Send message (Beehive Framework)
2. Send message (Beehive Framework)
Event Bus

Route to Edge

CHAPTER 11

DeviceTwin

11.1 Overview

DeviceTwin module is responsible for storing device status, dealing with device attributes, handling device twin operations, creating a membership between the edge device and edge node, syncing device status to the cloud and syncing the device twin information between edge and cloud. It also provides query interfaces for applications. Device twin consists of four sub modules (namely membership module, communication module, device module and device twin module) to perform the responsibilities of device twin module.

11.2 Operations Performed By Device Twin Controller

The following are the functions performed by device twin controller :-

- Sync metadata to/from db (Sqlite)
- Register and Start Sub Modules
- Distribute message to Sub Modules
- Health Check

11.2.1 Sync Metadata to/from db (Sqlite)

For all devices managed by the edge node , the device twin performs the below operations :-

- It checks if the device in the device twin context (the list of devices are stored inside the device twin context), if not it adds a mutex to the context.
- Query device from database
- Query device attribute from database
- Query device twin from database

- Combine the device, device attribute and device twin data together into a single structure and stores it in the device twin context.

11.2.2 Register and Start Sub Modules

Registers the four device twin modules and starts them as separate go routines

11.2.3 Distribute Message To Sub Modules

1. Continuously listen for any device twin message in the beehive framework.
2. Send the received message to the communication module of device twin
3. Classify the message according to the message source, i.e. whether the message is from eventBus, edgeManager or edgeHub, and fills the action module map of the module (ActionModuleMap is a map of action to module)
4. Send the message to the required device twin module

11.2.4 Health Check

The device twin controller periodically (every 60 s) sends ping messages to submodules. Each of the submodules updates the timestamp in a map for itself once it receives a ping. The controller checks if the timestamp for a module is more than 2 minutes old and restarts the submodule if true.

11.3 Modules

DeviceTwin consists of four modules, namely :-

- Membership Module
- Twin Module
- Communication Module
- Device Module

11.3.1 Membership Module

The main responsibility of the membership module is to provide membership to the new devices added through the cloud to the edge node. This module binds the newly added devices to the edge node and creates a membership between the edge node and the edge devices.

The major functions performed by this module are:-

1. Initialize action callback map which is a map[string]Callback that contains the callback functions that can be performed
2. Receive the messages sent to membership module
3. For each message the action message is read and the corresponding function is called
4. Receive heartbeat from the heartbeat channel and send a heartbeat to the controller

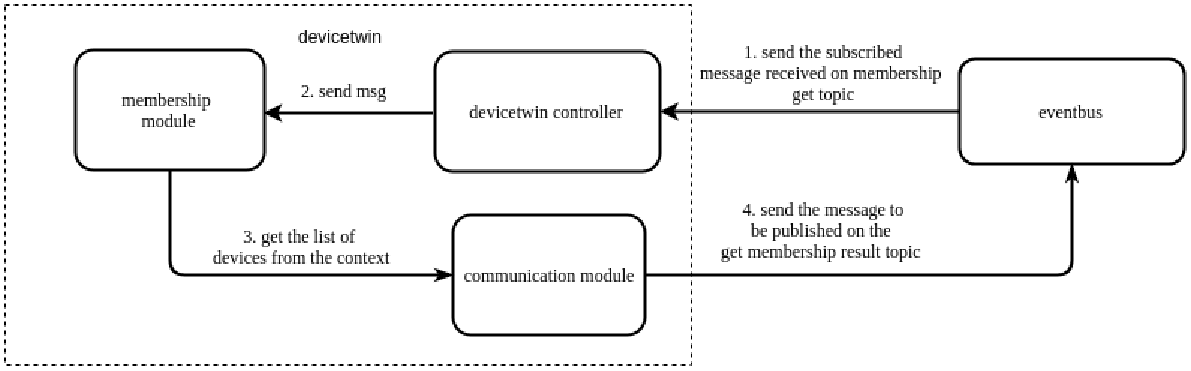
The following are the action callbacks which can be performed by the membership module :-

- dealMembershipGet

- dealMembershipUpdated
- dealMembershipDetail

dealMembershipGet: dealMembershipGet() gets the information about the devices associated with the particular edge node, from the cache.

- The eventbus first receives a message on its subscribed topic (membership-get topic).
- This message arrives at the devicetwin controller, which further sends the message to membership module.
- The membership module gets the devices associated with the edge node from the cache (context) and sends the information to the communication module. It also handles errors that may arise while performing the aforementioned process and sends the error to the communication module instead of device details.
- The communication module sends the information to the eventbus component which further publishes the result on the specified MQTT topic (get membership result topic).

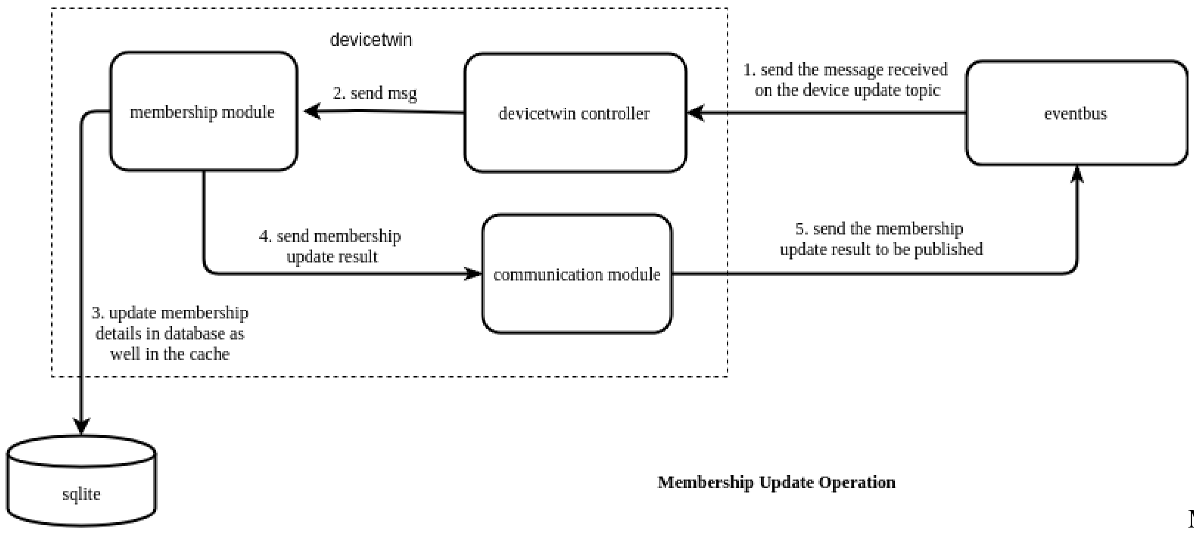

devicetwin
membership module
2. send msg
devicetwin controller
1. send the subscribed message received on membership get topic
eventbus
3. get the list of devices from the context
communication module
4. send the message to be published on the get membership result topic

Memebership Get Operation

Membership Get()

dealMembershipUpdated: dealMembershipUpdated() updates the membership details of the node. It adds the devices, that were newly added, to the edge group and removes the devices, that were removed, from the edge group and updates device details, if they have been altered or updated.

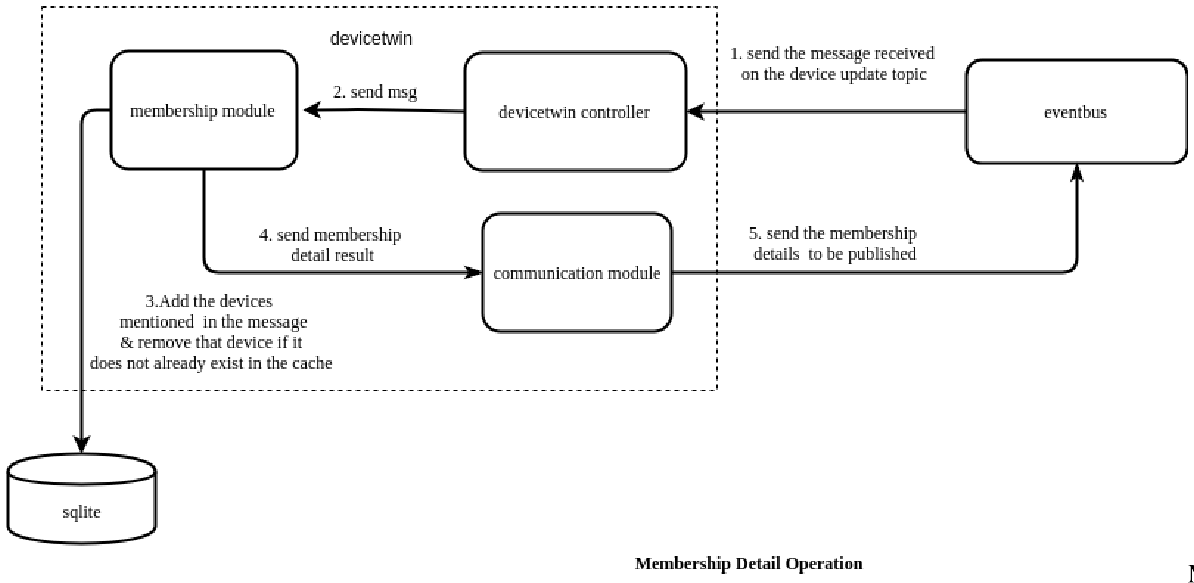
- The eventbus module receives the message that arrives on the subscribed topic and forwards the message to devicetwin controller which further forwards it to the membership module.
- The membership module adds devices that are newly added, removes devices that have been recently deleted and also updates the devices that were already existing in the database as well as in the cache.
- After updating the details of the devices a message is sent to the communication module of the device twin, which sends the message to eventbus module to be published on the given MQTT topic.


devicetwin
membership module
2. send msg
devicetwin controller
1. send the message received on the device update topic
eventbus
4. send membership update result
communication module
5. send the membership update result to be published
3. update membership details in database as well in the cache
sqlite
Membership Update Operation

Membership Update

dealMembershipDetail: dealMembershipDetail() provides the membership details of the edge node, providing information about the devices associated with the edge node, after removing the membership details of recently removed devices.

- The eventbus module receives the message that arrives on the subscribed topic,the message is then forwarded to the devicetwin controller which further forwards it to the membership module.
- The membership module adds devices that are mentioned in the message, removes devices that that are not present in the cache.
- After updating the details of the devices a message is sent to the communication module of the device twin.


devicetwin
membership module
2. send msg
devicetwin controller
1. send the message received on the device update topic
eventbus
4. send membership detail result
communication module
5. send the membership details to be published
3.Add the devices mentioned in the message & remove that device if it does not already exist in the cache
sqlite
Membership Detail Operation

Membership Detail

11.3.2 Twin Module

The main responsibility of the twin module is to deal with all the device twin related operations. It can perform operations like device twin update, device twin get and device twin sync-to-cloud.

The major functions performed by this module are:-

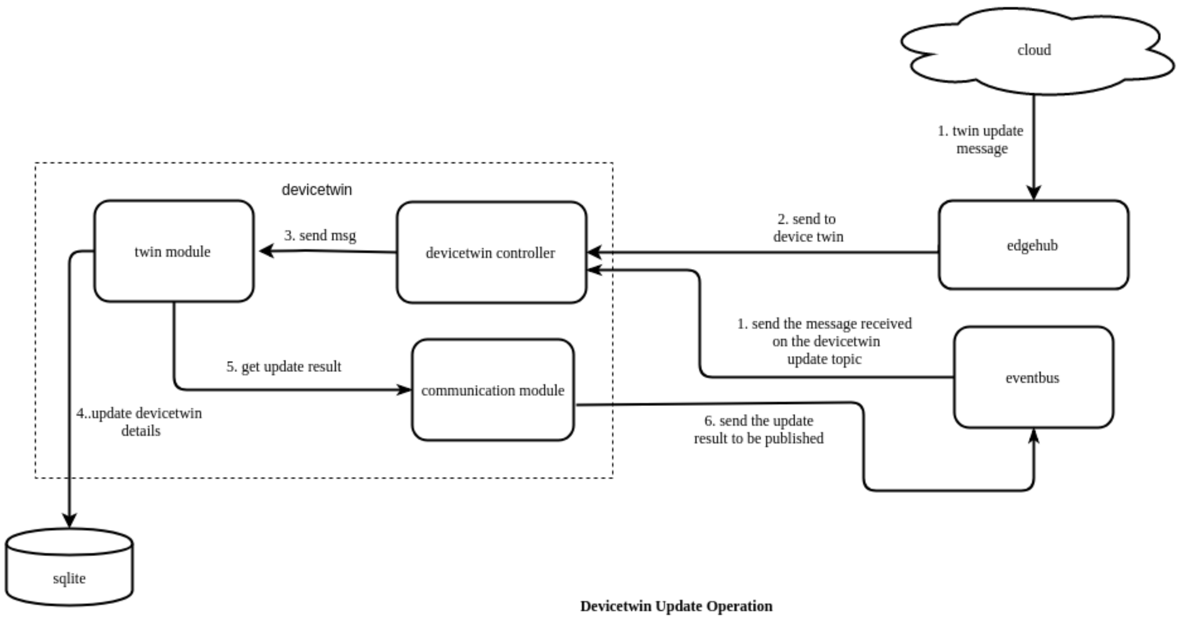
1. Initialize action callback map (which is a map of action(string) to the callback function that performs the requested action)
2. Receive the messages sent to twin module
3. For each message the action message is read and the corresponding function is called
4. Receive heartbeat from the heartbeat channel and send a heartbeat to the controller

The following are the action callbacks which can be performed by the twin module :-

- dealTwinUpdate
- dealTwinGet
- dealTwinSync

dealTwinUpdate: dealTwinUpdate() updates the device twin information for a particular device.

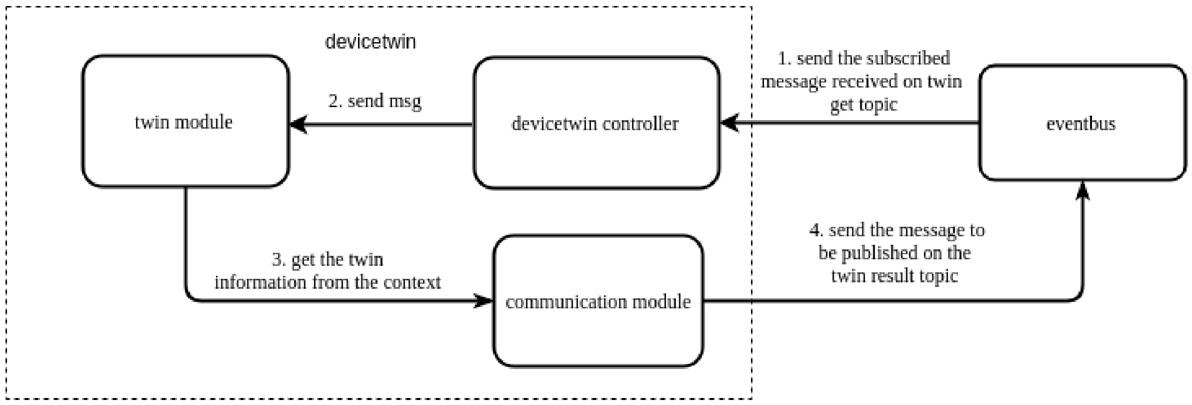
- The devicetwin update message can either be received by edgehub module from the cloud or from the MQTT broker through the eventbus component (mapper will publish a message on the device twin update topic) .
- The message is then sent to the device twin controller from where it is sent to the device twin module.
- The twin module updates the twin value in the database and sends the update result message to the communication module.
- The communication module will in turn send the publish message to the MQTT broker through the eventbus.



Device Twin Update

dealTwinGet: dealTwinGet() provides the device twin information for a particular device.

- The eventbus component receives the message that arrives on the subscribed twin get topic and forwards the message to devicetwin controller, which further sends the message to twin module.
- The twin module gets the devicetwin related information for the particular device and sends it to the communication module, it also handles errors that arise when the device is not found or if any internal problem occurs.
- The communication module sends the information to the eventbus component, which publishes the result on the topic specified .

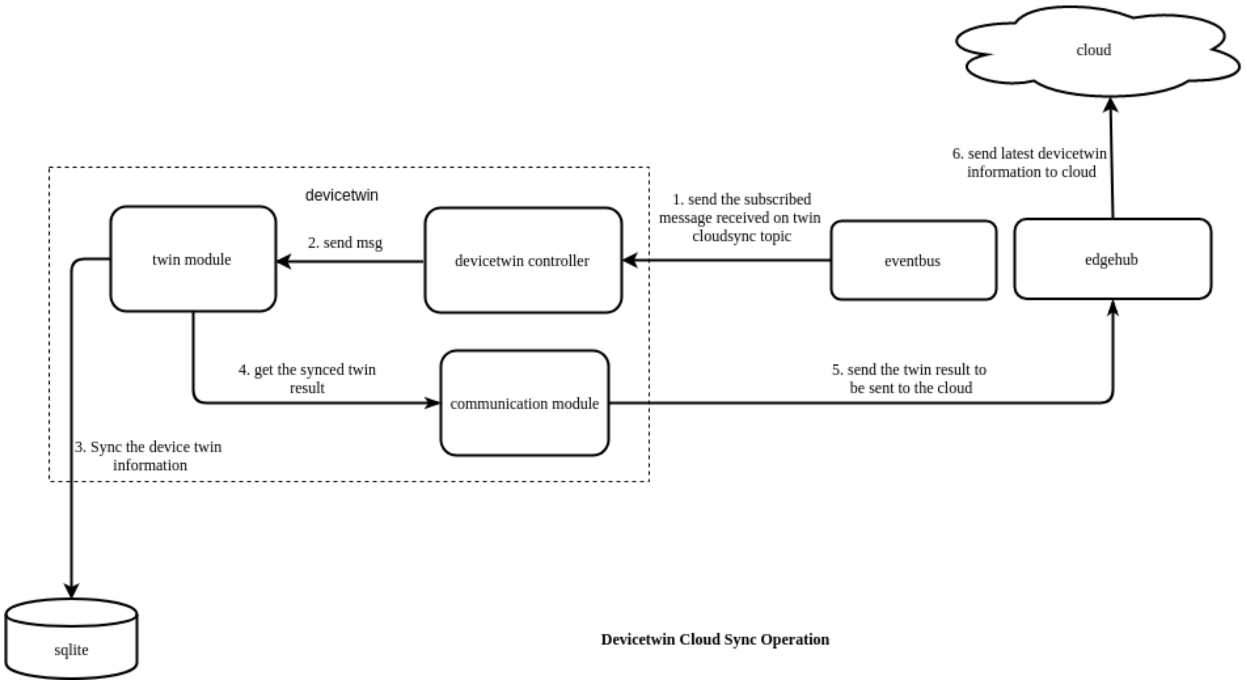

devicetwin
twin module
2. send msg
devicetwin controller
1. send the subscribed message received on twin get topic
eventbus
3. get the twin information from the context
communication module
4. send the message to be published on the twin result topic

Devicetwin Get Operation

Device Twin Get

dealTwinSync: dealTwinSync() syncs the device twin information to the cloud.

- The eventbus module receives the message on the subscribed twin cloud sync topic .
- This message is then sent to the devicetwin controller from where it is sent to the twin module.
- The twin module then syncs the twin information present in the database and sends the synced twin results to the communication module.
- The communication module further sends the information to edgehub component which will in turn send the updates to the cloud through the websocket connection.
- This function also performs operations like publishing the updated twin details document, delta of the device twin as well as the update result (in case there is some error) to a specified topic through the communication module, which sends the data to edgehub, which will send it to eventbus which publishes on the MQTT broker.


cloud
6. send latest devicetwin information to cloud
devicetwin
twin module
2. send msg
devicetwin controller
1. send the subscribed message received on twin cloudsync topic
eventbus
edgehub
4. get the synced twin result
communication module
5. send the twin result to be sent to the cloud
3. Sync the device twin information
sqlite

Devicetwin Cloud Sync Operation

Sync to Cloud

11.3.3 Communication Module

The main responsibility of communication module is to ensure the communication functionality between device twin and the other components.

The major functions performed by this module are:-

1. Initialize action callback map which is a map[string]Callback that contains the callback functions that can be performed
2. Receive the messages sent to communication module
3. For each message the action message is read and the corresponding function is called
4. Confirm whether the actions specified in the message are completed or not, if the action is not completed then redo the action
5. Receive heartbeat from the heartbeat channel and send a heartbeat to the controller

The following are the action callbacks which can be performed by the communication module :-

- dealSendToCloud
- dealSendToEdge
- dealLifeCycle
- dealConfirm

dealSendToCloud: dealSendToCloud() is used to send data to the cloudHub component. This function first ensures that the cloud is connected, then sends the message to the edgeHub module (through the beehive framework), which in turn will forward the message to the cloud (through the websocket connection).

dealSendToEdge: dealSendToEdge() is used to send data to the other modules present at the edge. This function sends the message received to the edgeHub module using beehive framework. The edgeHub module after receiving the message will send it to the required recipient.

dealLifeCycle: dealLifeCycle() checks if the cloud is connected and the state of the twin is disconnected, it then changes the status to connected and sends the node details to edgehub. If the cloud is disconnected then, it sets the state of the twin as disconnected.

dealConfirm: dealConfirm() is used to confirm the event. It checks whether the type of the message is right and then deletes the id from the confirm map.

11.3.4 Device Module

The main responsibility of the device module is to perform the device related operations like dealing with device state updates and device attribute updates.

The major functions performed by this module are :-

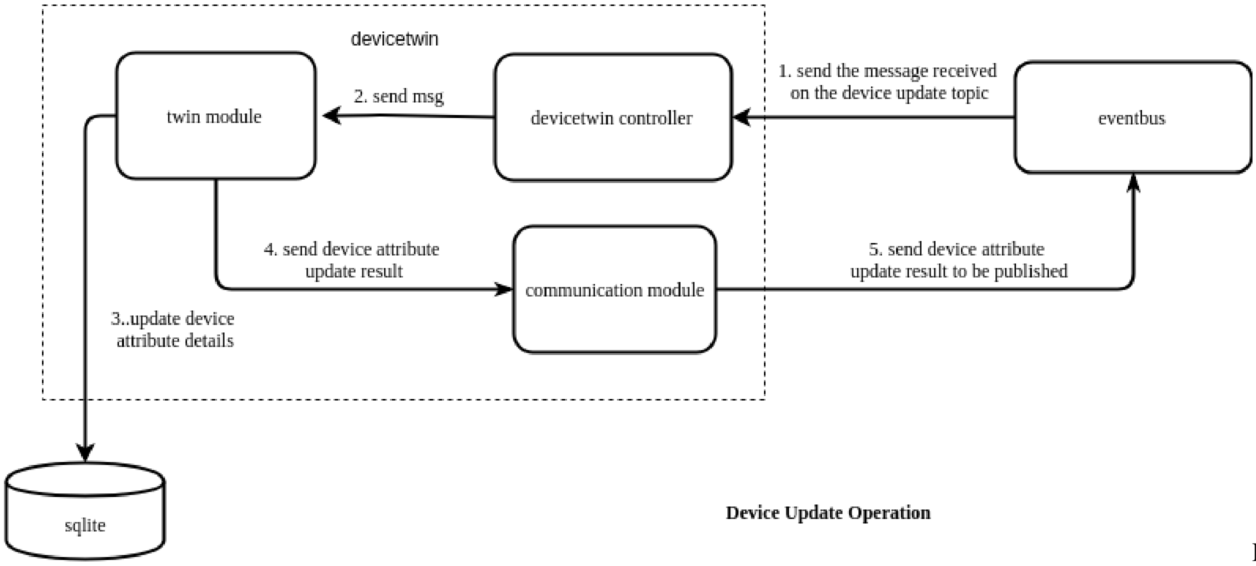
1. Initialize action callback map (which is a map of action(string) to the callback function that performs the requested action)
2. Receive the messages sent to device module
3. For each message the action message is read and the corresponding function is called
4. Receive heartbeat from the heartbeat channel and send a heartbeat to the controller

The following are the action callbacks which can be performed by the device module :-

- dealDeviceUpdated

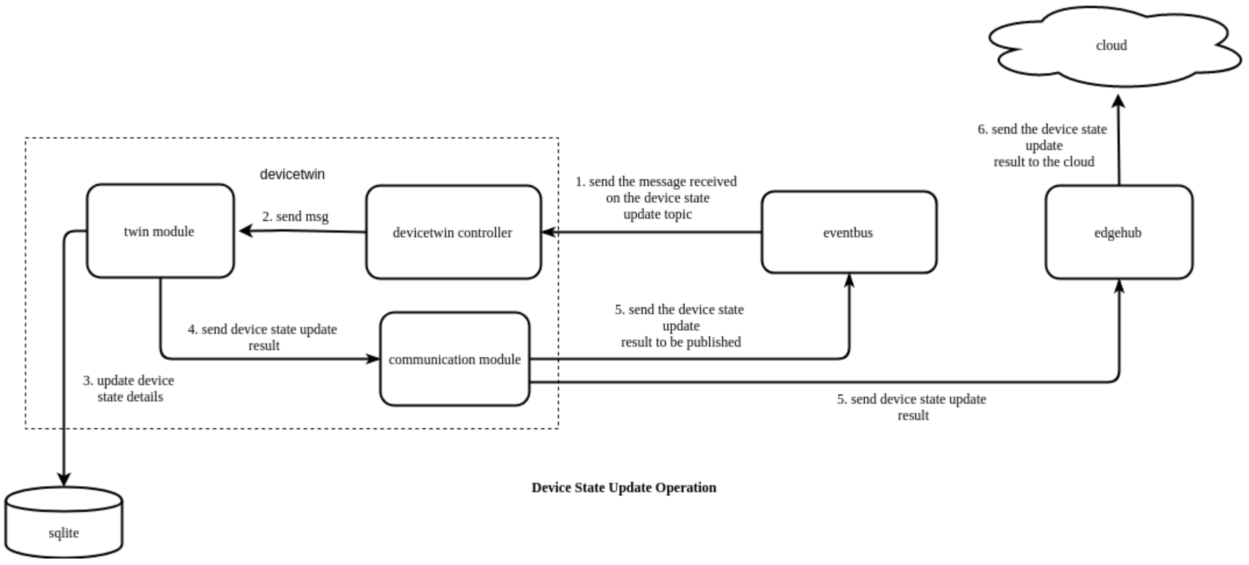
- dealDeviceStateUpdate

dealDeviceUpdated: dealDeviceUpdated() deals with the operations to be performed when a device attribute update is encountered. It updates the changes to the device attributes, like addition of attributes, updation of attributes and deletion of attributes, in the database. It also sends the result of the device attribute update to be published to the eventbus component through the communicate module of devicetwin. The eventbus component further publishes the result on the specified topic.


devicetwin
twin module
2. send msg
devicetwin controller
1. send the message received on the device update topic
eventbus
4. send device attribute update result
communication module
5. send device attribute update result to be published
3..update device attribute details
sqlite
Device Update Operation

Device Update

dealDeviceStateUpdate: dealDeviceStateUpdate() deals with the operations to be performed when a device status update is encountered. It updates the state of the device as well as the last online time of the device in the database. It also sends the update state result, through the communication module, to the cloud through the edgehub module and to the eventbus module which in turn publishes the result on the specified topic of the MQTT broker.


cloud
6. send the device state update result to the cloud
devicetwin
twin module
2. send msg
devicetwin controller
1. send the message received on the device state update topic
eventbus
edgehub
4. send device state update result
communication module
5. send the device state update result to be published
5. send device state update result
3. update device state details
sqlite
Device State Update Operation

Device State Update

11.4 Tables

DeviceTwin module creates three tables in the database, namely :-

- Device Table
- Device Attribute Table
- Device Twin Table

11.4.1 Device Table

Device table contains the data regarding the devices added to a particular edge node. The following are the columns present in the device table :

Operations Performed :-

The following are the operations that can be performed on this data :-

- **Save Device**: Inserts a device in the device table
- **Delete Device By ID**: Deletes a device by its ID from the device table
- **Update Device Field**: Updates a single field in the device table
- **Update Device Fields**: Updates multiple fields in the device table
- **Query Device**: Queries a device from the device table
- **Query Device All**: Displays all the devices present in the device table
- **Update Device Multi**: Updates multiple columns of multiple devices in the device table
- **Add Device Trans**: Inserts device, device attribute and device twin in a single transaction, if any of these operations fail, then it rolls back the other insertions
- **Delete Device Trans**: Deletes device, device attribute and device twin in a single transaction, if any of these operations fail, then it rolls back the other deletions

11.4.2 Device Attribute Table

Device attribute table contains the data regarding the device attributes associated with a particular device in the edge node. The following are the columns present in the device attribute table :

Operations Performed :-

The following are the operations that can be performed on this data :

- **Save Device Attr**: Inserts a device attribute in the device attribute table
- **Delete Device Attr By ID**: Deletes a device attribute by its ID from the device attribute table
- **Delete Device Attr**: Deletes a device attribute from the device attribute table by filtering based on device id and device name
- **Update Device Attr Field**: Updates a single field in the device attribute table
- **Update Device Attr Fields**: Updates multiple fields in the device attribute table
- **Query Device Attr**: Queries a device attribute from the device attribute table
- **Update Device Attr Multi**: Updates multiple columns of multiple device attributes in the device attribute table

- **Delete Device Attr Trans**: Inserts device attributes, deletes device attributes and updates device attributes in a single transaction.

11.4.3 Device Twin Table

Device twin table contains the data related to the device device twin associated with a particular device in the edge node. The following are the columns present in the device twin table :

Operations Performed :-

The following are the operations that can be performed on this data :-

- **Save Device Twin**: Inserts a device twin in the device twin table
- **Delete Device Twin By Device ID**: Deletes a device twin by its ID from the device twin table
- **Delete Device Twin**: Deletes a device twin from the device twin table by filtering based on device id and device name
- **Update Device Twin Field**: Updates a single field in the device twin table
- **Update Device Twin Fields**: Updates multiple fields in the device twin table
- **Query Device Twin**: Queries a device twin from the device twin table
- **Update Device Twin Multi**: Updates multiple columns of multiple device twins in the device twin table
- **Delete Device Twin Trans**: Inserts device twins, deletes device twins and updates device twins in a single transaction.

CHAPTER 12

Edge Controller

12.1 Edge Controller Overview

Controller(also known as edgecontroller) is the bridge between Kubernetes Api-Server and edgecore

12.2 Operations Performed By Edge Controller

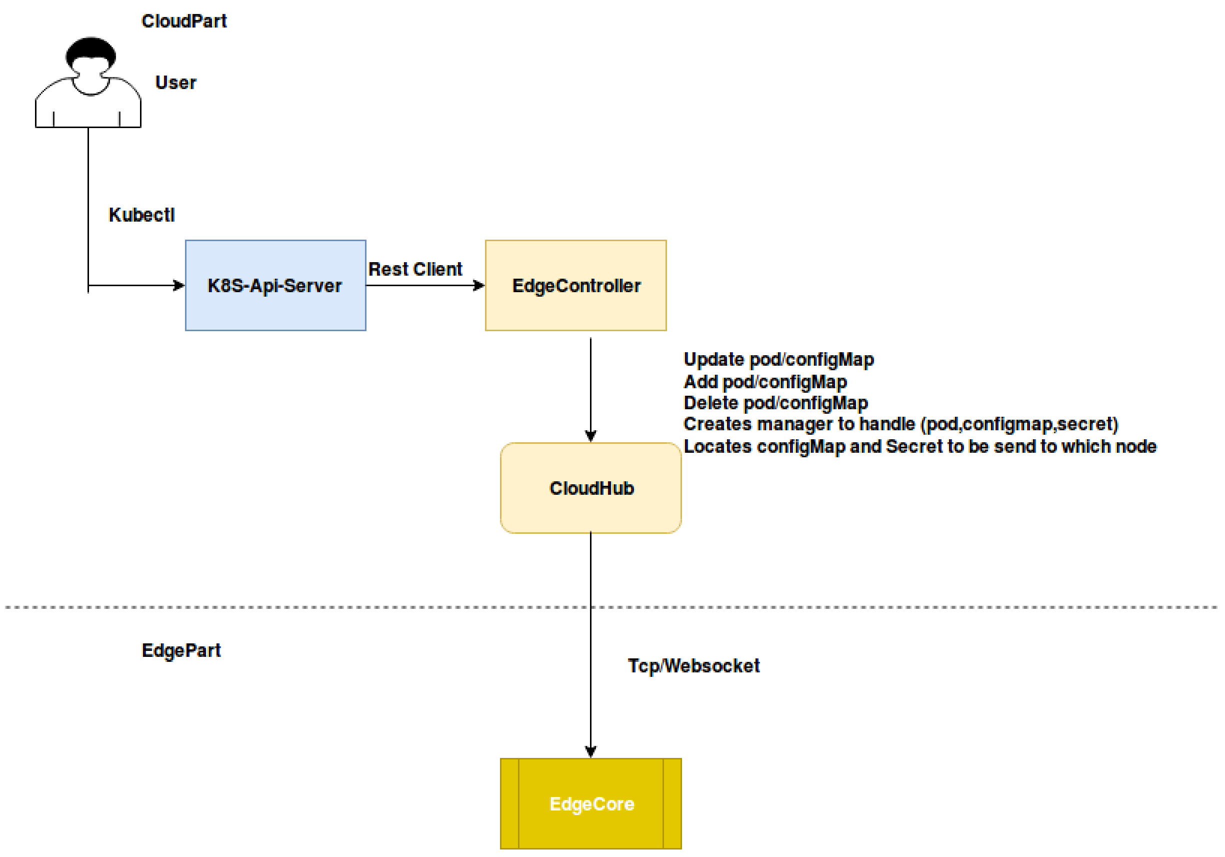
The following are the functions performed by Edge controller :-

- Downstream Controller:Sync add/update/delete event to edgecore from K8s Api-server
- Upstream Controller:Sync watch and Update status of resource and events(node, pod and configmap) to K8s-Api-server and also subscribe message from edgecore
- Controller Manager:Creates manager Interface which implements events for managing ConfigmapManager LocationCache and podManager

12.3 Downstream Controller:

12.3.1 Sync add/update/delete event to edge

- Downstream controller:Watches K8S-Api-server and send updates to edgecore via cloudHub
- Sync (pod,configmap,secret) add/update/delete event to edge via cloudHub
- Creates Respective manager (pod, configmap, secret) for handling events by calling manager interface
- Locates configmap and secret should be send to which node



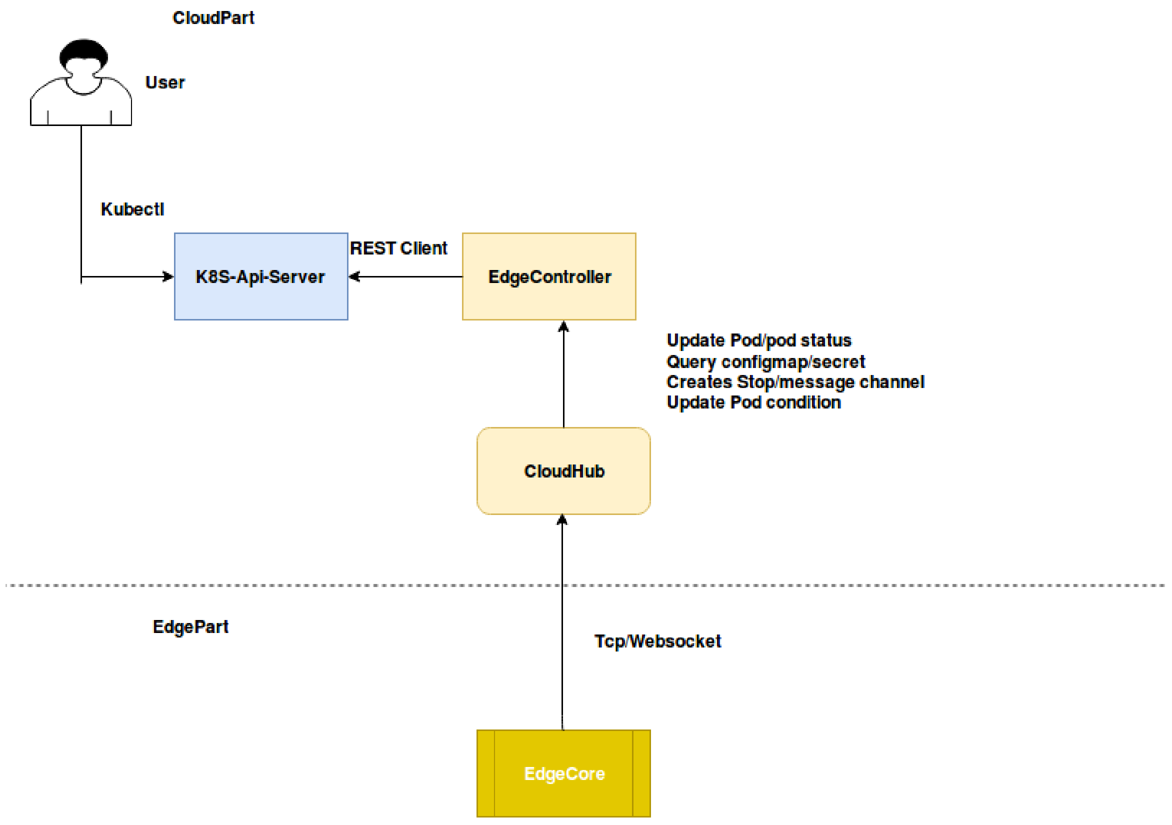
Downstream Controller

12.4 Upstream Controller:

12.4.1 Sync watch and Update status of resource and events

- UpstreamController receives messages from edgecore and sync the updates to K8S-Api-server
- Creates stop channel to dispatch and stop event to handle pods, configMaps, node and secrets
- Creates message channel to update Nodestatus, Podstatus, Secret and configmap related events
- Gets Podcondition information like Ready, Initialized, Podscheduled and Unschedulable details
- **Below is the information for PodCondition**
  - **Ready**:PodReady means the pod is able to service requests and should be added to the load balancing pools for all matching services
  - **PodScheduled**:It represents status of the scheduling process for this pod
  - **Unschedulable**:It means scheduler cannot schedule the pod right now, may be due to insufficient resources in the cluster
  - **Initialized**:It means that all Init containers in the pod have started sucessfully
  - **ContainersReady**:It indicates whether all containers in the pod are ready
- **Below is the information for PodStatus**

- **PodPhase**:Current condition of the pod
- **Conditions**:Details indicating why the pod is in this condition
- **HostIP**:IP address of the host to which pod is assigned
- **PodIp**:IP address allocated to the Pod
- **QosClass**:Assigned to the pod based on resource requirement



Upstream

Controller

12.5 Controller Manager:

12.5.1 Creates manager Interface and implements ConfigmapManager Location-Cache and podManager

- Manager defines the Interface of a manager, ConfigManager, Podmanager, secretmanager implements it
- Manages OnAdd, OnUpdate and OnDelete events which will be updated to the respective edge node from the K8s-Api-server
- Creates an eventManager(configMaps, pod, secrets) which will start a CommonResourceEventHandler, NewListWatch and a newShared Informer for each event to Sync(add/update/delete)event(pod, configmap, secret) to edgecore via cloudHub
- **Below is the List of handlers created by controller Manager**
  - **CommonResourceEventHandler**:NewcommonResourceEventHandler creates CommonResourceEventHandler used for Configmap and pod Manager

- **NewListWatch**:Creates a new ListWatch from the specified client resource namespace and field selector
- **NewSharedInformer**:Creates a new Instance for the Listwatcher

CHAPTER 13

CloudHub

13.1 CloudHub Overview

CloudHub is a web socket client, which is the mediator between EdgeController and the Edge side. Its function is enable the communication between edge and the EdgeController.

The connection to the edge(through EdgeHub module) is done through the HTTP over websocket connection. For internal communication it directly communicates with the EdgeController. All the request send to CloudHub are of context object which are stored in channelQ along with the mapped channels of event object marked to its nodeID.

The main functions performed by CloudHub are :-

- Get message context and create ChannelQ for events
- Create http connection over websocket
- Serve websocket connection
- Read message from edge
- Write message to edge
- Publish message to Controller

13.1.1 Get message context and create ChannelQ for events:

The context object is stored in a channelQ. For all nodeID channel is created and the message is converted to event object Event object is then passed through the channel.

13.1.2 Create http connection over websocket:

- TLS certificates are loaded through the path provided in the context object
- HTTP server is started with TLS configurations
- Then HTTP connection is upgraded to websocket connection receiving conn object.

- ServeConn function the serves all the incoming connections

13.1.3 Read message from edge:

- First a deadline is set for keepalive interval
- Then the JSON message from connection is read
- After that Message Router details are set
- Message is then converted to event object for cloud internal communication
- In the end the event is published to EdgeController

13.1.4 Write Message to Edge:

- First all event objects are received for the given nodeID
- The existence of same request and the liveness of the node is checked
- The event object is converted to message structure
- Write deadline is set. Then the message is passed to the websocket connection

13.1.5 Publish Message to EdgeController:

- A default message with timestamp, clientID and event type is sent to controller every time a request is made to websocket connection
- If the node gets disconnected then error is thrown and an event describing node failure is published to the controller.

CHAPTER 14

Pre-requisites

For best understanding of the guides, it's useful to have some knowledge of the following systems:

- Kubernetes
- Mosquitto
- Docker

CHAPTER 15

Setup KubeEdge

15.1 Prerequisites

To use KubeEdge, you will need to have **docker** installed. If you don't, please follow these steps to install docker.

15.2 Install docker

For ubuntu:

```
# Install Docker from Ubuntu's repositories:
apt-get update
apt-get install -y docker.io

# or install Docker CE 18.06 from Docker's repositories for Ubuntu or Debian:
apt-get update && apt-get install apt-transport-https ca-certificates curl software-
↪properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
apt-get update && apt-get install docker-ce=18.06.0~ce~3-0~ubuntu
```

For centOS:

```
# Install Docker from CentOS/RHEL repository:
yum install -y docker

# or install Docker CE 18.06 from Docker's CentOS repositories:
yum install yum-utils device-mapper-persistent-data lvm2
yum-config-manager \
    --add-repo \
```

(continues on next page)

(continued from previous page)

```
    https://download.docker.com/linux/centos/docker-ce.repo
yum update && yum install docker-ce-18.06.1.ce
```

KubeEdge's Cloud(edgecontroller) connects to Kubernetes master to sync updates of node/pod status. If you don't have Kubernetes setup, please follow these steps to install Kubernetes using kubeadm.

15.3 Install kubeadm/kubectl

For Ubuntu:

```
apt-get update && apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
apt-mark hold kubelet kubeadm kubectl
```

For CentOS:

```
at <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.
↪google.com/yum/doc/rpm-package-key.gpg
exclude=kube*
EOF

# Set SELinux in permissive mode (effectively disabling it)

setenforce 0
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config

yum install -y kubelet kubeadm kubectl --disableexcludes=kubernetes

systemctl enable --now kubelet
```

15.4 Install Kubernetes

To initialize Kubernetes master, follow the below step:

```
kubeadm init
```

After initializing Kubernetes master, we need to expose insecure port 8080 for edgecontroller/kubectl to work with http connection to api-server Please follow below steps to enable http port in apiserver

```
vi /etc/kubernetes/manifests/kube-apiserver.yaml
# Add the following flags in spec: containers: -command section
- --insecure-port=8080
- --insecure-bind-address=0.0.0.0
```

KubeEdge uses MQTT for communication between deviceTwin and devices. KubeEdge supports 3 MQTT modes:

- `0 - internalMqttMode`: internal mqtt broker is enabled
- `1 - bothMqttMode`: internal as well as external broker are enabled
- `2 - externalMqttMode`: only external broker is enabled

Use mode field in edge.yaml to select the desired mode

```
mqtt:
    server: tcp://127.0.0.1:1883 # external mqtt broker url.
    internal-server: tcp://127.0.0.1:1884 # internal mqtt broker url.
    mode: 0 # 0: internal mqtt broker enable only. 1: internal and external mqtt␣
↪broker enable. 2: external mqtt broker enable only.
    qos: 0 # 0: QOSAtMostOnce, 1: QOSAtLeastOnce, 2: QOSExactlyOnce.
    retain: false # if the flag set true, server will store the message and can be␣
↪delivered to future subscribers.
    session-queue-size: 100 # A size of how many sessions will be handled. default to␣
↪100.
```

To use kubeedge in double mqtt or external mode, make sure you have **mosquitto** in your environment. If you do not already have it, you may install as follows.

15.5 Install mosquitto

For ubuntu:

```
apt install mosquitto
```

For centOS:

```
yum install mosquitto
```

See mosquitto official website for more information.

15.6 Authentication

KubeEdge has certificate based authentication/authorization between cloud and edge. Certificates can be generated using openssl. Please follow the steps below to generate certificates.

15.6.1 Install openssl

If openssl is not already present using below command to install openssl

```
apt-get install openssl
```

15.6.2 Generate Certificates

RootCA certificate and a cert/key pair is required to have a setup for KubeEdge. Same cert/key pair can be used in both cloud and edge.

```
# Generete Root Key
openssl genrsa -des3 -out rootCA.key 4096
# Generate Root Certificate
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.crt
# Generate Key
openssl genrsa -out kubeedge.key 2048
# Generate csr, Fill required details after running the command
openssl req -new -key kubeedge.key -out kubeedge.csr
# Generate Certificate
openssl x509 -req -in kubeedge.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -
↪out kubeedge.crt -days 500 -sha256
```

15.7 Build

15.7.1 Clone KubeEdge

```
git clone https://github.com/kubeedge/kubeedge.git $GOPATH/src/github.com/kubeedge/
↪kubeedge
cd $GOPATH/src/github.com/kubeedge/kubeedge
```

15.7.2 Build Cloud

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/cloud/edgecontroller
make # or `make edgecontroller`
```

15.7.3 Build Edge

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/edge
make # or `make edgecontroller`
```

KubeEdge can also be cross compiled to run on ARM based processors. Please click Cross Compilation for the instructions.

15.8 Run KubeEdge

15.8.1 Run Cloud

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/cloud/edgecontroller
# run edge controller
# `conf/` should be in the same directory as the binary
# verify the configurations before running cloud(edgecontroller)
./edgecontroller
```

15.8.2 Run Edge

We have provided a sample node.json to add a node in kubernetes. Please make sure edge-node is added in kubernetes. Run below steps to add edge-node

```
kubectl apply -f $GOPATH/src/github.com/kubeedge/kubeedge/build/node.json
```

```
# run mosquitto
mosquitto -d -p 1883

# run edge_core
# `conf/` should be in the same directory as the binary
# verify the configurations before running edge(edge_core)
./edge_core
# or
nohup ./edge_core > edge_core.log 2>&1 &
```

If you are using HuaweiCloud IEF, then the edge node you created should be running (check it in the IEF console page).

15.9 Deploy Application

Try out a sample application deployment by following below steps

```
kubectl apply -f $GOPATH/src/github.com/kubeedge/kubeedge/build/deployment.yaml
```

Note: Currently, for edge node, we must use hostPort in the Pod container spec so that the pod comes up normally, or the pod will be always in *ContainerCreating* status. The hostPort must be equal to containerPort and can not be 0.

15.10 Run Edge Unit Tests

```
make edge_test
```

To run unit tests of a package individually

```
export GOARCHAIUS_CONFIG_PATH=$GOPATH/src/github.com/kubeedge/kubeedge/edge
cd <path to package to be tested>
go test -v
```

15.11 Run Edge Integration Tests

```
make edge_integration_test
```

15.11.1 Details and use cases of integration test framework

Please find the link to use cases of intergration test framework for kubeedge

CHAPTER 16

Cross Compiling KubeEdge

16.1 For ARM Architecture from x86 Architecture

Clone KubeEdge

```
# Build and run KubeEdge on a ARMv6 target device.

git clone https://github.com/kubeedge/kubeedge.git $GOPATH/src/github.com/kubeedge/
↪kubeedge
cd $GOPATH/src/github.com/kubeedge/kubeedge/edge
sudo apt-get install gcc-arm-linux-gnueabi
export GOARCH=arm
export GOOS="linux"
export GOARM=6                          #Pls give the appropriate arm version of␣
↪your device
export CGO_ENABLED=1
export CC=arm-linux-gnueabi-gcc
make # or `make edge_core`
```

CHAPTER 17

Try KubeEdge with HuaweiCloud (IEF)

17.1 Intelligent EdgeFabric (IEF)

Note: The HuaweiCloud IEF is only available in China now.

1. Create an account in HuaweiCloud.
2. Go to IEF and create an Edge node.
3. Download the node configuration file (<node_name>.tar.gz).
4. Run `cd $GOPATH/src/github.com/kubeedge/kubeedge/edge` to enter edge directory.
5. Run `bash -x hack/setup_for_IEF.sh /PATH/TO/<node_name>.tar.gz` to modify the configuration files in `conf/`.

CHAPTER 18

MQTT Message Topics

KubeEdge uses MQTT for communication between deviceTwin and devices/apps. EventBus can be started in multiple MQTT modes and acts as an interface for sending/receiving messages on relevant MQTT topics.

The purpose of this document is to describe the topics which KubeEdge uses for communication. Please read Beehive documentation for understanding about message format used by KubeEdge.

18.1 Subscribe Topics

On starting EventBus, it subscribes to these 5 topics:

```
1. "$hw/events/node/+/membership/get"
2. "$hw/events/device/+/state/update"
3. "$hw/events/device/+/twin/+"
4. "$hw/events/upload/#"
5. "SYS/dis/upload_records"
```

If the the message is received on first 3 topics, the message is sent to deviceTwin, else the message is sent to cloud via edgeHub.

We will focus on the message expected on the first 3 topics.

1. `"$hw/events/node/+/membership/get"`: This topics is used to get membership details of a node i.e the devices that are associated with the node. The response of the message is published on `"$hw/events/node/+/membership/get/result"` topic.

2. `"$hw/events/device/+/state/update`": This topic is used to update the state of the device. + symbol can be replaced with ID of the device whose state is to be updated.

3. `"$hw/events/device/+/twin/+"`: The two + symbols can be replaced by the deviceID on whose twin the operation is to be performed and any one of(update,cloud_updated,get) respectively.

Following is the explanation of the three suffix used:

1. `update`: this suffix is used to update the twin for the deviceID.

2. `cloud_updated`: this suffix is used to sync the twin status between edge and cloud.
3. `get`: is used to get twin status of a device. The response is published on `"$hw/events/device/+/twin/get/result"` topic.

CHAPTER 19

Unit Test Guide

The purpose of this document is to give introduction about unit tests and to help contributors in writing unit tests.

19.1 Unit Test

Read this article for a simple introduction about unit tests and benefits of unit testing. Go has its own built-in package called testing and command called `go test`.For more detailed information on golang's builtin testing package read this document.

19.2 Mocks

The object which needs to be tested may have dependencies on other objects. To confine the behavior of the object under test, replacement of the other objects by mocks that simulate the behavior of the real objects is necessary. Read this article for more information on mocks.

GoMock is a mocking framework for Go programming language. Read godoc for more information about gomock.

Mock for an interface can be automatically generated using GoMocks mockgen package.

Note There is gomock package in kubeedge vendor directory without mockgen. Please use mockgen package of tagged version ***v1.1.1*** of GoMocks github repository to install mockgen and generate mocks. Using higher version may cause errors/panics during execution of you tests.

There is gomock package in kubeedge vendor directory without mockgen. Please use mockgen package of tagged version ***v1.1.1*** of GoMocks github repository to install mockgen and generate mocks. Using higher version may cause errors/panics during execution of you tests.

Read this article for a short tutorial of usage of gomock and mockgen.

19.3 Ginkgo

Ginkgo is one of the most popular framework for writing tests in go.

Read godoc for more information about ginkgo.

See a sample in kubeedge where go builtin package testing and gomock is used for writing unit tests.

See a sample in kubeedge where ginkgo is used for testing.

19.4 Writing UT using GoMock

19.4.1 Example : metamanager/dao/meta.go

After reading the code of meta.go, we can find that there are 3 interfaces of beego which are used. They are Ormer, QuerySeter and RawSeter.

We need to create fake implementations of these interfaces so that we do not rely on the original implementation of this interface and their function calls.

Following are the steps for creating fake/mock implementation of Ormer, initializing it and replacing the original with fake.

1. Create directory mocks/beego.
2. use mockgen to generate fake implementation of the Ormer interface

```
mockgen -destination=mocks/beego/fake_ormer.go -package=beego github.com/astaxie/
↪beego/orm Ormer
```

- `destination` : where you want to create the fake implementation.
- `package` : package of the created fake implementation file
- `github.com/astaxie/beego/orm` : the package where interface definition is there
- `Ormer` : generate mocks for this interface

1. Initialize mocks in your test file. eg meta_test.go

```
mockCtrl := gomock.NewController(t)
defer mockCtrl.Finish()
ormerMock = beego.NewMockOrmer(mockCtrl)
```

1. ormermock is now a fake implementation of Ormer interface. We can make any function in ormermock return any value you want.
2. replace the real Ormer implementation with this fake implementation. DBAccess is variable to type Ormer which we will replace with mock implemention

```
dbm.DBAccess = ormerMock
```

1. If we want Insert function of ormer interface which has return types as (int64,err) to return (1 nil), it can be done in 1 line in your test file using gomock.

```
ormerMock.EXPECT().Insert(gomock.Any()).Return(int64(1), nil).Times(1)
```

`Expect()` : is to tell that a function of ormermock will be called.

`Insert(gomock.Any())` : expect Insert to be called with any parameter.

`Return(int64(1), nil)` : return 1 and error nil

`Times(1)`: expect insert to be called once and return 1 and nil only once.

So whenever insert is called, it will return 1 and nil, thus removing the dependency on external implementation.

CHAPTER 20

FAQs

This page contains a few commonly occuring questions. For further support please contact us using the support page