KubeEdge Documentation

Release 0.1

KubeEdge

Aug 20, 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 onwards

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

- Istio-based service mesh across Edge and Cloud where micro-services can communicate freely in the mesh.
- Enhance performance and reliability of KubeEdge infrastructure.
- Enable function as a service at the Edge.
- Support more types of device protocols to Edge nodes such as AMQP, BlueTooth, ZigBee, etc.
- Evaluate and enable much larger scale Edge clusters with thousands of Edge nodes and millions of devices.
- Enable intelligent scheduling of applications to large scale Edge clusters.
- Data management/analytics framework.

3.2 Release 0.3

- Device CRD API and management framework.
- Performance test framework.
- KubeEdge installer.
- CRI support.

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

KubeEdge Community Membership

Note : This document keeps changing based on the status and feedback of KubeEdge Community.

This document gives a brief overview of the KubeEdge community roles with the requirements and responsibilities associated with them.

Note : It is mandatory for all KubeEdge community members to follow KubeEdge Code of Conduct.

5.1 Member

Members are active participants in the community who contribute by authoring PRs, reviewing issues/PRs or participate in community discussions on slack/mailing list.

5.1.1 Requirements

- Sponsor from 2 approvers
- Enabled two-factor authentication on their GitHub account
- Actively contributed to the community. Contributions may include, but are not limited to:
  - Authoring PRs
  - Reviewing issues/PRs authored by other community members
  - Participating in community discussions on slack/mailing list
  - Participate in KubeEdge community meetings

5.1.2 Responsibilities and privileges

- Member of the KubeEdge GitHub organization
- Can be assigned to issues and PRs and community members can also request their review

- Participate in assigned issues and PRs
- Welcome new contributors
- Guide new contributors to relevant docs/files
- Help/Motivate new members in contributing to KubeEdge

5.2 Approver

Approvers are active members who have good experience and knowledge of the domain. They have actively participated in the issue/PR reviews and have identified relevant issues during review.

5.2.1 Requirements

- Sponsor from 2 maintainers
- Member for at least 2 months
- Have reviewed good number of PRs
- Have good codebase knowledge

5.2.2 Responsibilities and Privileges

- Review code to maintain/improve code quality
- Acknowledge and work on review requests from community members
- May approve code contributions for acceptance related to relevant expertise
- Have 'write access' to specific packages inside a repo, enforced via bot
- Continue to contribute and guide other community members to contribute in KubeEdge project

5.3 Maintainer

Maintainers are approvers who have shown good technical judgement in feature design/development in the past. Has overall knowledge of the project and features in the project.

5.3.1 Requirements

- Sponsor from 2 owners
- Approver for at least 2 months
- Nominated by a project owner
- Good technical judgement in feature design/development

5.3.2 Responsibilities and privileges

- Participate in release planning
- Maintain project code quality
- Ensure API compatibility with forward/backward versions based on feature graduation criteria
- Analyze and propose new features/enhancements in KubeEdge project
- Demonstrate sound technical judgement
- Mentor contributors and approvers
- Have top level write access to relevant repository (able click Merge PR button when manual check-in is necessary)
- Name entry in Maintainers file of the repository
- Participate & Drive design/development of multiple features

5.4 Owner

Owners are maintainers who have helped drive the overall project direction. Has deep understanding of KubeEdge and related domain and facilitates major agreement in release planning

5.4.1 Requirements

- Sponsor from 3 owners
- Maintainer for at least 2 months
- Nominated by a project owner
- Not opposed by any project owner
- Helped in driving the overall project

5.4.2 Responsibilities and Privileges

- Make technical decisions for the overall project
- Drive the overall technical roadmap of the project
- Set priorities of activities in release planning
- Guide and mentor all other community members
- Ensure all community members are following Code of Conduct
- Although given admin access to all repositories, make sure all PRs are properly reviewed and merged
- May get admin access to relevant repository based on requirement
- Participate & Drive design/development of multiple features

Note : These roles are applicable only for KubeEdge github organization and repositories. Currently KubeEdge doesn't have a formal process for review and acceptance into these roles. We will come-up with a process soon.

CHAPTER 6

Setup using Release package

6.1 Prerequisites

- Install docker
- Install kubeadm/kubectl
- Creating cluster with kubeadm
- After initializing Kubernetes master, we need to expose insecure port 8080 for edgecontroller/kubectl to work with http connection to Kubernetes apiserver. Please follow below steps to enable http port in Kubernetes 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
```

- (**Optional**)KubeEdge also supports https connection to Kubernetes apiserver. Follow the steps in Kubernetes Documentation to create the kubeconfig file.

  Enter the path to kubeconfig file in controller.yaml

```
controller:
  kube:
    ...
    kubeconfig: "path_to_kubeconfig_file" #Enter path to kubeconfig file to␣
↪enable https connection to k8s apiserver
```

6.2 Cloud Vm

Note:execute the below commands as root user

```
VERSION="v0.3.0"
OS="linux"
ARCH="amd64"
curl -L "https://github.com/kubeedge/kubeedge/releases/download/${VERSION}/kubeedge-$
↪{VERSION}-${OS}-${ARCH}.tar.gz" --output kubeedge-${VERSION}-${OS}-${ARCH}.tar.gz &&
↪ tar -xf kubeedge-${VERSION}-${OS}-${ARCH}.tar.gz  -C /etc
```

6.2.1 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.

```
wget -L https://github.com/kubeedge/kubeedge/blob/master/build/tools/certgen.sh
# make script executable
chmod +x certgen.sh
bash -x ./certgen.sh genCertAndKey edge
```

NOTE: The cert/key will be generated in the `/etc/kubeedge/ca` and `/etc/kubeedge/certs` respectively.

- The path to the generated certificates should be updated in `etc/kubeedge/cloud/conf/controller.yaml`. Please update the correct paths for the following :
  - cloudhub.ca
  - cloudhub.cert
  - cloudhub.key
- Create device model and device CRDs.

```
    wget -L https://github.com/kubeedge/kubeedge/blob/master/build/crds/devices/
↪devices_v1alpha1_devicemodel.yaml
    # make script executable
    chmod +x devices_v1alpha1_devicemodel.yaml
    kubectl create -f devices_v1alpha1_devicemodel.yaml
    wget -L https://github.com/kubeedge/kubeedge/blob/master/build/crds/devices/
↪devices_v1alpha1_device.yaml
     # make script executable
    chmod +x devices_v1alpha1_device.yaml
    kubectl create -f devices_v1alpha1_device.yaml
```

- Run cloud

```
    cd /etc/kubeedge/cloud
    # run edge controller
    # `conf/` should be in the same directory where edgecontroller resides
    # verify the configurations before running cloud(edgecontroller)
    ./edgecontroller
```

6.3 Edge Vm

6.3.1 Prerequisites

- Install Docker and/or Containerd based on the runtime to be used at edge

6.3.2 Configuring MQTT mode

The Edge part of KubeEdge uses MQTT for communication between deviceTwin and devices. KubeEdge supports 3 MQTT modes:

1. internalMqttMode: internal mqtt broker is enabled.
2. bothMqttMode: internal as well as external broker are enabled.
3. externalMqttMode: only external broker is enabled.

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

To use KubeEdge in double mqtt or external mode, you need to make sure that mosquitto or emqx edge is installed on the edge node as an MQTT Broker.

- 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.
- Deploy node `shell wget -L https://github.com/kubeedge/kubeedge/blob/master/build/node.json #Modify the node.json` file and change `metadata.name` to the name of the edge node kubectl apply -f node.json`
- Modify the `/etc/kubeedge/edge/conf/edge.yaml` configuration file
  - Replace `edgehub.websocket.certfile` and `edgehub.websocket.keyfile` with your own certificate path
  - Update the IP address of the master in the `websocket.url` field.
  - replace `fb4ebb70-2783-42b8-b3ef-63e2fd6d242e` with edge node name in edge.yaml for the below fields :
    * `websocket:URL`
    * `controller:node-id`
    * `edged:hostname-override`
  - Configure the desired container runtime in /etc/kubeedge/edge/conf/edge.yaml configuration file
  - Specify the runtime type to be used as either docker or remote (for all CRI based runtimes including containerd). If this parameter is not specified docker runtime will be used by default
    * `runtime-type:docker` or `runtime-type:remote`
  - Additionally specify the following parameters for remote/CRI based runtimes
    * `remote-runtime-endpoint:/var/run/containerd/containerd.sock`
    * `remote-image-endpoint:/var/run/containerd/containerd.sock`
    * `runtime-request-timeout:  2`
    * `podsandbox-image:  k8s.gcr.io/pause`
    * `kubelet-root-dir:  /var/run/kubelet/`
- Run edge

```
# run edge_core
    # `conf/` should be in the same directory as the cloned KubeEdge repository
    cd /etc/kubeedge/edge
    # verify the configurations before running edge(edge_core)
    ./edge_core
    # or
```

(continues on next page)

(continued from previous page)

```
        nohup ./edge_core > edge_core.log 2>&1 &
```

```
**Note**: Running edge_core on ARM based processors,follow the above steps as␣
↪mentioned for Edge Vm
```

```
    VERSION="v0.3.0"
    OS="linux"
    ARCH="arm"
    curl -L "https://github.com/kubeedge/kubeedge/releases/download/${VERSION}/
↪kubeedge-${VERSION}-${OS}-${ARCH}.tar.gz" --output kubeedge-${VERSION}-${OS}-${ARCH}
↪.tar.gz && tar -xf kubeedge-${VERSION}-${OS}-${ARCH}.tar.gz  -C /etc
```

- Monitoring containers status
  - If the container runtime configured to manage containers is containerd , then the following commands can be used to inspect container status and list images.
    - sudo ctr –namespace k8s.io containers ls
    - sudo ctr –namespace k8s.io images ls
    - sudo crictl exec -ti /bin/bash

NOTE: scp kubeedge folder from cloud vm to edge vm

```
In cloud
scp -r /etc/kubeedge root@edgeip:/etc
```

CHAPTER 7

Reporting bugs

If any part of the kubeedge project has bugs or documentation mistakes, please let us know by opening an issue. We treat bugs and mistakes very seriously and believe no issue is too small. Before creating a bug report, please check that an issue reporting the same problem does not already exist.

To make the bug report accurate and easy to understand, please try to create bug reports that are:

- Specific. Include as much details as possible: which version, what environment, what configuration, etc. If the bug is related to running the kubeedge server, please attach the kubeedge log (the starting log with kubeedge configuration is especially important).
- Reproducible. Include the steps to reproduce the problem. We understand some issues might be hard to reproduce, please includes the steps that might lead to the problem.
- Isolated. Please try to isolate and reproduce the bug with minimum dependencies. It would significantly slow down the speed to fix a bug if too many dependencies are involved in a bug report.
- Unique. Do not duplicate existing bug report.
- Scoped. One bug per report. Do not follow up with another bug inside one report.

We might ask for further information to locate a bug. A duplicated bug report will be closed.

CHAPTER 8

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.

8.1 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**

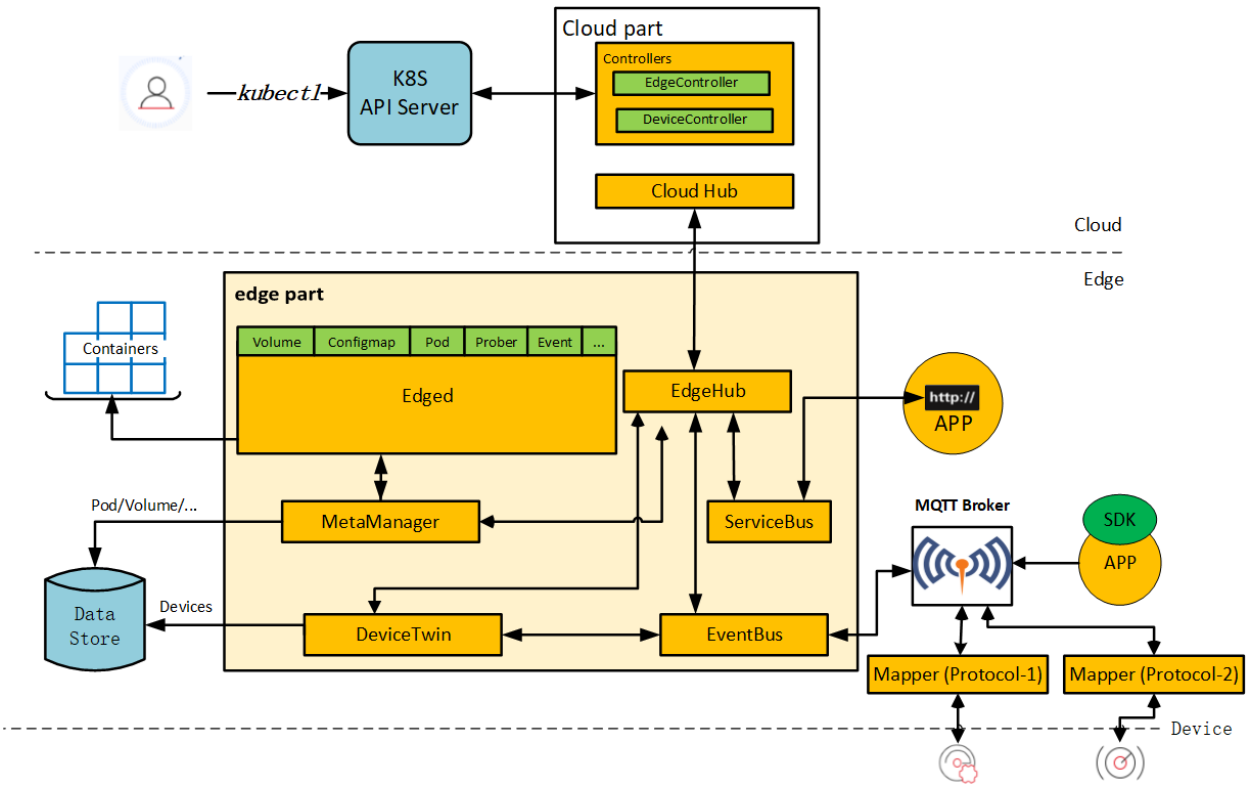
  It is easy to get and deploy existing complicated machine learning, image recognition, event processing and other high level applications to the Edge.

8.2 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).

8.3 Architecture



KubeEdge Architecture

8.4 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 9

Beehive

9.1 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

9.2 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{})

9.3 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.

9.4 Channel Context Structure Fields

9.4.1 (*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 a 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

9.5 Module Operations

9.5.1 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")
```

9.5.2 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”)
```

9.5.3 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”)
```

9.6 Message Operations

9.6.1 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)
```

9.6.2 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.
```

9.6.3 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")
```

9.6.4 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)
```

9.6.5 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)
```

9.6.6 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 10

EdgeD

10.1 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.

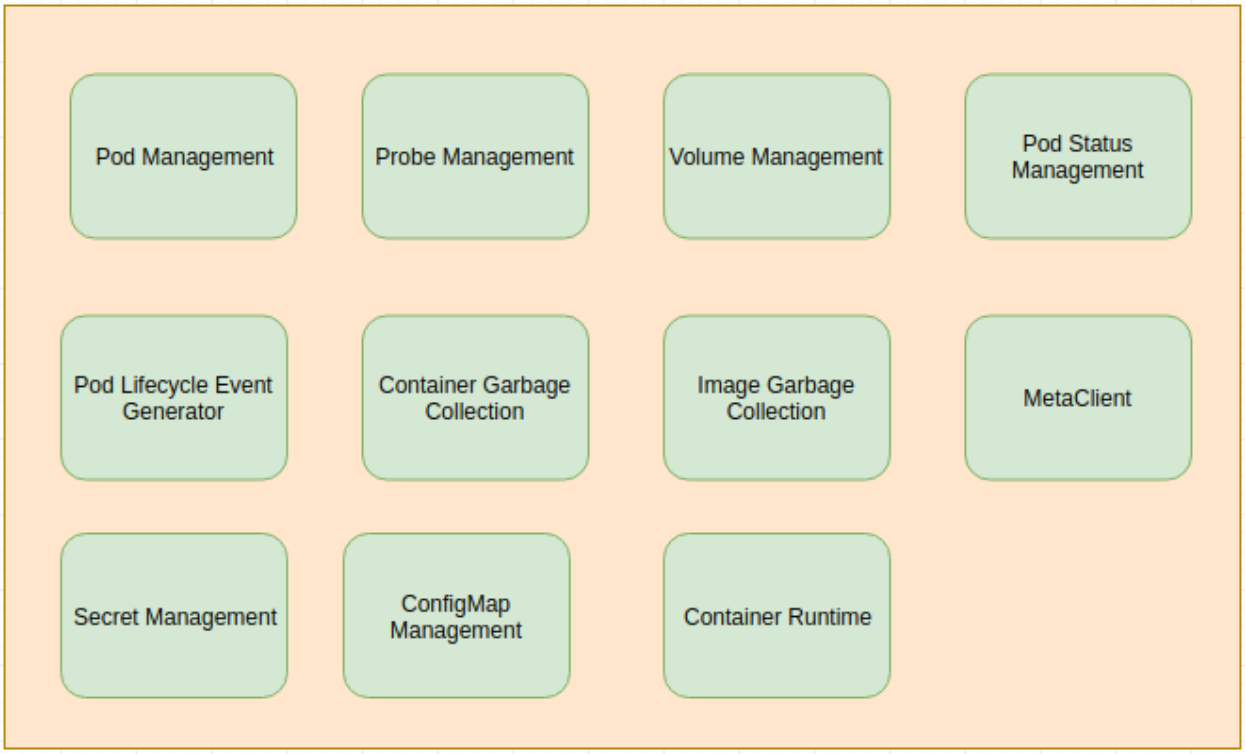
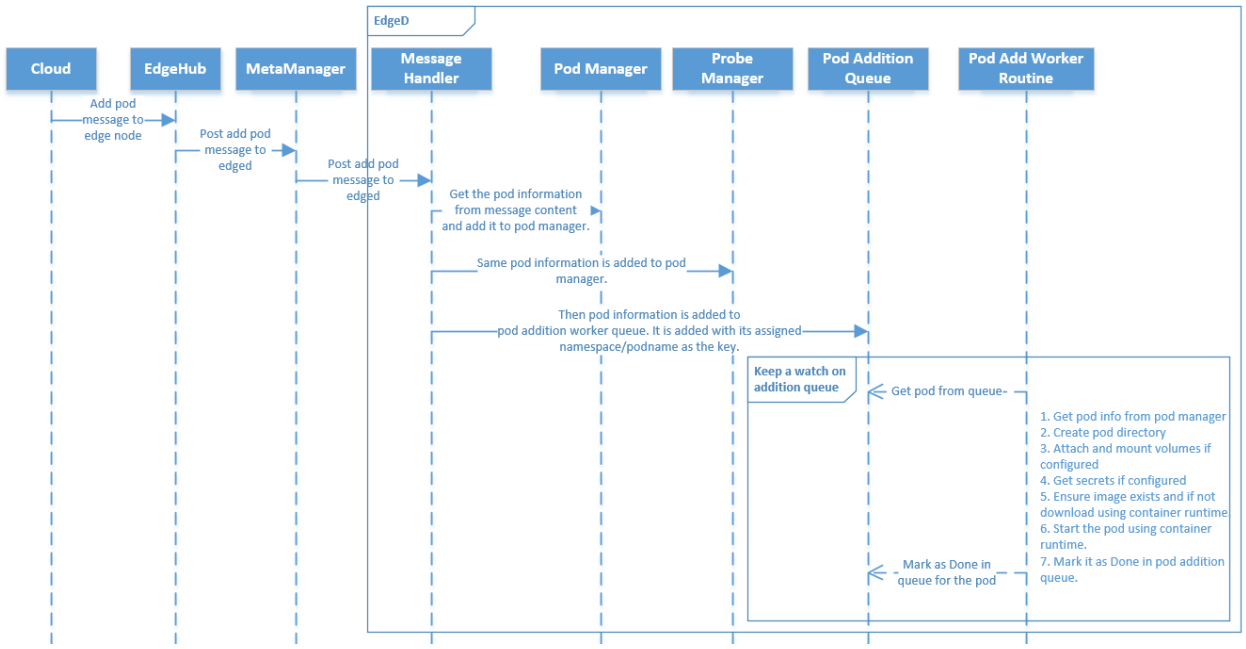


Fig 1: EdgeD Functionalities

10.2 Pod Management

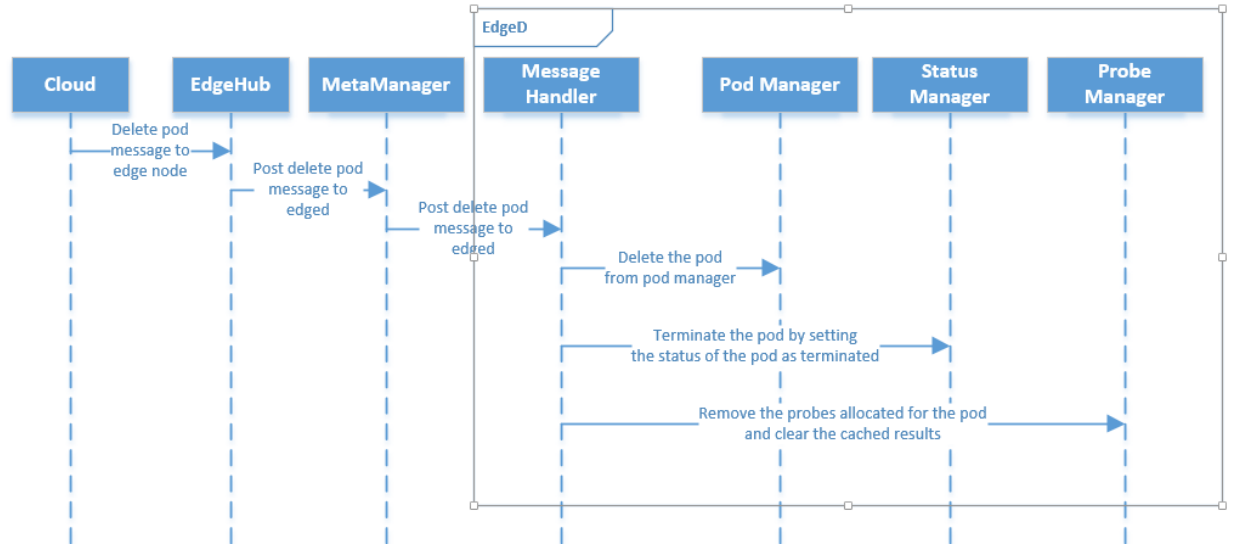
It is handles for pod addition, deletion and modification. It also tracks the health of the pods using pod status manager and pleg. Its primary jobs are as follows:

- 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



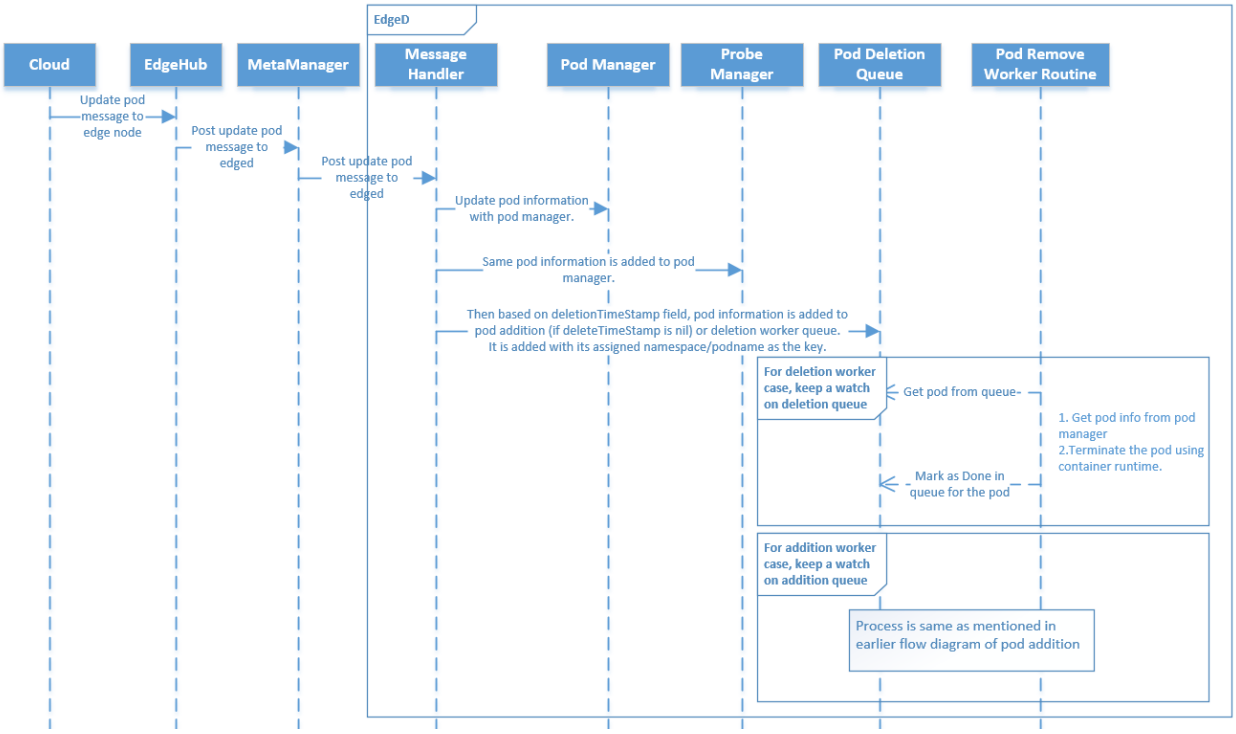
Pod Addition Flow

Fig 2: Pod Addition Flow



Pod Deletion Flow

Fig 3: Pod Deletion Flow

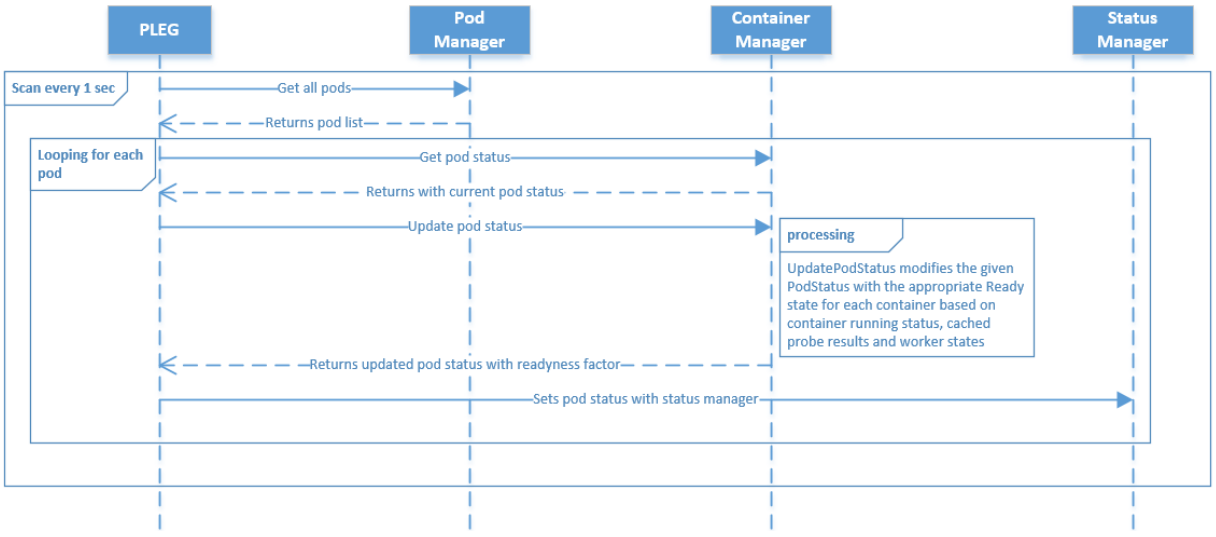


Pod Updation Flow

Fig 4: Pod Updation Flow

10.3 Pod Lifecycle Event Generator

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



PLEG Design

Fig 5: PLEG at EdgeD

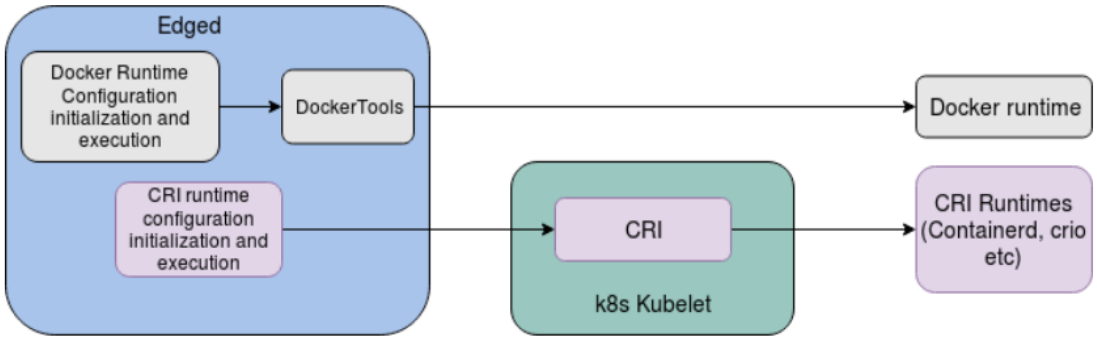
10.4 CRI for edged

Container Runtime Interface (CRI) – a plugin interface which enables edged to use a wide variety of container runtimes, without the need to recompile and also support multiple runtimes like docker, containerd, cri-o etc

10.4.1 Why CRI for edge?

Currently kubeedge edged supports only docker runtime using the legacy dockertools.

- CRI support for multiple container runtime in kubeedge is needed due to below mentioned factors
  - Include CRI support as in kubernetes kubelet to support containerd, cri-o etc
  - Continue with docker runtime support using legacy dockertools until CRI support for the same is available i.e. support for docker runtime using dockershim is not considered in edged
  - Support light weight container runtimes on resource constrained edge node which are unable to run the existing docker runtime
  - Support multiple container runtimes like docker, containerd, cri-o etc on the edge node.
  - Support for corresponding CNI with pause container and IP will be considered later
  - Customer can run light weight container runtime on resource constrained edge node that cannot run the existing docker runtime
  - Customer has the option to choose from multiple container runtimes on his edge platform

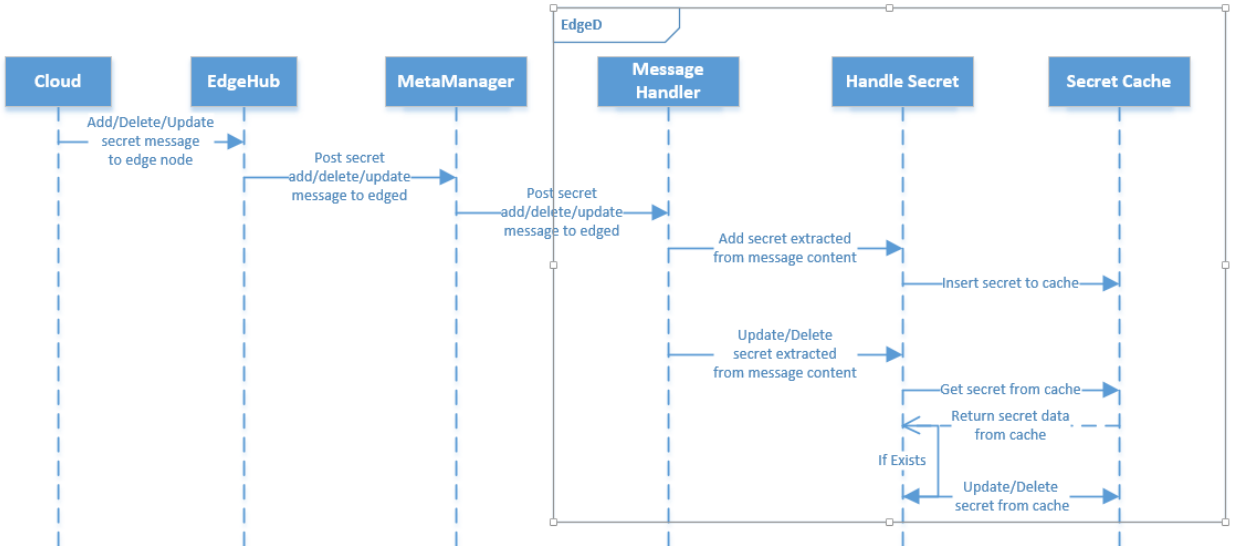


CRI Design

Fig 6: CRI at EdgeD

10.5 Secret Management

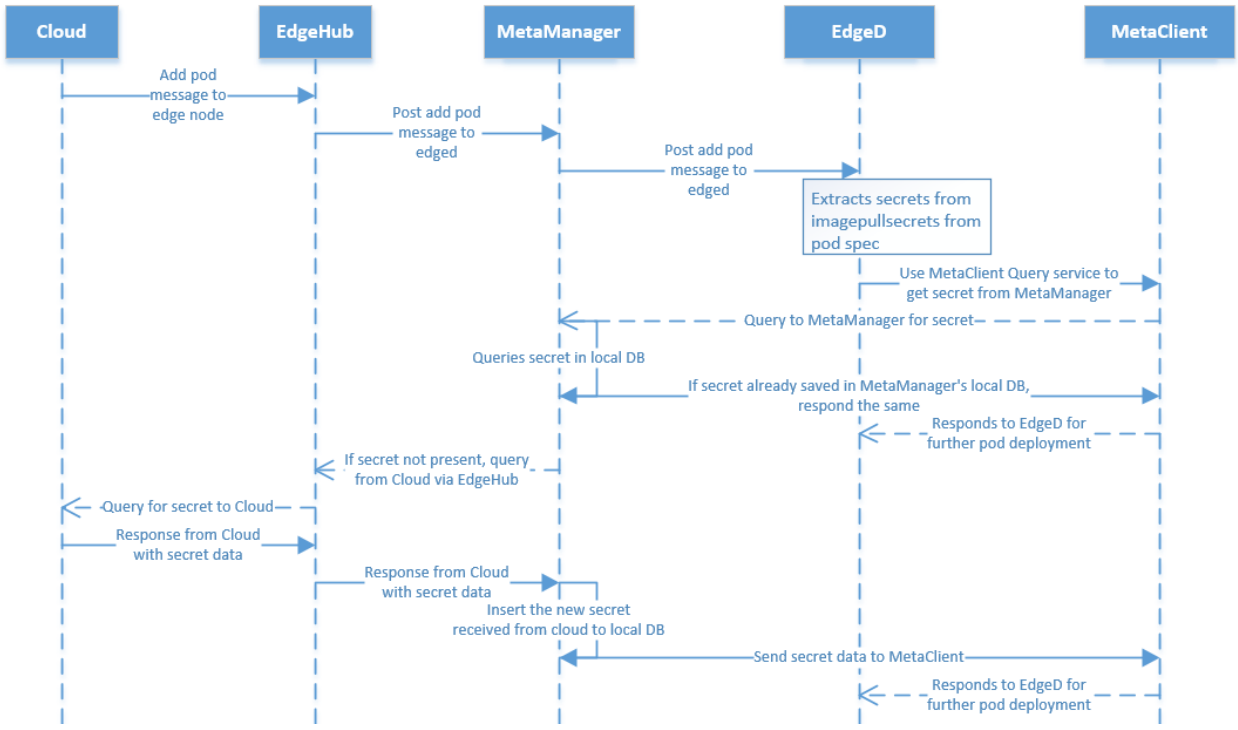
At edged, Secrets are handled separately. For its operations like addition, deletion and modifications; there are separate set of config messages or interfaces. Using these interfaces, secrets are updated in cache store. Below flow diagram explains the message flow.



Secret Message Handling

Fig 7: 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 forwared 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.



Query Secret

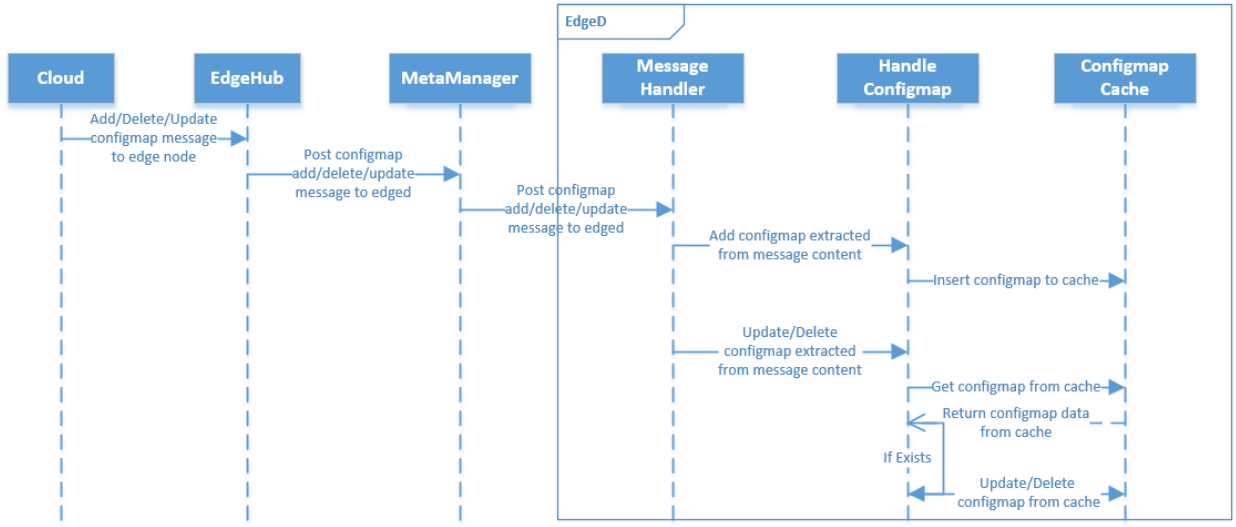
Fig 8: Query Secret by EdgeD

10.6 Probe Management

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

10.7 ConfigMap Management

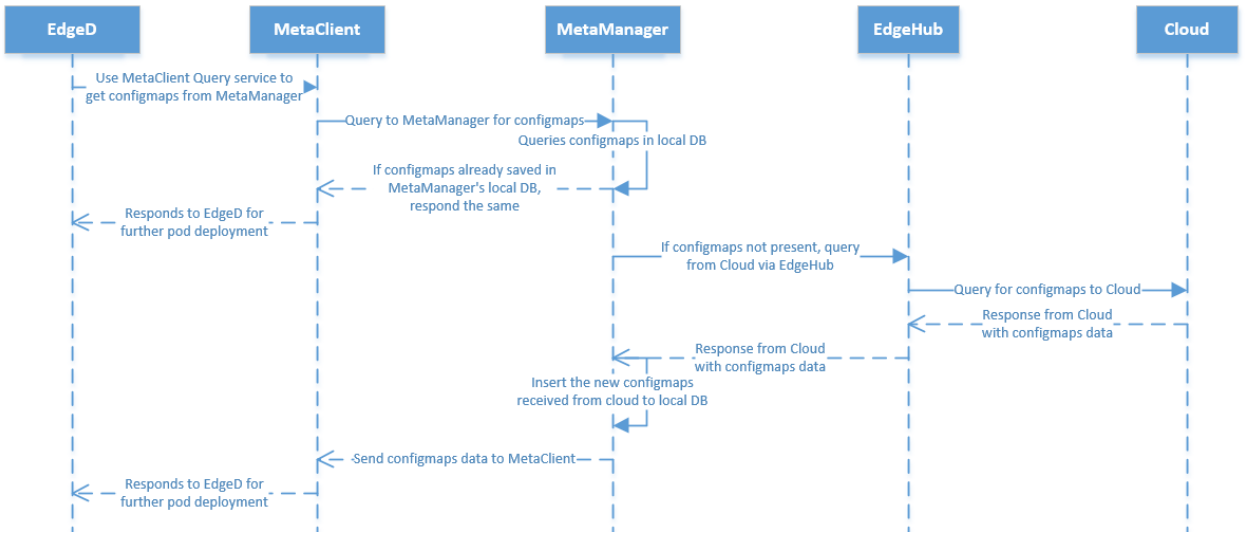
At edged, ConfigMap are also handled separately. For its operations like addition, deletion and modifications; there are separate set of config messages or interfaces. Using these interfaces, configMaps are updated in cache store. Below flow diagram explains the message flow.



ConfigMap Message Handling

Fig 9: 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.



Query Configmaps

Fig 10: Query Configmaps by EdgeD

10.8 Container GC

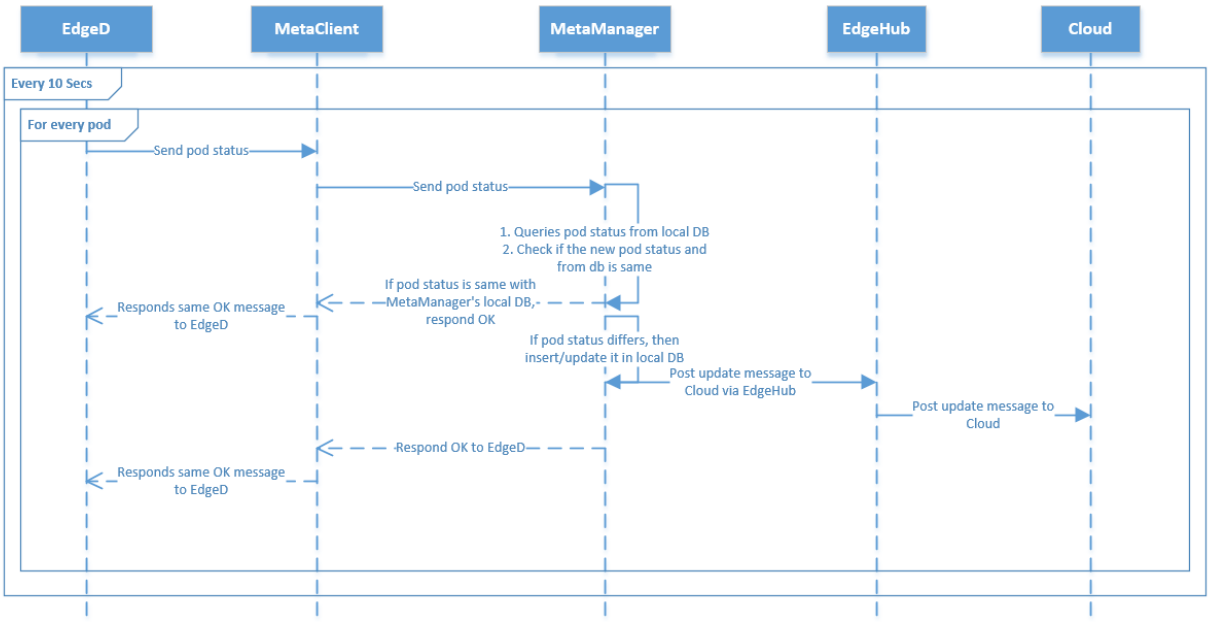
Container garbage collector is an edged routine which wakes up every minute, collecting and removing dead containers using the specified container gc policy. The policy for garbage collecting containers we apply takes on three variables, which can be user-defined. MinAge is the minimum age at which a 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. Generally, the oldest containers are removed first.

10.9 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.

10.10 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.



Status Manager Flow

Fig 11: Status Manager Flow

10.11 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.

10.12 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 11

EventBus

11.1 Overview

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

It supports 3 kinds of mode:

- internalMqttMode
- externalMqttMode
- bothMqttMode

11.2 Topic

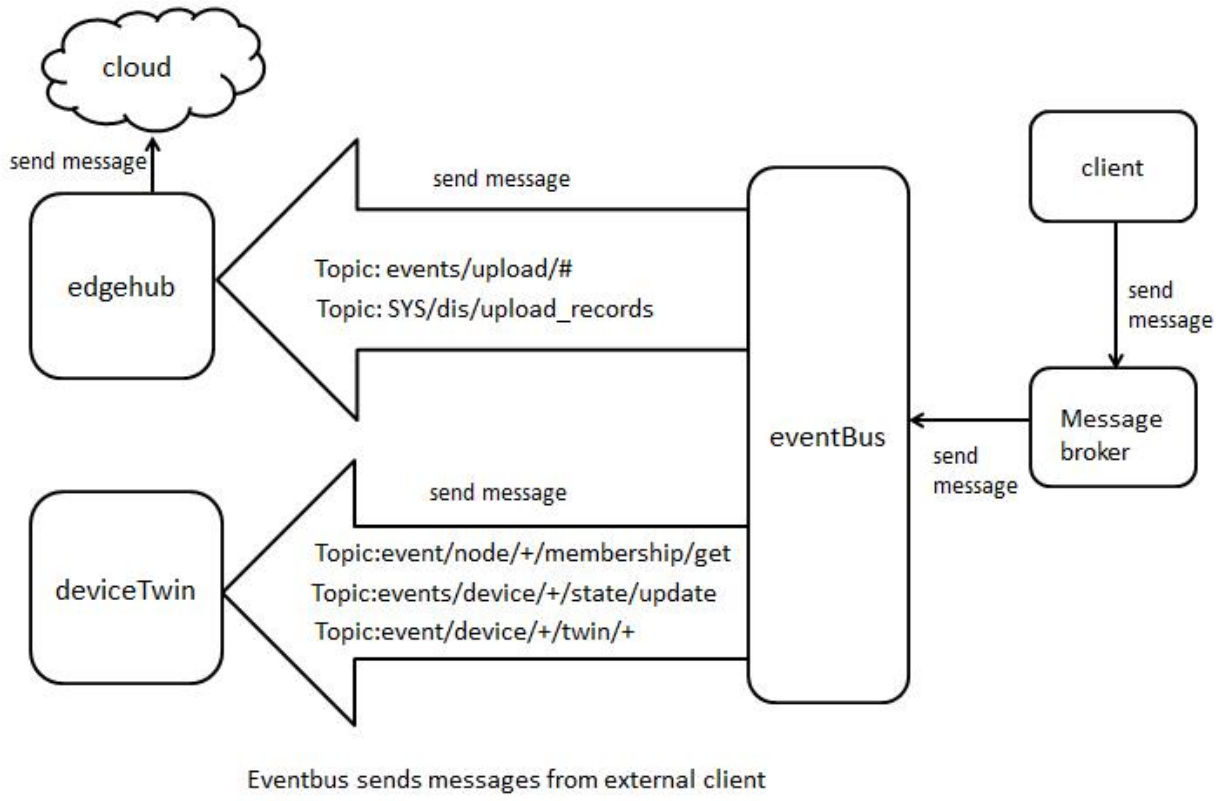
eventbus subscribes to the following topics:

```
- $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

11.3 Flow chart

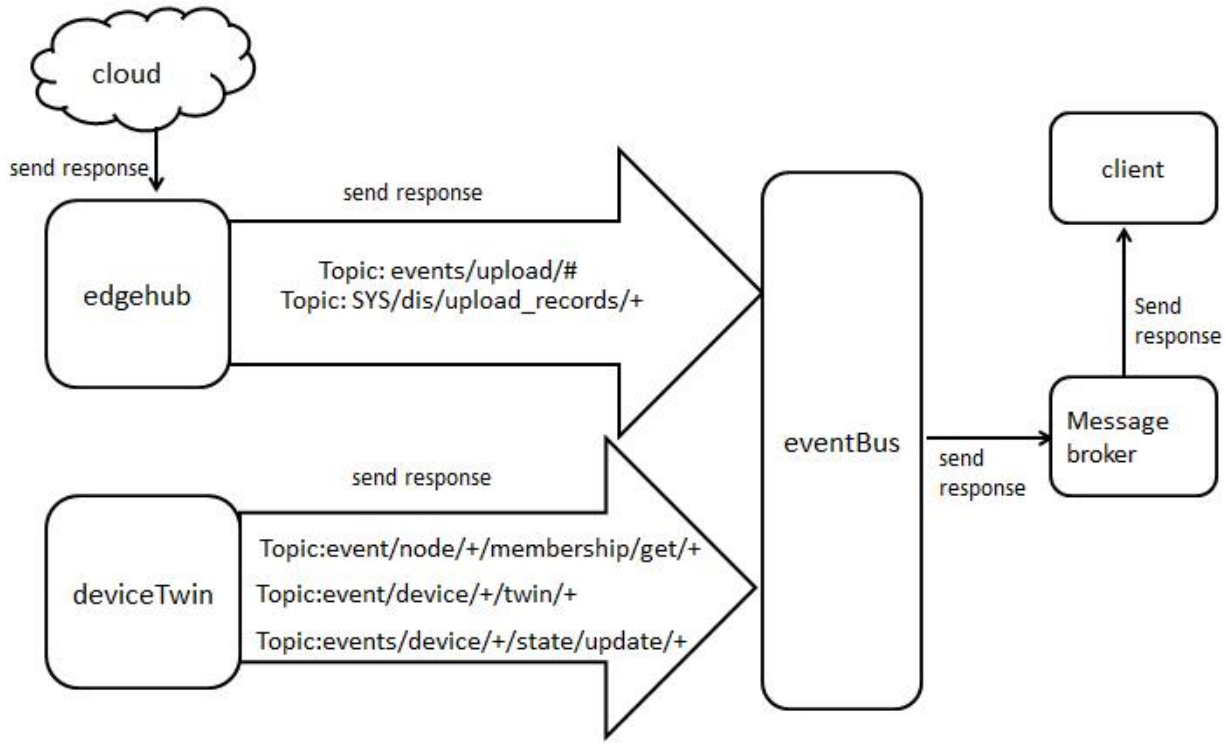
11.3.1 1. eventbus sends messages from external client



Eventbus sends messages from external client

eventbus sends messages from external client

11.3.2 2. eventbus sends response messages to external client



Eventbus sends response messages to external client

eventbus sends response messages to external client

CHAPTER 12

MetaManager

12.1 Overview

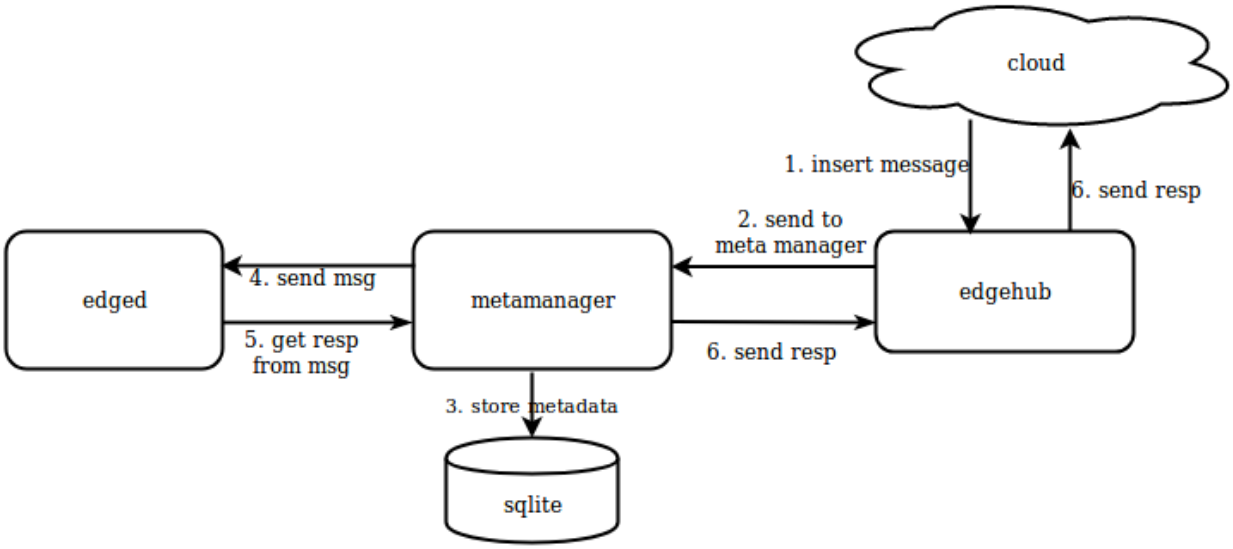
MetaManager is the message processor between edged and edgehub. It's also responsible for storing/retrieving metadata to/from a lightweight database(SQLite).

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

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

12.2 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

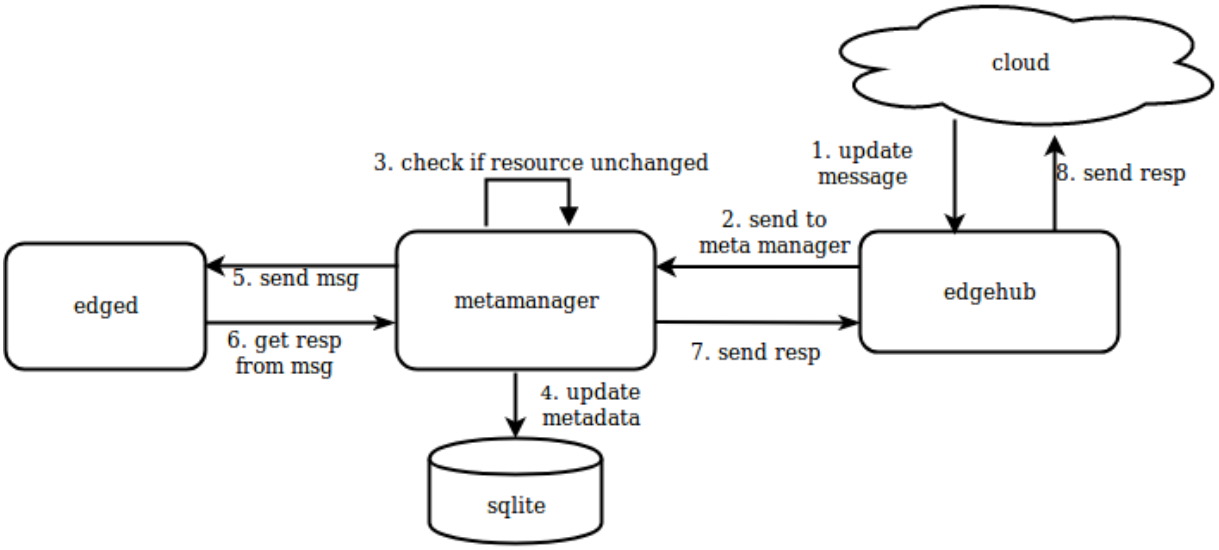
Insert Operation

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.

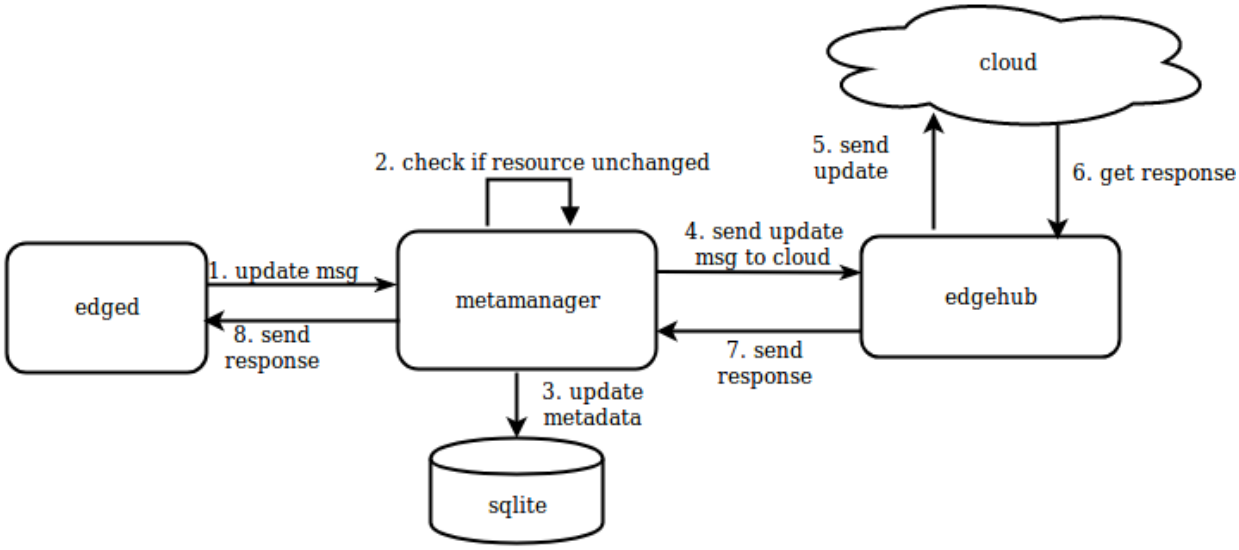
12.3 Update Operation

`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 From Cloud To Edge

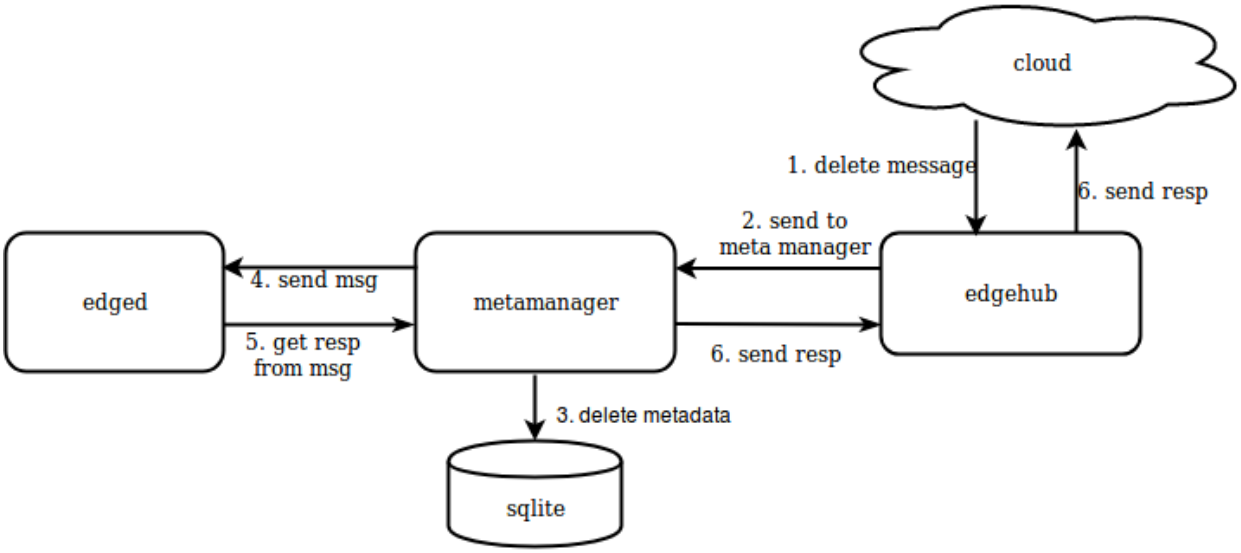


Update From Edge To Cloud

Update Operation

12.4 Delete Operation

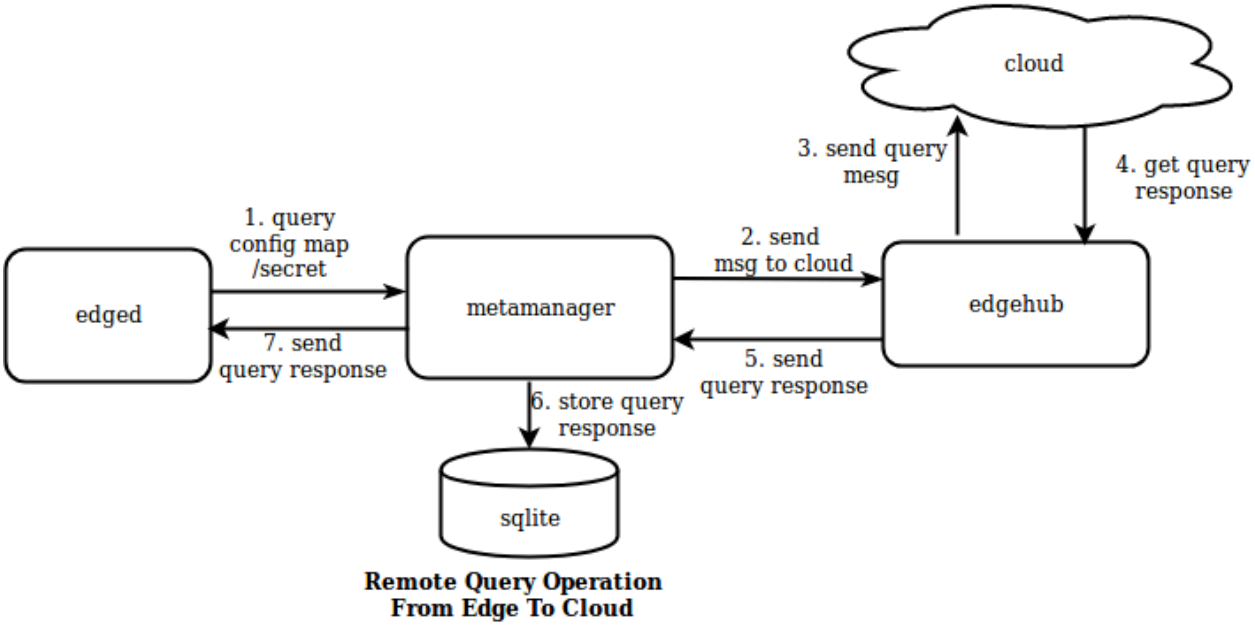
`Delete` operations are triggered when objects like pods are deleted from the cloud.



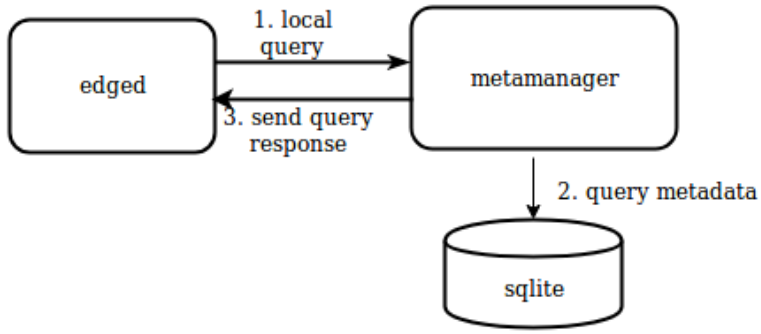
Delete Operation

12.5 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 '/'.



Remote Query Operation From Edge To Cloud



Local Query Operation At Edge

Query Operation

12.6 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.

12.7 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.

12.8 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).

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

CHAPTER 13

Edgehub

13.1 Overview

Edge hub is responsible for interacting with CloudHub component present in the cloud. It can connect to the CloudHub using either a web-socket connection or using QUIC protocol. 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

13.2 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)
```

13.3 Keep Alive

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

13.4 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.

13.5 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.

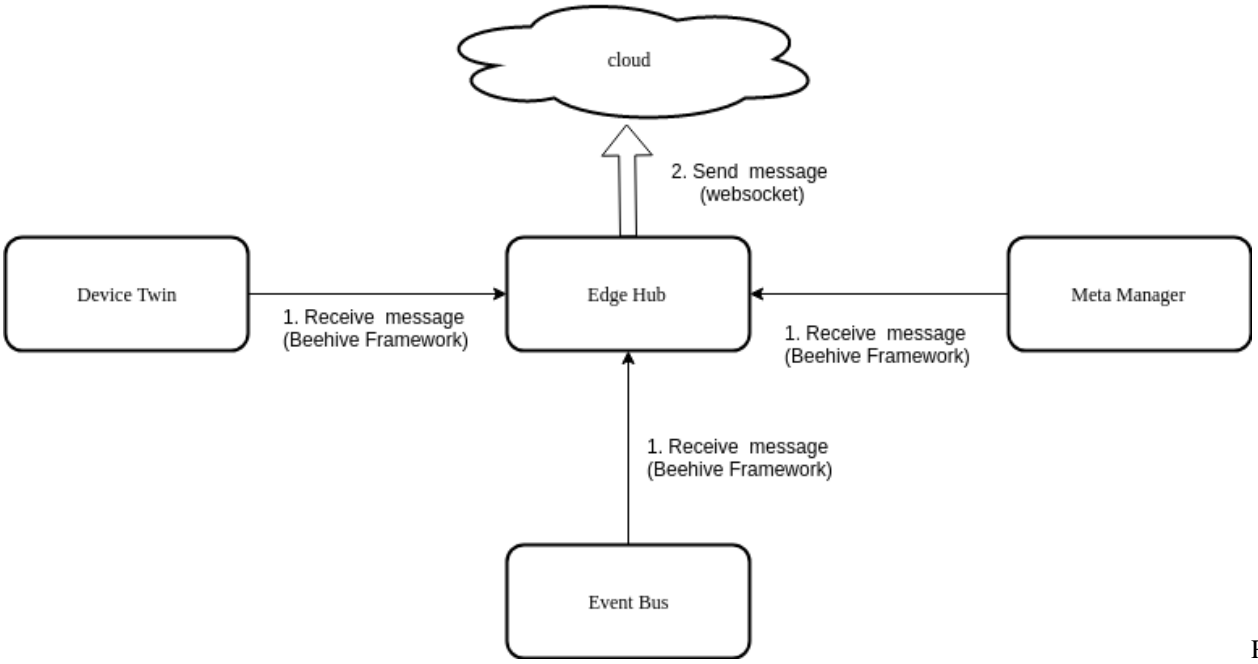
The major steps involved in this process are as follows :-

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.



Route

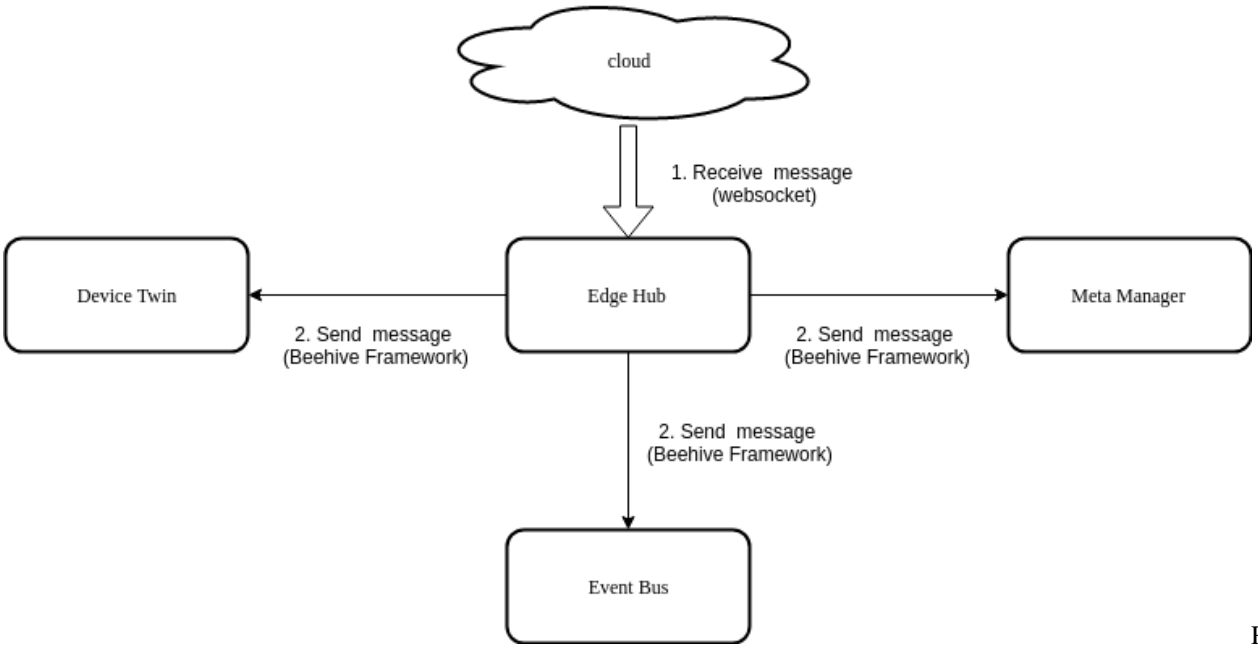
to Cloud

13.6 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



Route to Edge

13.7 Usage

EdgeHub can be configured to communicate in two ways as mentioned below:

- **Through websocket protocol**: Click here for details.
- **Through QUIC protocol**: Click here for details.

CHAPTER 14

DeviceTwin

14.1 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.

14.2 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

14.2.1 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.

14.2.2 Register and Start Sub Modules

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

14.2.3 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

14.2.4 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.

14.3 Modules

DeviceTwin consists of four modules, namely :-

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

14.3.1 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

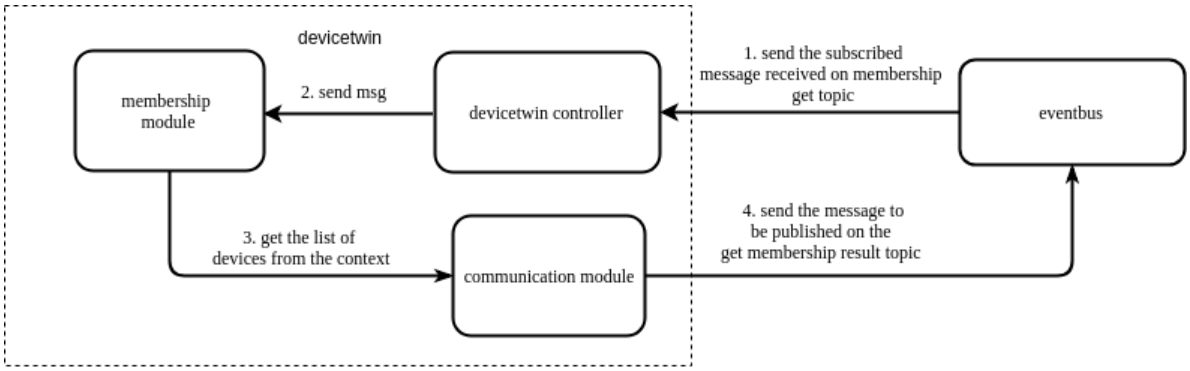
The following are the action callbacks which can be performed by the membership module :-

- 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).

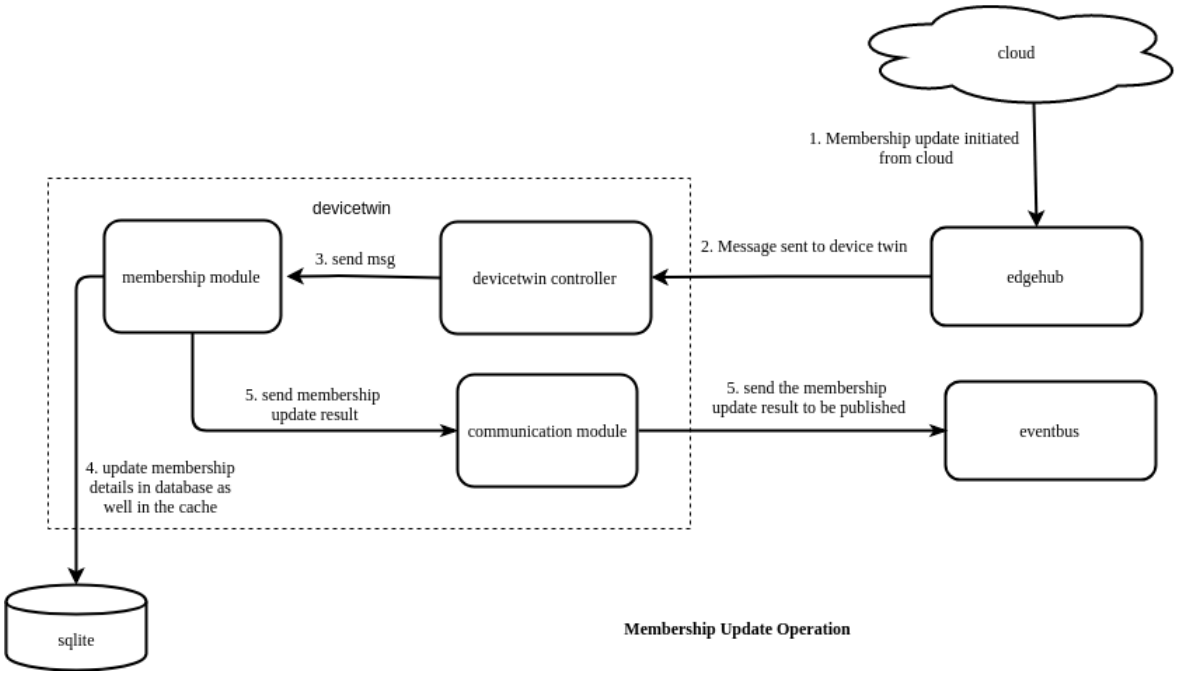


Memebership Get Operation

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.

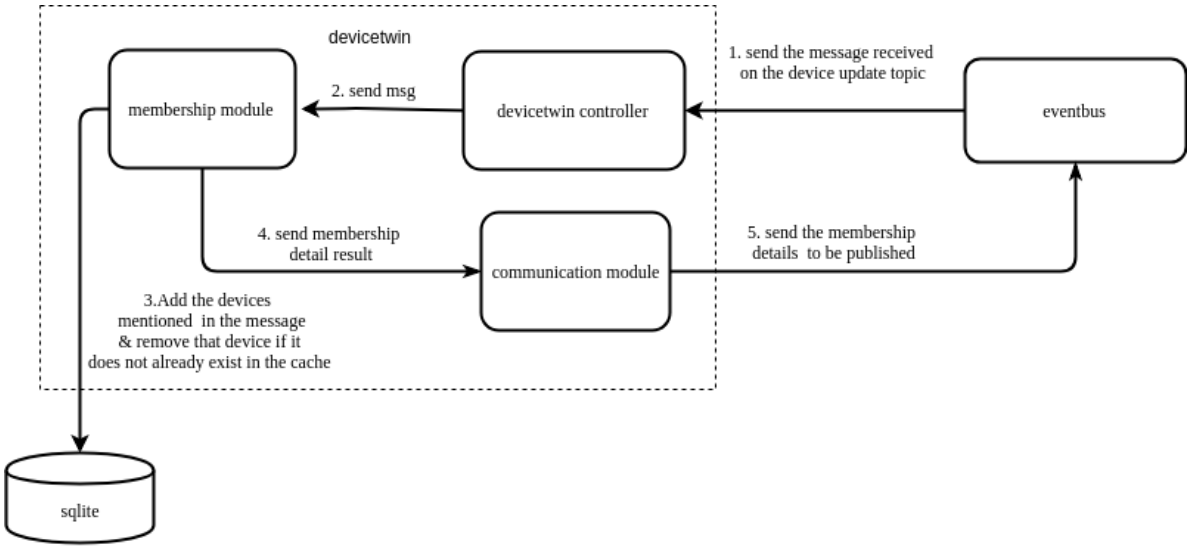
- The edgehub module receives the membership update message from the cloud and forwards the message to devicetwin controller which further forwards it to the membership module.
- The membership module adds devices that are newly added, removes devices that have been recently deleted and also updates 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.



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.



Membership Detail

14.3.2 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:-

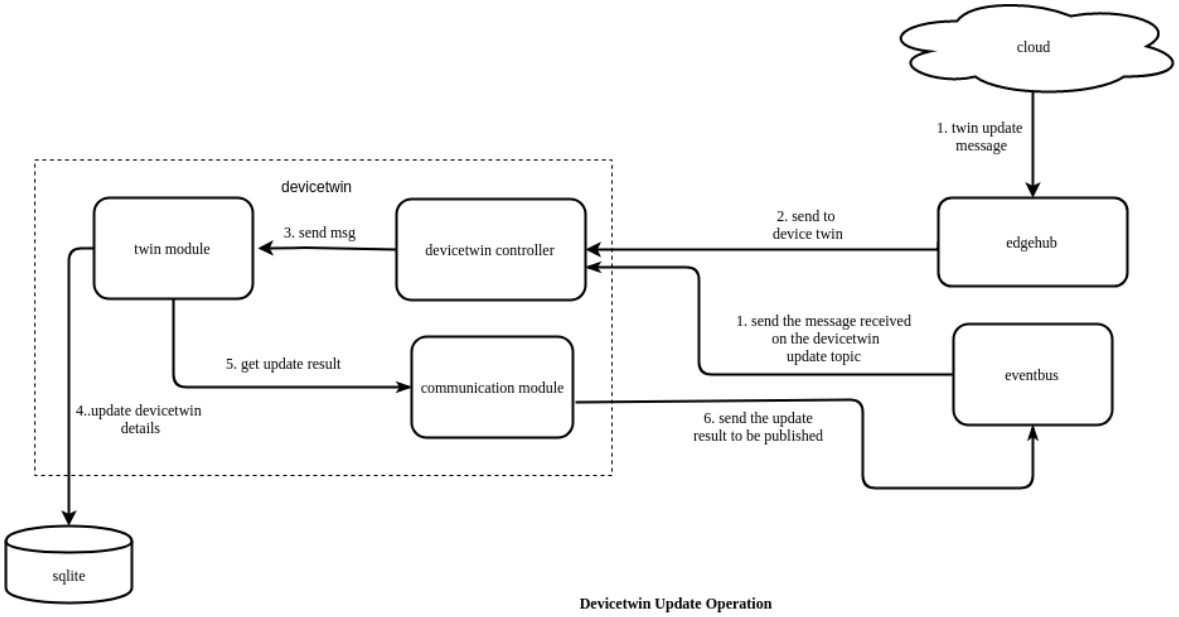
1. Initialize action callback map (which is a map of action(string) to the callback function that performs the requested action)
2. Receive the messages sent to twin module
3. For each message the action message is read and the corresponding function is called
4. Receive heartbeat from the heartbeat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



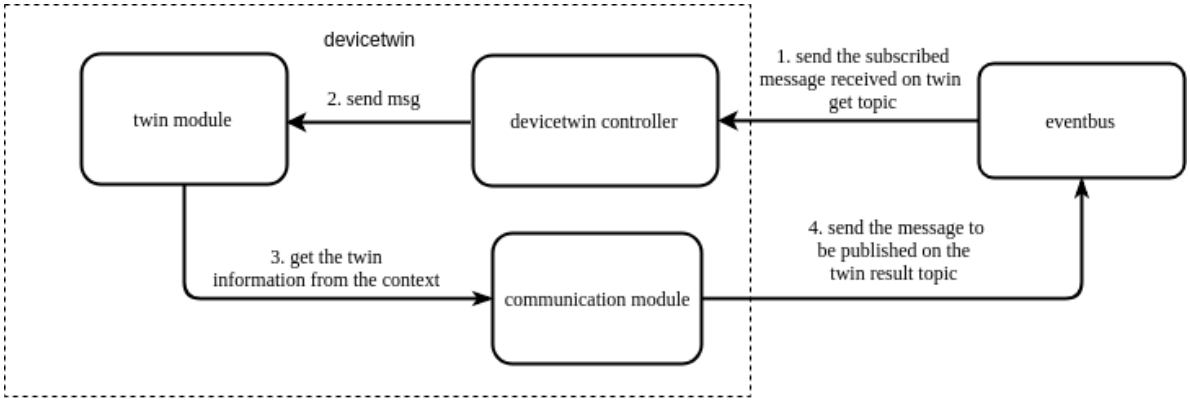
Devicetwin Update Operation

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 .

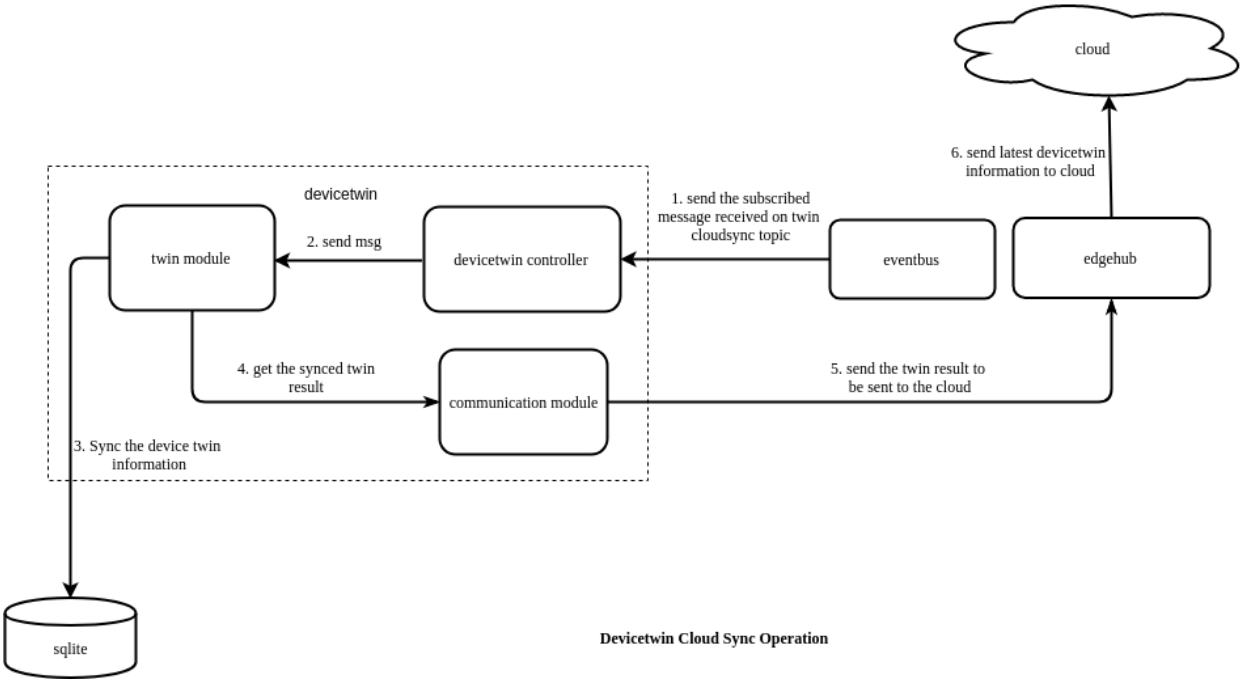


Devicetwin Get Operation

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.



Devicetwin Cloud Sync Operation

Sync to Cloud

14.3.3 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.

14.3.4 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 :-

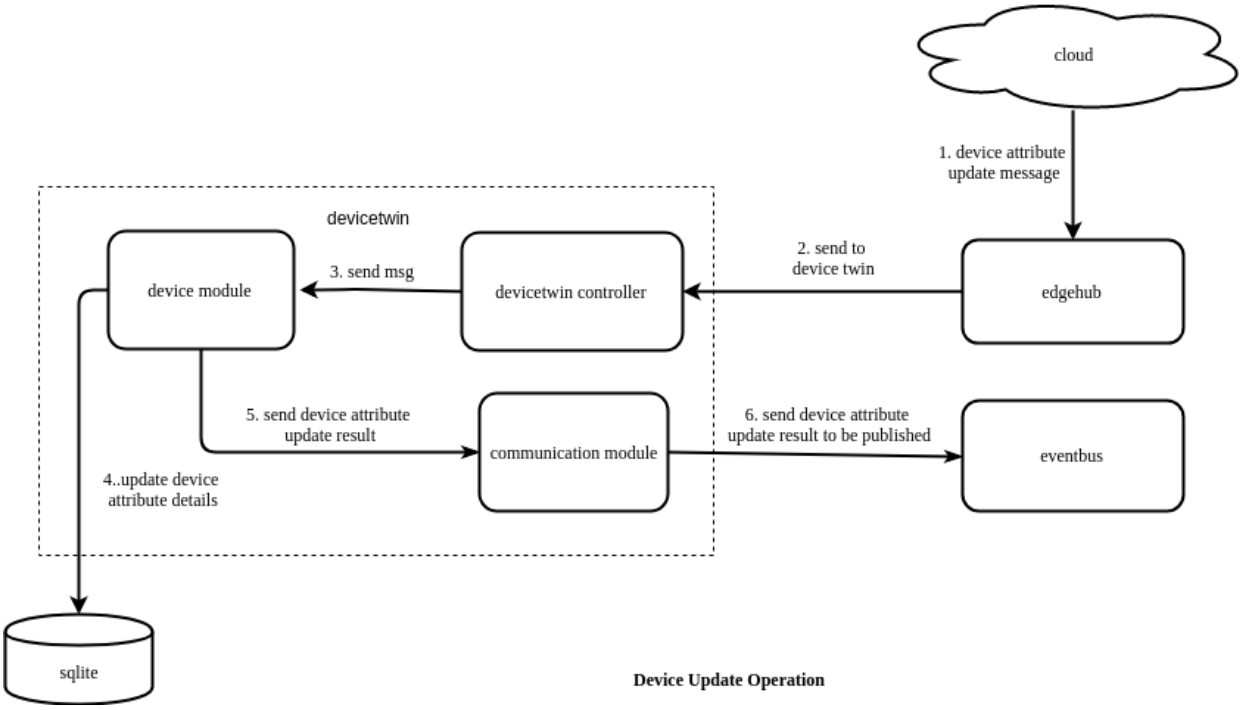
1. Initialize action callback map (which is a map of action(string) to the callback function that performs the requested action)
2. Receive the messages sent to device module
3. For each message the action message is read and the corresponding function is called
4. Receive heartbeat from the heartbeat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

- The device attribute updation is initiated from the cloud, which sends the update to edgehub.
- The edgehub component sends the message to the device twin controller which forwards the message to the device module.
- The device module updates the device attribute details into the database after which, the device module 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.

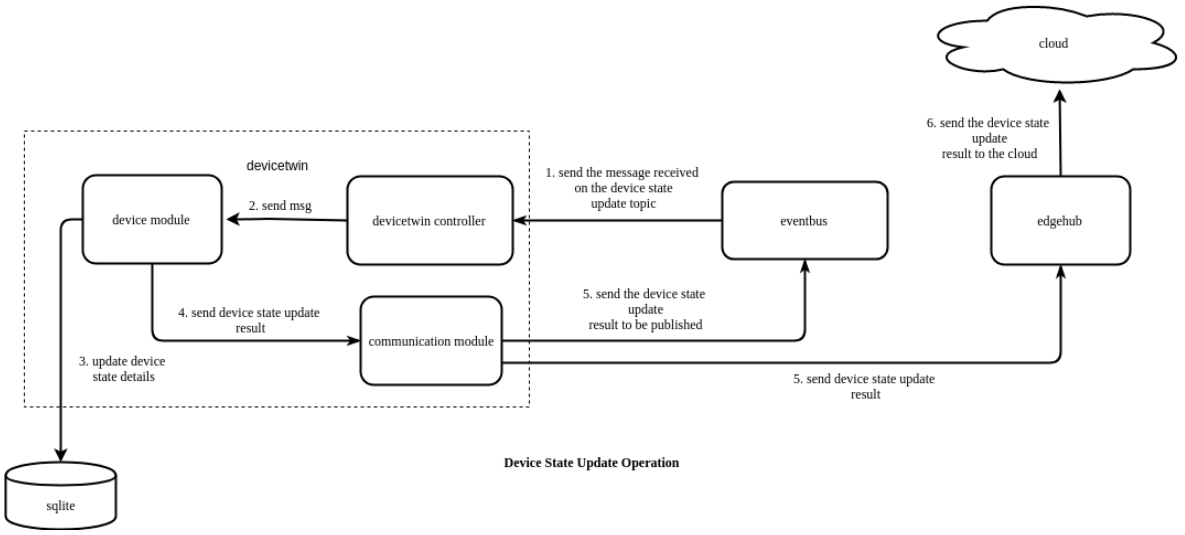


Device Update Operation

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.

- The device state updation is initiated by publishing a message on the specified topic which is being subscribed by the eventbus component.
- The eventbus component sends the message to the device twin controller which forwards the message to the device module.
- The device module updates the state of the device as well as the last online time of the device in the database.
- The device module then sends the result of the device state update to the eventbus component and edgehub component through the communicate module of devicetwin. The eventbus component further publishes the result on the specified topic, while the edgehub component sends the device status update to the cloud.



Device State Update

14.4 Tables

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

- Device Table
- Device Attribute Table
- Device Twin Table

14.4.1 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

14.4.2 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.

14.4.3 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 15

Edge Controller

15.1 Edge Controller Overview

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

15.2 Operations Performed By Edge Controller

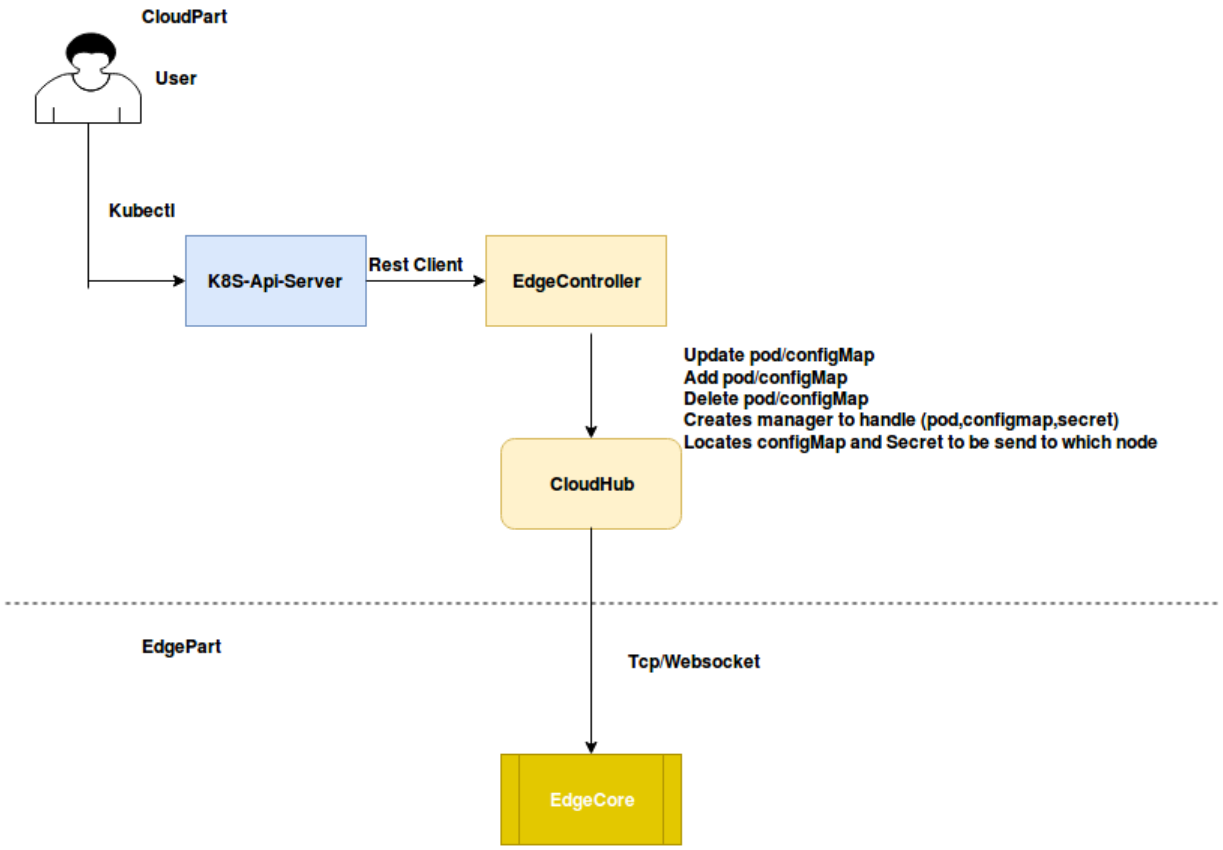
The following are the functions performed by Edge controller :-

- 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

15.3 Downstream Controller:

15.3.1 Sync add/update/delete event to edge

- Downstream controller: Watches K8S-Api-server and sends 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



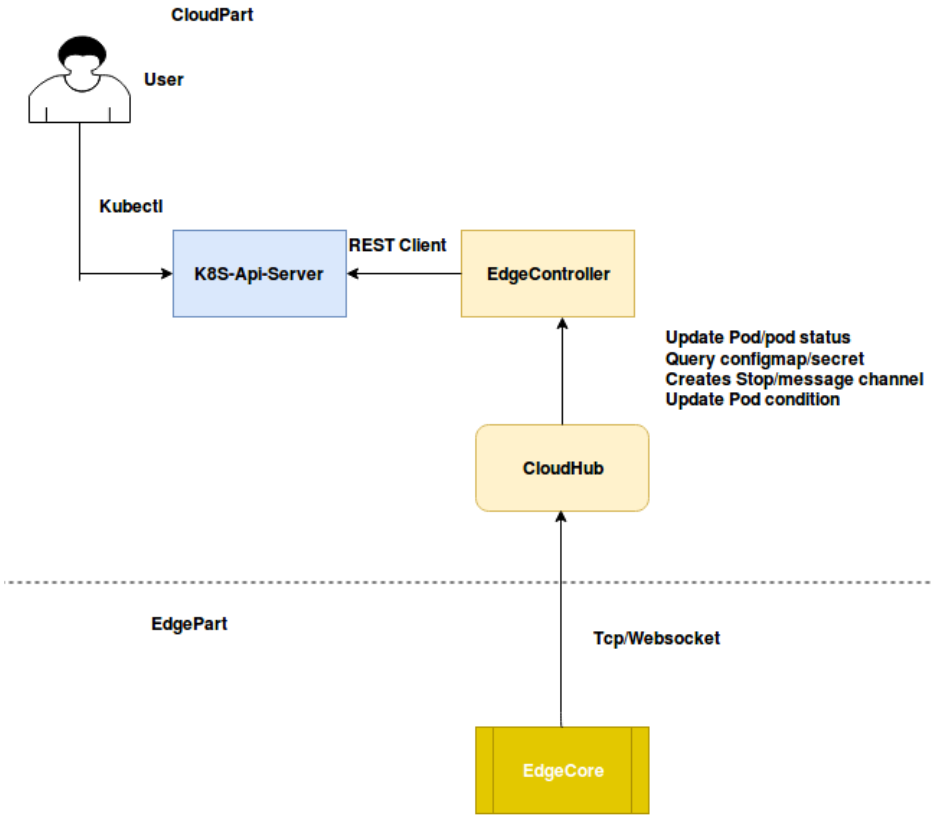
Downstream Controller

15.4 Upstream Controller:

15.4.1 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

15.5 Controller Manager:

15.5.1 Creates manager Interface and implements ConfigmapManager, LocationCache 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 16

CloudHub

16.1 CloudHub Overview

CloudHub is the mediator between EdgeController and the Edge side. It supports both web-socket based connection as well as a QUIC protocol access at the same time. The edgehub can choose one of the protocols to access to the cloudhub. CloudHub's function is to 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

16.1.1 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.

16.1.2 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

16.1.3 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

16.1.4 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

16.1.5 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.

16.2 Usage

The CloudHub can be configured in three ways as mentioned below :

- **Start the websocket server only**: Click here to see the details.
- **Start the quic server only**: Click here to see the details.
- **Start the websocket and quic server at the same time**: Click here to see the details

CHAPTER 17

Device Controller

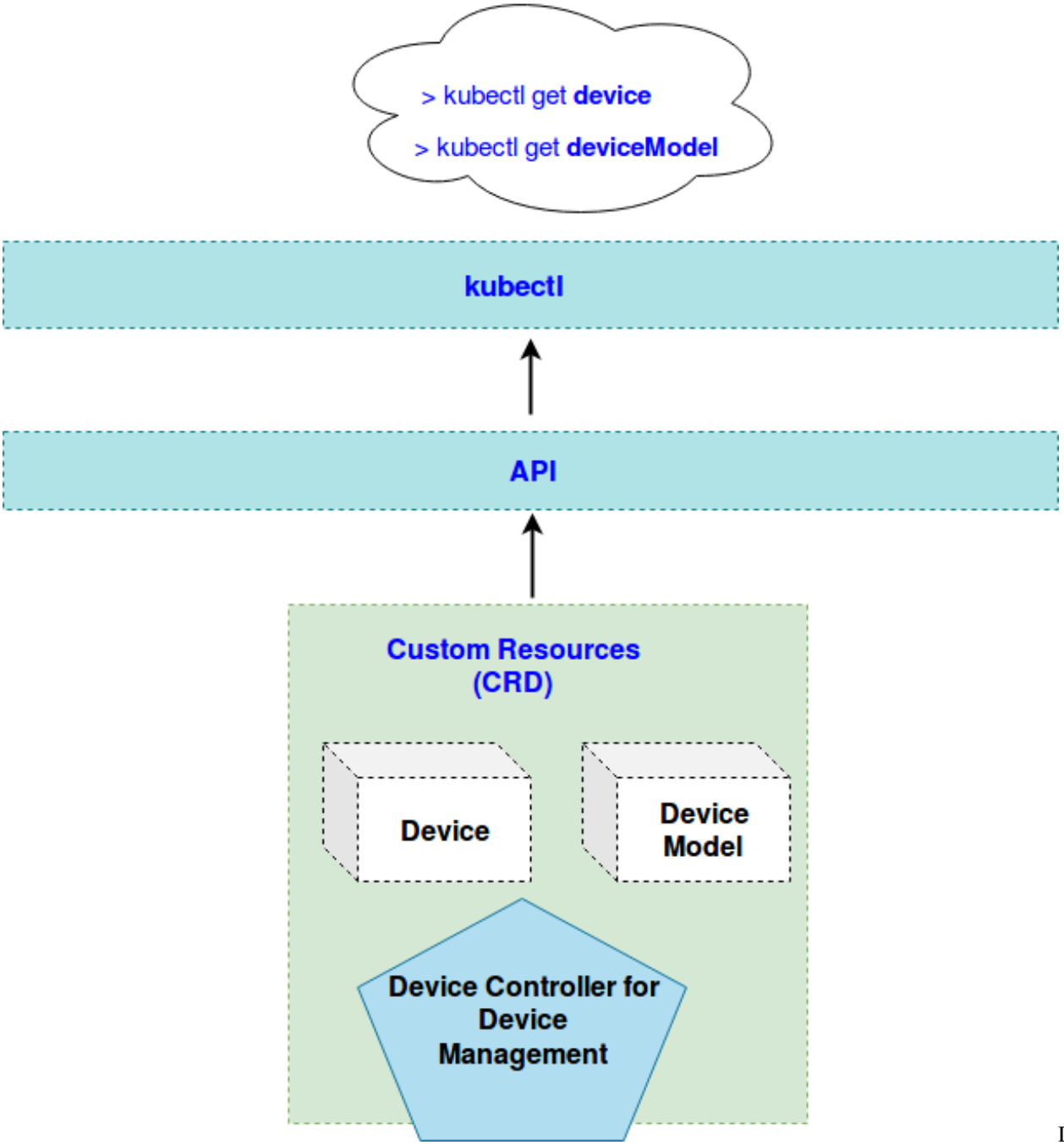
17.1 Device Controller Overview

The device controller is the cloud component of KubeEdge which is responsible for device management. Device management im KubeEdge is implemented by making use of Kubernetes Custom Resource Definitions (CRDs) to describe device metadata/status and device controller to synchronize these device updates between edge and cloud. The device controller starts two separate goroutines called `upstream controller` and `downstream controller`. These are not separate controllers as such but named here for clarity.

The device controller makes use of device model and device instance to implement device management :

- **Device Model**: A `device model` describes the device properties exposed by the device and property visitors to access these properties. A device model is like a reusable template using which many devices can be created and managed. Details on device model definition can be found here.
- **Device Instance**: A `device` instance represents an actual device object. It is like an instantiation of the `device model` and references properties defined in the model. The device spec is static while the device status contains dynamically changing data like the desired state of a device property and the state reported by the device. Details on device instance definition can be found here.

Note: Sample device model and device instance for a few protocols can be found at $GOPATH/src/github.com/kubeedge/kubeedge/build/crd-samples/devices



Device Model

17.2 Operations Performed By Device Controller

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

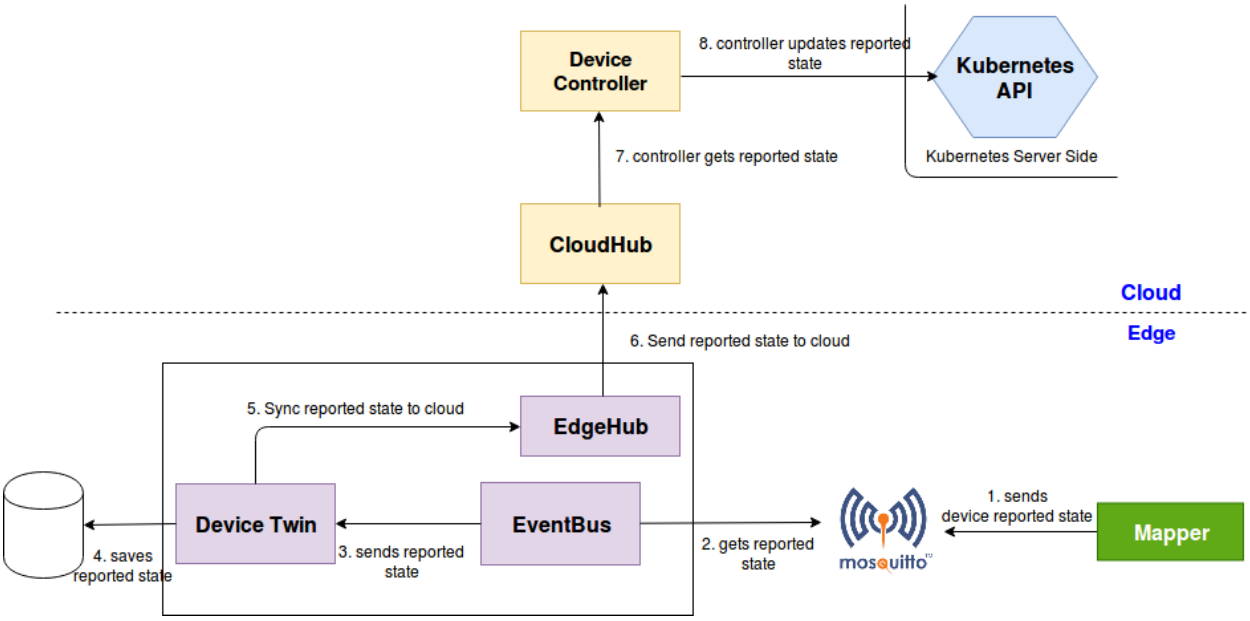
- **Downstream Controller**: Synchronize the device updates from the cloud to the edge node, by watching on K8S API server
- **Upstream Controller**: Synchronize the device updates from the edge node to the cloud using device twin

component

17.3 Upstream Controller:

The upstream controller watches for updates from the edge node and applies these updates against the API server in the cloud. Updates are categorized below along with the possible actions that the upstream controller can take:

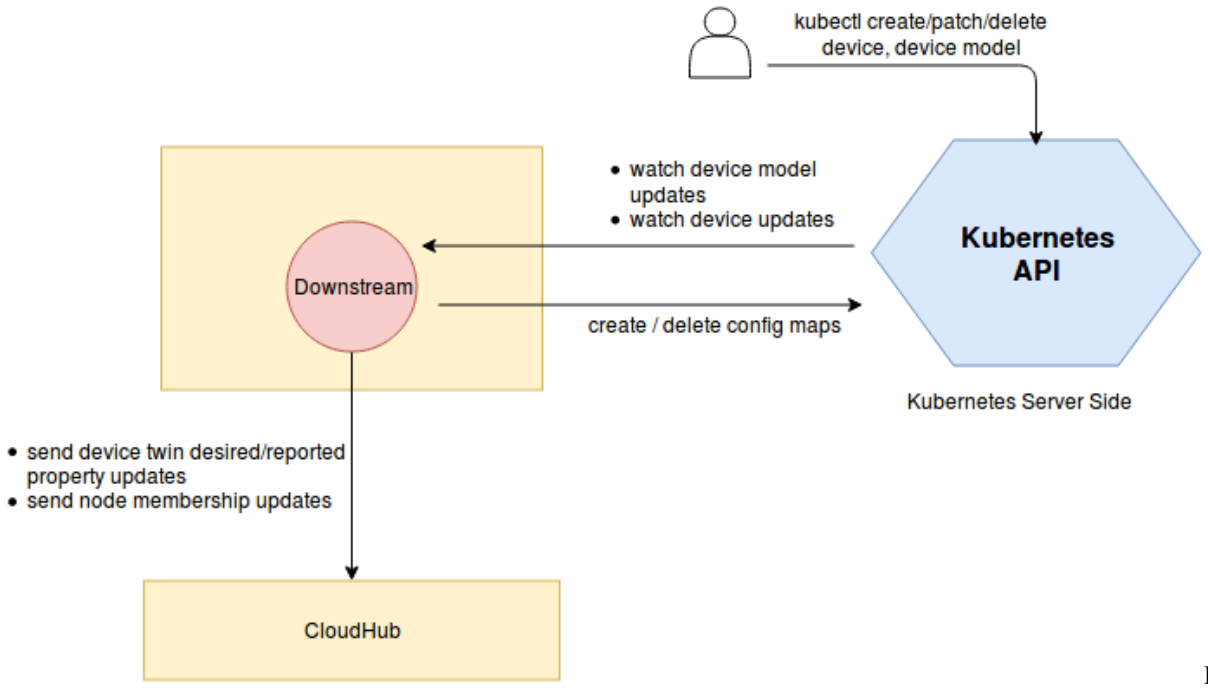
The mapper watches devices for updates and reports them to the event bus via the MQTT broker. The event bus sends the reported state of the device to the device twin which stores it locally and then syncs the updates to the cloud. The device controller watches for device updates from the edge (via the cloudhub) and updates the reported state in the cloud.



Device Updates Edge To Cloud

17.4 Downstream Controller:

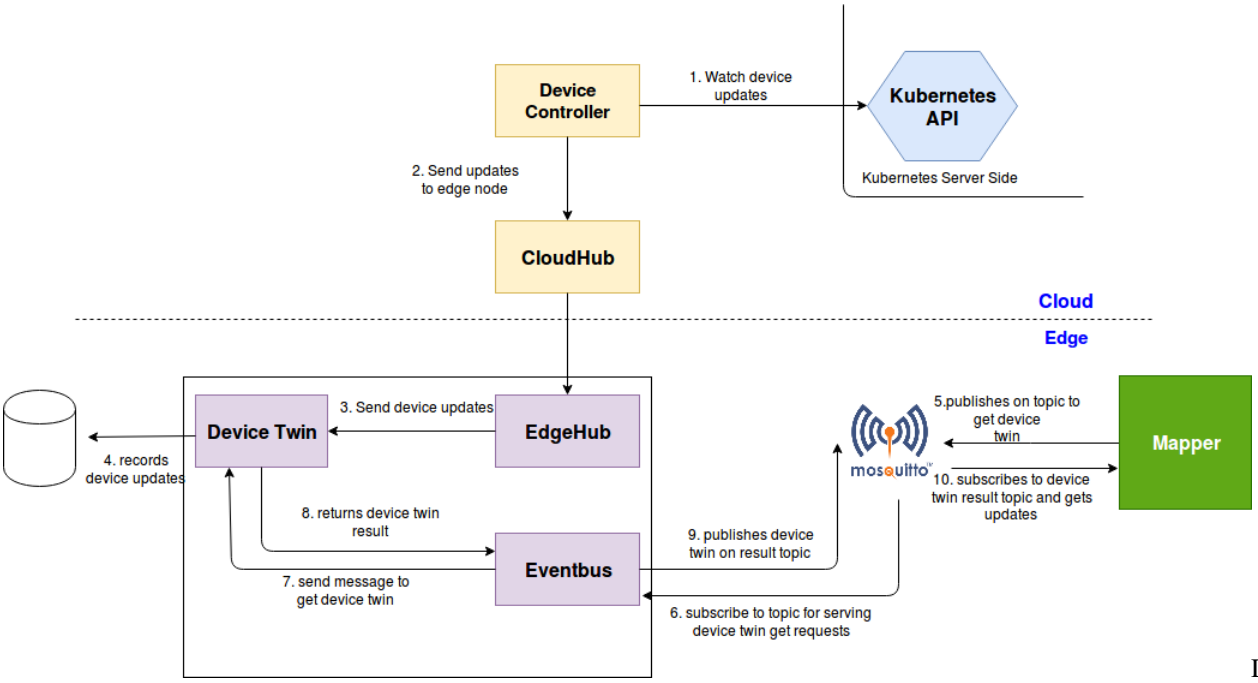
The downstream controller watches for device updates against the K8S API server. Updates are categorized below along with the possible actions that the downstream controller can take:



Device Downstream Controller

The idea behind using config map to store device properties and visitors is that these metadata are only required by the mapper applications running on the edge node in order to connect to the device and collect data. Mappers if run as containers can load these properties as config maps . Any additions , deletions or updates to properties , visitors etc in the cloud are watched upon by the downstream controller and config maps are updated in etcd. If the mapper wants to discover what properties a device supports, it can get the model information from the device instance. Also, it can get the protocol information to connect to the device from the device instace. Once it has access to the device model, it can get the properties supported by the device. In order to access the property, the mapper needs to get the corresponding visitor information. This can be retrieved from the propertyVisitors list. Finally, using the visitorConfig, the mapper can read/write the data associated with the property.

17.4.1 Syncing Desired Device Twin Property Update From Cloud To Edge



Device Updates Cloud To Edge The device controller watches device updates in the cloud and relays them to the edge node. These updates are stored locally by the device twin. The mapper gets these updates via the MQTT broker and operates on the device based on the updates.

CHAPTER 18

EdgeSite: Standalone Cluster at edge

18.1 Abstract

In Edge computing, there are scenarios where customers would like to have a whole cluster installed at edge location. As a result, admins/users can leverage the local control plane to implement management functionalities and take advantages of all edge computing's benefits.

EdgeSite helps running lightweight clusters at edge.

18.2 Motivation

There are scenarios user need to run a standalone Kubernetes cluster at edge to get full control and improve the offline scheduling capability. There are two scenarios user need to do that:

- The edge cluster is in CDN instead of the user's site

  The CDN sites usually be large around the world and the network connectivity and quality cannot be guaranteed. Another factor is that the application deployed in CDN edge do not need to interact with center usually. For those deploy edge cluster in CDN resources, they need to make sure the cluster is workable without the connection with central cloud not only for the deployed applicatons but also the schedule capabilities. So that the CDN edge is manageable regardless the connection to one center.

- User need to deploy an edge environment with limited resources and offline running for most of the time

  In some IOT scenarios, user need to deploy a full control edge environment and running offline.

For these use cases, a standalone, full controlled, light weight Edge cluster is required. By integrating KubeEdge and standard Kubernetes, this EdgeSite enables customers to run an efficient kubernetes cluster for Edge/IOT computing.

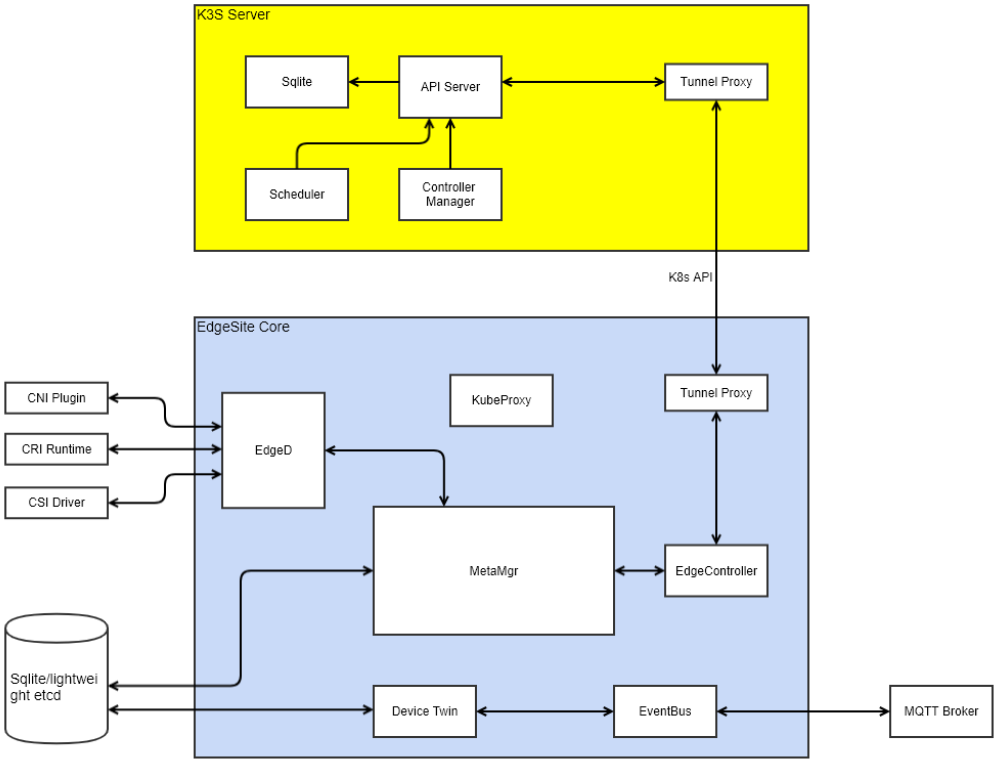
18.3 Assumptions

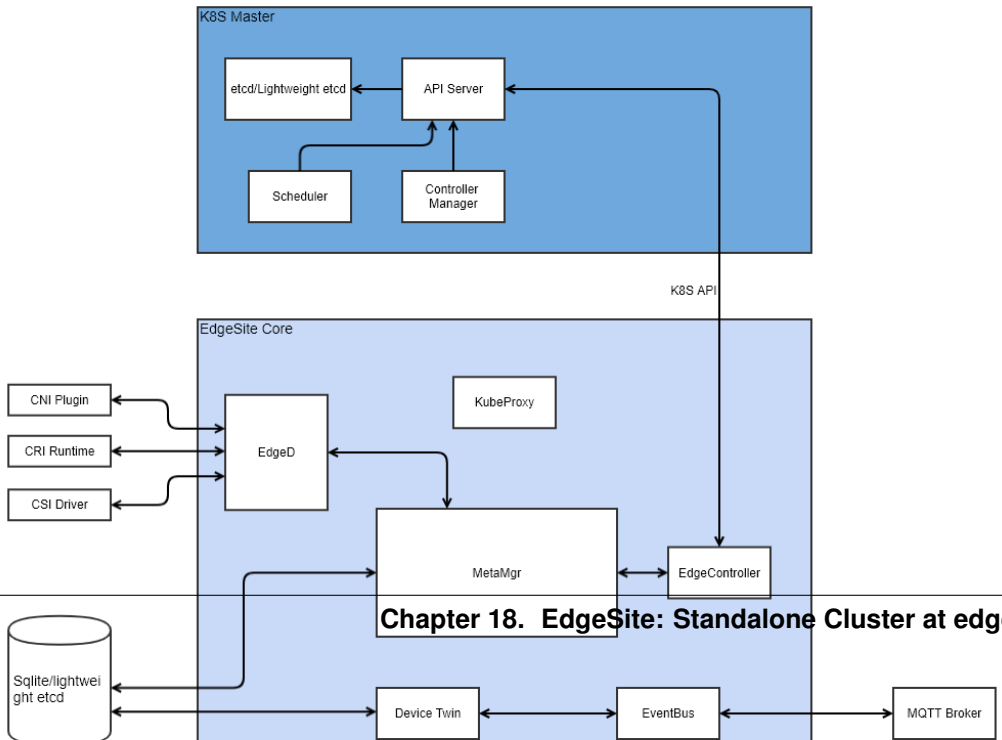
Here we assume a cluster is deployed at edge location including the management control plane. For the management control plane to manage some scale of edge worker nodes, the hosting master node needs to have sufficient resources.

The assumptions are

1. EdgeSite cluster master node is of no less than 2 CPUs and no less than 1GB memory
2. If high availability is required, 2-3 master nodes are needed at different edge locations
3. The same Kubernetes security (authN and authZ) mechanisms are used to ensure the secure handshake between master and worker nodes
4. The same K8s HA mechanism is to be used to enable HA

18.4 Architecture Design





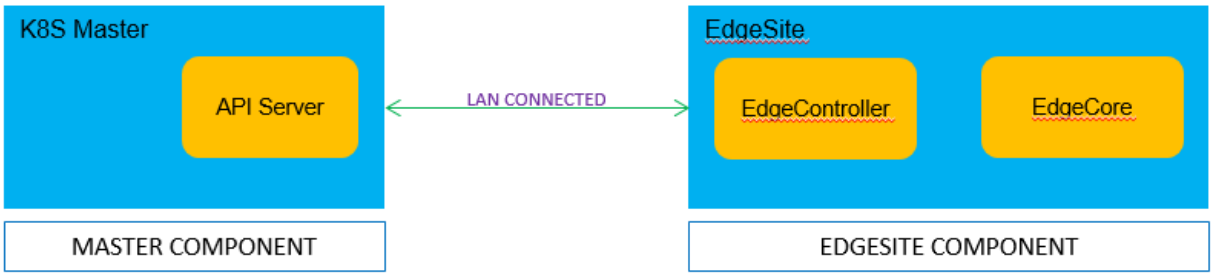
18.5 Advantages

With the integration, the following can be enabled

1. Full control of Kubernetes cluster at edge
2. Light weight control plane and agent
3. Edge worker node autonomy in case of network disconnection/reconnection
4. All benefits of edge computing including latency, data locality, etc.

18.6 Getting Started

18.6.1 Setup



EdgeSite Setup

18.6.2 Steps for K8S (API server) Cluster

- Install docker
- Install kubeadm/kubectl
- Creating cluster with kubeadm
- After initializing Kubernetes master, we need to expose insecure port 8080 for edgecontroller/kubectl to work with http connection to Kubernetes apiserver. Please follow below steps to enable http port in Kubernetes 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
```

- (**Optional**)KubeEdge also supports https connection to Kubernetes apiserver. Follow the steps in Kubernetes Documentation to create the kubeconfig file.

  Enter the path to kubeconfig file in controller.yaml

```
controller:
  kube:
    ...
    kubeconfig: "path_to_kubeconfig_file" #Enter path to kubeconfig file to␣
↪enable https connection to k8s apiserver
```

18.6.3 Steps for EdgeSite

Getting EdgeSite Binary

Using Source code

- Clone KubeEdge (EdgeSite) code

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

- Build EdgeSite

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/edgesite
make
```

Download Release packages

TBA

Configuring EdgeSite

Modify edgeSite.yaml configuration file, with the IP address of K8S API server

- Configure K8S (API Server)

  Replace `localhost` at `controller.kube.master` with the IP address

```
controller:
  kube:
    master: http://localhost:8080
    ...
```

- Add EdgeSite (Worker) Node ID/name

  Replace `fb4ebb70-2783-42b8-b3ef-63e2fd6d242e` with an unique edge id/name in below fields :

  - `controller.kube.node-id`
  - `controller.edged.hostname-override`

```
controller:
  kube:
    ...
    node-id: fb4ebb70-2783-42b8-b3ef-63e2fd6d242e
    node-name: fb4ebb70-2783-42b8-b3ef-63e2fd6d242e
    ...
  edged:
    ...
    hostname-override: fb4ebb70-2783-42b8-b3ef-63e2fd6d242e
    ...
```

- Configure MQTT (**Optional**)

  The Edge part of KubeEdge uses MQTT for communication between deviceTwin and devices. KubeEdge supports 3 MQTT modes:

1. internalMqttMode: internal mqtt broker is enabled. (Default)
2. bothMqttMode: internal as well as external broker are enabled.
3. externalMqttMode: only external broker is enabled.

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

```
mqtt:
  ...
  mode: 0 # 0: internal mqtt broker enable only. 1: internal and external mqtt␣
↪broker enable. 2: external mqtt broker enable only.
  ...
```

To use KubeEdge in double mqtt or external mode, you need to make sure that mosquitto or emqx edge is installed on the edge node as an MQTT Broker.

Run EdgeSite

```
# run edgesite
# `conf/` should be in the same directory as the cloned KubeEdge repository
# verify the configurations before running edgesite
./edgesite
# or
nohup ./edgesite > edgesite.log 2>&1 &
```

Note: Please run edgesite using the users who have root permission.

18.6.4 Deploy EdgeSite (Worker) Node to K8S Cluster

We have provided a sample node.json to add a node in kubernetes. Please make sure edgesite (worker) node is added to k8s api-server. Run below steps:

- Modify node.json

  Replace `fb4ebb70-2783-42b8-b3ef-63e2fd6d242e` in node.json file, to the id/name of the edgesite node. ID/Name should be same as used before while updating `edgesite.yaml`

```
{
  "metadata": {
    "name": "fb4ebb70-2783-42b8-b3ef-63e2fd6d242e",
  }
}
```

- Add node in K8S API server

  In the console execute the below command

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

- Check node status

  Below command to check the edgesite node status.

```
kubectl get nodes

NAME          STATUS     ROLES    AGE      VERSION
testing123    Ready      <none>   6s       0.3.0-beta.0
```

Observe the edgesite node is in `Ready` state

18.6.5 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 edgesite 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.

Then you can use below command to check if the application is normally running.

```
  kubectl get pods
```

CHAPTER 19

Bluetooth Mapper

19.1 Introduction

Mapper is an application that is used to connect and control devices. This is an implementation of mapper for bluetooth protocol. The aim is to create an application through which users can easily operate devices using bluetooth protocol for communication to the KubeEdge platform. The user is required to provide the mapper with the information required to control their device through the configuration file. These can be changed at runtime by providing the input through the MQTT broker.

19.2 Running the mapper

1. Please ensure that bluetooth service of your device is ON
2. Set 'bluetooth=true' label for the node (This label is a prerequisite for the scheduler to schedule bluetooth_mapper pod on the node)

```
kubectl label nodes <name-of-node> bluetooth=true
```

3. Build and deploy the mapper by following the steps given below.

19.2.1 Building the bluetooth mapper

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/device/bluetooth_mapper
make bluetooth_mapper_image
docker tag bluetooth_mapper:v1.0 <your_dockerhub_username>/bluetooth_mapper:v1.0
docker push <your_dockerhub_username>/bluetooth_mapper:v1.0

Note: Before trying to push the docker image to the remote repository please ensure␣
↪that you have signed into docker from your node, if not please type the followig␣
↪command to sign in
```

(continues on next page)

(continued from previous page)

```
docker login
# Please enter your username and password when prompted
```

19.2.2 Deploying bluetooth mapper application

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/device/bluetooth_mapper

# Please enter the following details in the deployment.yaml :-
#    1. Replace <edge_node_name> with the name of your edge node at spec.template.
↪spec.voluems.configMap.name
#    2. Replace <your_dockerhub_username> with your dockerhub username at spec.
↪template.spec.containers.image

kubectl create -f deployment.yaml
```

19.3 Modules

The bluetooth mapper consists of the following five major modules :-

1. Action Manager
2. Scheduler
3. Watcher
4. Controller
5. Data Converter

19.3.1 Action Manager

A bluetooth device can be controlled by setting a specific value in physical register(s) of a device and readings can be acquired by getting the value from specific register(s). We can define an Action as a group of read/write operations on a device. A device may support multiple such actions. The registers are identified by characteristic values which are exposed by the device through entities called characteristic-uuids. Each of these actions should be supplied through config-file to action manager or at runtime through MQTT. The values specified initially through the configuration file can be modified at runtime through MQTT. Given below is a guide to provide input to action manager through the configuration file.

```
action-manager:
   actions:          # Multiple actions can be added
     - name: <name of the action>
       perform-immediately: <true/false>
       device-property-name: <property-name defined in the device model>
     - .......
       .......
```

1. Multiple actions can be added in the action manager module. Each of these actions can either be executed by the action manager of invoked by other modules of the mapper like scheduler and watcher.
2. Name of each action should be unique, it is using this name that the other modules like the scheduler or watcher can invoke which action to perform.

3. Perform-immediately field of the action manager tells the action manager whether it is supposed to perform the action immediately or not, if it set to true then the action manger will perform the event once.
4. Each action is associated with a device-property-name, which is the property-name defined in the device CRD, which in turn contains the implementation details required by the action.

19.3.2 Scheduler

Scheduler is a component which can perform an action or a set of actions at regular intervals of time. They will make use of the actions previously defined in the action manager module, it has to be ensured that before the execution of the schedule the action should be defined, otherwise it would lead to an error. The schedule can be configured to run for a specified number of times or run infinitely. The scheduler is an optional module and need not be specified if not required by the user. The user can provide input to the scheduler through configuration file or through MQTT at runtime. The values specified initially by the user through the configuration file can be modified at runtime through MQTT. Given below is a guide to provide input to scheduler through the configuration file.

```
      scheduler:
        schedules:
          - name: <name of schedule>
            interval: <time in milliseconds>
            occurrence-limit: <number of times to be executed>            # if it is␣
↪0, then the event will execute infinitely
            actions:
              - <action name>
              - <action name>
          - ......
            ......
```

1. Multiple schedules can be defined by the user by providing an array as input though the configuration file.
2. Name specifies the name of the schedule to be executed, each schedule must have a unique name as it is used as a method of identification by the scheduler.
3. Interval refers to the time interval at which the schedule is meant to be repeated. The user is expected to provide the input in milliseconds.
4. Occurrence-limit refers to the number of times the action(s) is supposed to occur. If the user wants the event to run infinitely then it can be set to 0 or the field can be skipped.
5. Actions refer to the action names which are supposed to be executed in the schedule. The actions will be defined in the same order in which they are mentioned here.
6. The user is expected to provide the names of the actions to be performed in the schedule, in the same order that they are to be executed.

19.3.3 Watcher

The following are the main responsibilities of the watcher component: a) To scan for bluetooth devices and connect to the correct device once it is Online/In-Range.

b) Keep a watch on the expected state of the twin-attributes of the device and perform the action(s) to make actual state equal to expected.

c) To report the actual state of twin attributes back to the cloud.

The watcher is an optional component and need not be defined or used by the user if not necessary. The input to the watcher can be provided through the configuration file or through mqtt at runtime. The values that are defined through

the configuration file can be changed at runtime through MQTT. Given below is a guide to provide input to the watcher through the configuration file.

```
    watcher:
        device-twin-attributes :
        - device-property-name: <name of attribute>
            - <action name>
            - <action name>
        - ......
          ......
```

1. Device-property-name refers to the device twin attribute name that was given when creating the device. It is using this name that the watcher watches for any change in expected state.
2. Actions refers to a list of action names, these are the names of the actions using which we can convert the actual state to the expected state.
3. The names of the actions being provided must have been defined using the action manager before the mapper begins execution. Also the action names should be mentioned in the same order in which they have to be executed.

19.3.4 Controller

The controller module is responsible for exposing MQTT APIs to perform CRUD operations on the watcher, scheduler and action manager. The controller is also responsible for starting the other modules like action manager, watcher and scheduler. The controller first connects the MQTT client to the broker (using the mqtt configurations, specified in the configuration file), it then initiates the watcher which will connect to the device (based on the configurations provided in the configuration file) and the watcher runs parallelly, after this it starts the action manger which executes all the actions that have been enabled in it, after which the scheduler is started to run parallelly as well. Given below is a guide to provide input to the controller through the configuration file.

```
    mqtt:
      mode: 0       # 0 -internal mqtt broker  1 - external mqtt broker
      server: tcp://127.0.0.1:1883 # external mqtt broker url.
      internal-server: tcp://127.0.0.1:1884 # internal mqtt broker url.
    device-model-name: <device_model_name>
```

19.4 Usage

19.4.1 Configuration File

The user can give the configurations specific to the bluetooth device using configurations provided in the configuration file present at $GOPATH/src/github.com/kubeedge/kubeedge/device/bluetooth_mapper/configuration/config.yaml. The details provided in the configuration file are used by action-manager module, scheduler module, watcher module, the data-converter module and the controller.

Example: Given below is the instructions using which user can create their own configuration file, for their device.

```
    mqtt:
      mode: 0       # 0 -internal mqtt broker  1 - external mqtt broker
      server: tcp://127.0.0.1:1883 # external mqtt broker url.
      internal-server: tcp://127.0.0.1:1884 # internal mqtt broker url.
    device-model-name: <device_model_name>        #deviceID received while␣
↪registering device with the cloud
```

(continues on next page)

(continued from previous page)

```
    action-manager:
      actions:          # Multiple actions can be added
      - name: <name of the action>
        perform-immediately: <true/false>
        device-property-name: <property-name defined in the device model>
      - .......
        .......
    scheduler:
      schedules:
      - name: <name of schedule>
        interval: <time in milliseconds>
        occurrence-limit: <number of times to be executed>            # if it is 0,␣
↪then the event will execute infinitely
        actions:
        - <action name>
        - <action name>
        - ......
      - ......
    watcher:
      device-twin-attributes :
      - device-property-name: <name of attribute>
        actions:        # Multiple actions can be added
        - <action name>
        - <action name>
        - ......
      - ......
```

19.4.2 Runtime Configuration Modifications

The configuration of the mapper as well as triggering of the modules of the mapper can be done during runtime. The user can do this by publishing messages on the respective MQTT topics of each module. Please note that we have to use the same MQTT broker that is being used by the mapper i.e. if the mapper is using the internal MQTT broker then the messages have to be published on the internal MQTT broker and if the mapper is using the external MQTT broker then the messages have to be published on the external MQTT broker.

The following properties can be changed at runtime by publishing messages on MQTT topics of the MQTT broker:

- Watcher
- Action Manager
- Scheduler

Watcher

The user can add or update the watcher properties of the mapper at runtime. It will overwrite the existing watcher configurations (if exists)

Topic: $ke/device/bluetooth-mapper/< deviceID >/watcher/create

Message:

```
        {
         "device-twin-attributes": [
           {
```

(continues on next page)

(continued from previous page)

```
            "device-property-name": "IOControl",
            "actions": [                    # List of names of actions to be␣
↪performed (actions should have been defined before watching)
              "IOConfigurationInitialize",
              "IODataInitialize",
              "IOConfiguration",
              "IOData"
            ]
          }
        ]
      }
```

Action Manager

In the action manager module the user can perform two types of operations at runtime, i.e. : 1. The user can add or update the actions to be performed on the bluetooth device. 2. The user can delete the actions that were previously defined for the bluetooth device.

Action Add

The user can add a set of actions to be performed by the mapper. If an action with the same name as one of the actions in the list exists then it updates the action and if the action does not already exist then it is added to the existing set of actions.

Topic: $ke/device/bluetooth-mapper/< deviceID >/action-manager/create

Message:

```
  [
    {
      "name": "IRTemperatureConfiguration",          # name of action
      "perform-immediately": true,                   # whether the action is to␣
↪performed immediately or not
      "device-property-name": "temperature-enable"   #property-name defined in the␣
↪device model
    },
    {
      "name": "IRTemperatureData",
      "perform-immediately": true,
      "device-property-name": "temperature"          #property-name defined in the␣
↪device model
    }
  ]
```

Action Delete

The users can delete a set of actions that were previously defined for the device. If the action mentioned in the list does not exist then it returns an error message.

Topic: $ke/device/bluetooth-mapper/< deviceID >/action-manager/delete

Message:

```
    [
      {
        "name": "IRTemperatureConfiguration"        #name of action to be deleted
      },
      {
        "name": "IRTemperatureData"
      },
      {
        "name": "IOConfigurationInitialize"
      },
      {
        "name": "IOConfiguration"
      }
    ]
```

Scheduler

In the scheduler module the user can perform two types of operations at runtime, i.e. : 1. The user can add or update the schedules to be performed on the bluetooth device. 2. The user can delete the schedules that were previously defined for the bluetooth device.

Schedule Add

The user can add a set of schedules to be performed by the mapper. If a schedule with the same name as one of the schedules in the list exists then it updates the schedule and if the action does not already exist then it is added to the existing set of schedules.

Topic: $ke/device/bluetooth-mapper/< deviceID >/scheduler/create

Message:

```
[
  {
    "name": "temperature",            # name of schedule
    "interval": 3000,           # frequency of the actions to be executed (in␣
↪milliseconds)
    "occurrence-limit": 25,         # Maximum number of times the event is to be␣
↪executed, if not given then it runs infinitely
    "actions": [                          # List of names of actions to be performed␣
↪(actions should have been defined before execution of schedule)
      "IRTemperatureConfiguration",
      "IRTemperatureData"
    ]
  }
]
```

Schedule Delete

The users can delete a set of schedules that were previously defined for the device. If the schedule mentioned in the list does not exist then it returns an error message.

Topic: $ke/device/bluetooth-mapper/< deviceID >/scheduler/delete

Message:

```
[
  {
    "name": "temperature"                    #name of schedule to be deleted
  }
]
```

CHAPTER 20

Modbus Mapper

20.1 Introduction

Mapper is an application that is used to connect and control devices. This is an implementation of mapper for Modbus protocol. The aim is to create an application through which users can easily operate devices using ModbusTCP/ModbusRTU protocol for communication to the KubeEdge platform. The user is required to provide the mapper with the information required to control their device through the dpl configuration file. These can be changed at runtime by updating configmap.

20.2 Running the mapper

1. Please ensure that Modbus device is connected to your edge node
2. Set 'modbus=true' label for the node (This label is a prerequisite for the scheduler to schedule modbus_mapper pod on the node)

```
kubectl label nodes <name-of-node> modbus=true
```

3. Build and deploy the mapper by following the steps given below.

20.2.1 Building the modbus mapper

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/device/modbus_mapper
make # or `make modbus_mapper`
docker tag modbus_mapper:v1.0 <your_dockerhub_username>/modbus_mapper:v1.0
docker push <your_dockerhub_username>/modbus_mapper:v1.0

Note: Before trying to push the docker image to the remote repository please ensure␣
↪that you have signed into docker from your node, if not please type the followig␣
↪command to sign in
```

(continues on next page)

(continued from previous page)

```
docker login
# Please enter your username and password when prompted
```

20.2.2 Deploying modbus mapper application

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/device/modbus_mapper

# Please enter the following details in the deployment.yaml :-
#    1. Replace <edge_node_name> with the name of your edge node at spec.template.
↪spec.voluems.configMap.name
#    2. Replace <your_dockerhub_username> with your dockerhub username at spec.
↪template.spec.containers.image

kubectl create -f deployment.yaml
```

20.3 Modules

The modbus mapper consists of the following four major modules :-

1. Controller
2. Modbus Manager
3. Devicetwin Manager
4. File Watcher

20.3.1 Controller

The main entry is index.js. The controller module is responsible for subscribing edge MQTT devicetwin topic and perform check/modify operation on connected modbus devices. The controller is also responsible for loading the configuration and starting the other modules. The controller first connects the MQTT client to the broker to receive message of expected devicetwin value (using the mqtt configurations in conf.json), it then connects to the devices and check all the properties of devices every 2 seconds (based on dpl configuration provided in the configuration file) and the file watcher runs parallelly to check whether the dpl configuration file is changed.

20.3.2 Modbus Manager

Modbus Manager is a component which can perform an read or write action on modbus device. The following are the main responsibilities of this component: a) When controller receives message of expected devicetwin value, Modbus Manager will connect to the device and change the registers to make actual state equal to expected.

b) When controller checks all the properties of devices, Modbus Manager will connect to the device and read the actual value in registers accroding to the dpl configuration.

20.3.3 Devicetwin Manager

Devicetwin Manager is a component which can transfer the edge devicetwin message. The following are the main responsibilities of this component: a) To receive the edge devicetwin message from edge mqtt broker and parse message.

b) To report the actual value of device properties in devicetwin format to the cloud.

20.3.4 File Watcher

File Watcher is a component which can load dpl and mqtt configuration from configuration files.The following are the main responsibilities of this component: a) To monitor the dpl configuration file. If this file changed, file watcher will reload the dpl configuration to the mapper.

b) To load dpl and mqtt configuration when mapper starts first time.

CHAPTER 21

Pre-requisites

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

- Kubernetes
- Mosquitto
- Docker

CHAPTER 22

Setup KubeEdge

22.1 Prerequisites

- Install docker
- Install kubeadm/kubectl
- Creating cluster with kubeadm
- After initializing Kubernetes master, we need to expose insecure port 8080 for edgecontroller/kubectl to work with http connection to Kubernetes apiserver. Please follow below steps to enable http port in Kubernetes 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
```

- **Go** The minimum required go version is 1.11. You can install this version by using this website.
- (**Optional**)KubeEdge also supports https connection to Kubernetes apiserver. Follow the steps in Kubernetes Documentation to create the kubeconfig file.

  Enter the path to kubeconfig file in controller.yaml

```
controller:
  kube:
    ...
    kubeconfig: "path_to_kubeconfig_file" #Enter path to kubeconfig file to␣
↪enable https connection to k8s apiserver
```

22.1.1 Clone KubeEdge

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

22.1.2 Configuring MQTT mode

The Edge part of KubeEdge uses MQTT for communication between deviceTwin and devices. KubeEdge supports 3 MQTT modes:

1. internalMqttMode: internal mqtt broker is enabled.
2. bothMqttMode: internal as well as external broker are enabled.
3. externalMqttMode: only external broker is enabled.

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

To use KubeEdge in double mqtt or external mode, you need to make sure that mosquitto or emqx edge is installed on the edge node as an MQTT Broker.

22.1.3 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.

```
# $GOPATH/src/github.com/kubeedge/kubeedge/build/tools/certgen.sh genCertAndKey edge
```

The cert/key will be generated in the `/etc/kubeedge/ca` and `/etc/kubeedge/certs` respectively.

22.2 Run KubeEdge

22.2.1 Run Cloud

Run as a binary

- Firstly, make sure gcc is already installed on your host. You can verify it via:

```
gcc --version
```

- Build Cloud and edge

```
cd $GOPATH/src/github.com/kubeedge/kubeedge
make
```

- Build Cloud

```
cd $GOPATH/src/github.com/kubeedge/kubeedge
make all WHAT=cloud
```

- The path to the generated certificates should be updated in `$GOPATH/src/github.com/kubeedge/kubeedge/cloud/conf/controller.yaml`. Please update the correct paths for the following :

  - cloudhub.ca
  - cloudhub.cert
  - cloudhub.key

- Create device model and device CRDs.

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/build/crds/devices
kubectl create -f devices_v1alpha1_devicemodel.yaml
kubectl create -f devices_v1alpha1_device.yaml
```

- Run cloud

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

Run as Kubernetes deployment

22.2.2 Run Edge

Deploy the Edge node

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.

- Modify the `$GOPATH/src/github.com/kubeedge/kubeedge/build/node.json` file and change `metadata.name` to the name of the edge node
- Make sure role is set to edge for the node. For this a key of the form `"node-role.kubernetes.io/edge"` must be present in `labels` tag of `metadata`.
- Please ensure to add the label `node-role.kubernetes.io/edge` to the `build/node.json` file.

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "fb4ebb70-2783-42b8-b3ef-63e2fd6d242e",
    "labels": {
      "name": "edge-node",
      "node-role.kubernetes.io/edge": ""
    }
  }
}
```

- If role is not set for the node, the pods, configmaps and secrets created/updated in the cloud cannot be synced with the node they are targeted for.
- Deploy node

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

- Transfer the certificate file to the edge node

Run Edge

Run as a binary

- Build Edge

```
cd $GOPATH/src/github.com/kubeedge/kubeedge
make all WHAT=edge
```

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

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/edge
make edge_cross_build
```

KubeEdge can also be compiled with a small binary size. Please follow the below steps to build a binary of lesser size:

```
apt-get install upx-ucl
cd $GOPATH/src/github.com/kubeedge/kubeedge/edge
make edge_small_build
```

Note: If you are using the smaller version of the binary, it is compressed using upx, therefore the possible side effects of using upx compressed binaries like more RAM usage, lower performance, whole code of program being loaded instead of it being on-demand, not allowing sharing of memory which may cause the code to be loaded to memory more than once etc. are applicable here as well.

- Modify the `$GOPATH/src/github.com/kubeedge/kubeedge/edge/conf/edge.yaml` configuration file
  - Replace `edgehub.websocket.certfile` and `edgehub.websocket.keyfile` with your own certificate path
  - Update the IP address of the master in the `websocket.url` field.
  - replace `fb4ebb70-2783-42b8-b3ef-63e2fd6d242eq` with edge node name in edge.yaml for the below fields :
    - `websocket:URL`
    - `controller:node-id`
    - `edged:hostname-override`
- Run edge

```
# run mosquitto
mosquitto -d -p 1883
# or run emqx edge
# emqx start

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

Note: Please run edge using the users who have root permission.

Run as container

Run as Kubernetes deployment

Check status

After the Cloud and Edge parts have started, you can use below command to check the edge node status.

```
kubectl get nodes
```

Please make sure the status of edge node you created is **ready**.

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

22.3 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.

Then you can use below command to check if the application is normally running.

```
kubectl get pods
```

22.4 Run Tests

22.4.1 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
```

22.4.2 Run Edge Integration Tests

```
make edge_integration_test
```

22.4.3 Details and use cases of integration test framework

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

CHAPTER 23

Getting Started with KubeEdge Installer

Please refer to KubeEdge Installer proposal document for details on the motivation of having KubeEdge Installer. It also explains the functionality of the proposed commands. KubeEdge Installer Doc

23.1 Limitation

- Currently support of `KubeEdge installer` is available only for Ubuntu OS. CentOS support is in-progress.

23.2 Downloading KubeEdge Installer

1. Go to KubeEdge Release page and download `keadm-$VERSION-$OS-$ARCH.tar.gz`..
2. Untar it at desired location, by executing `tar -xvzf keadm-$VERSION-$OS-$ARCH.tar.gz`.
3. kubeedge folder is created after execution the command.

23.3 Building from source

1. Download the source code either by

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

1. `cd $GOPATH/src/github.com/kubeedge/kubeedge/keadm`
2. `make`
3. Binary `keadm` is available in current path

23.4 Installing KubeEdge Master Node (on the Cloud) component

Referring to `KubeEdge Installer Doc`, the command to install KubeEdge cloud component (edge controller) and pre-requisites. Port 8080, 6443 and 10000 in your cloud component needs to be accessible for your edge nodes.

- Execute `keadm init`

23.4.1 Command flags

The optional flags with this command are mentioned below

```
$  keadm init --help

keadm init command bootstraps KubeEdge's cloud component.
It checks if the pre-requisites are installed already,
If not installed, this command will help in download,
install and execute on the host.

Usage:
  keadm init [flags]

Examples:

keadm init

Flags:
      --docker-version string[="18.06.0"]          Use this key to download and use␣
↪the required Docker version (default "18.06.0")
  -h, --help                                       help for init
      --kubeedge-version string[="0.3.0-beta.0"]   Use this key to download and use␣
↪the required KubeEdge version (default "0.3.0-beta.0")
      --kubernetes-version string[="1.14.1"]       Use this key to download and use␣
↪the required Kubernetes version (default "1.14.1")
```

1. `--docker-version`, if mentioned with any version > 18.06.0, will install the same on the host. Default is 18.06.0. It is optional.
2. `--kubernetes-version`, if mentioned with any version > 1.14.1, will install the same on the host. Default is 1.14.1. It is optional. It will install `kubeadm`, `kubectl` and `kubelet` in this host.
3. `--kubeedge-version`, if mentioned with any version > 0.2.1, will install the same on the host. Default is 0.3.0-beta.0. It is optional.

command format is

```
keadm init --docker-version=<expected version> --kubernetes-version=<expected version>
↪ --kubeedge-version=<expected version>
```

NOTE: Version mentioned as defaults for Docker and K8S are being tested with.

23.5 Installing KubeEdge Worker Node (at the Edge) component

Referring to `KubeEdge Installer Doc`, the command to install KubeEdge Edge component (edge core) and pre-requisites

- Execute `keadm join <flags>`

23.5.1 Command flags

The optional flags with this command are shown in below shell

```
$  keadm join --help

"keadm join" command bootstraps KubeEdge's edge component.
It checks if the pre-requisites are installed already,
If not installed, this command will help in download,
to install the prerequisites.
It will help the edge node to connect to the cloud.

Usage:
  keadm join [flags]

Examples:

keadm join --edgecontrollerip=<ip address> --edgenodeid=<unique string as edge␣
↪identifier>

  - For this command --edgecontrollerip flag is a Mandatory flag
  - This command will download and install the default version of pre-requisites and␣
↪KubeEdge

keadm join --edgecontrollerip=10.20.30.40 --edgenodeid=testing123 --kubeedge-
↪version=0.2.1 --k8sserverip=50.60.70.80:8080

  - In case, any option is used in a format like as shown for "--docker-version" or "-
↪-docker-version=", without a value
        then default values will be used.
        Also options like "--docker-version", and "--kubeedge-version", version␣
↪should be in
        format like "18.06.3" and "0.2.1".

Flags:
      --docker-version string[="18.06.0"]          Use this key to download and use␣
↪the required Docker version (default "18.06.0")
  -e, --edgecontrollerip string                    IP address of KubeEdge␣
↪edgecontroller
  -i, --edgenodeid string                          KubeEdge Node unique␣
↪identification string, If flag not used then the command will generate a unique id␣
↪on its own
  -h, --help                                       help for join
  -k, --k8sserverip string                         IP:Port address of K8S API-Server
      --kubeedge-version string[="0.3.0-beta.0"]   Use this key to download and use␣
↪the required KubeEdge version (default "0.3.0-beta.0")
```

1. For KubeEdge flag the functionality is same as mentioned in `keadm init`
2. -k, –k8sserverip, It should be in the format IPAddress:Port, where the default port is 8080. Please see the example above.

IMPORTANT NOTE: The KubeEdge version used in cloud and edge side should be same.

23.6 Reset KubeEdge Master and Worker nodes

Referring to `KubeEdge Installer Doc`, the command to stop KubeEdge cloud (edge controller). It doesn't uninstall/remove any of the pre-requisites.

- Execute `keadm reset`

23.6.1 Command flags

```
keadm reset --help

keadm reset command can be executed in both cloud and edge node
In master node it shuts down the cloud processes of KubeEdge
In worker node it shuts down the edge processes of KubeEdge

Usage:
  keadm reset [flags]

Examples:

For master node:
keadm reset

For worker node:
keadm reset --k8sserverip 10.20.30.40:8080

Flags:
  -h, --help                 help for reset
  -k, --k8sserverip string   IP:Port address of cloud components host/VM
```

23.7 Simple steps to bring up KubeEdge setup and deploy a pod

NOTE: All the below steps are executed as root user, to execute as sudo user ,Please add **sudo** infront of all the commands

23.7.1 1. Deploy KubeEdge edgeController (With K8S Cluster)

Install tools with the particular version

```
keadm init --kubeedge-version=<kubeedge Version>  --kubernetes-version=<kubernetes␣
↪Version> --docker-version=<Docker version>
```

Install tools with the default version

```
keadm init --kubeedge-version= --kubernetes-version= --docker-version
or
keadm init
```

NOTE: On the console output, obeserve the below line

kubeadm join **192.168.20.134**:6443 –token 2lze16.l06eeqzgdz8sfcvh –discovery-token-ca-cert-hash sha256:1e5c808e1022937474ba264bb54fea42b05eddb9fde2d35c9cad5b83cf5ef9acAfter Kubeedge init ,please note the **cloudIP** as highlighted above generated from console output and port is **8080**.

23.7.2 2. Manually copy certs.tgz from cloud host to edge host(s)

On edge host

```
mkdir -p /etc/kubeedge
```

On cloud host

```
cd /etc/kubeedge/
scp -r certs.tgz username@ipEdgevm:/etc/kubeedge
```

On edge host untar the certs.tgz file

```
cd /etc/kubeedge
tar -xvzf certs.tgz
```

23.7.3 3. Deploy KubeEdge edge core

Install tools with the particular version

```
keadm join --edgecontrollerip=<cloudIP> --edgenodeid=<unique string as edge␣
↪identifier> --k8sserverip=<cloudIP>:8080 --kubeedge-version=<kubeedge Version> --
↪docker-version=<Docker version>
```

Install tools with the default version

```
keadm join --edgecontrollerip=<cloudIP> --edgenodeid=<unique string as edge␣
↪identifier> --k8sserverip=<cloudIP>:8080 --kubeedge-version=<kubeedge Version> --
↪docker-version=<Docker version>
```

Sample execution output:

```
# ./keadm join --edgecontrollerip=192.168.20.50 --edgenodeid=testing123 --
↪k8sserverip=192.168.20.50:8080
Same version of docker already installed in this host
Host has mosquit+ already installed and running. Hence skipping the installation␣
↪steps !!!
Expected or Default KubeEdge version 0.3.0-beta.0 is already downloaded
kubeedge/
kubeedge/edge/
kubeedge/edge/conf/
kubeedge/edge/conf/modules.yaml
kubeedge/edge/conf/logging.yaml
kubeedge/edge/conf/edge.yaml
kubeedge/edge/edge_core
kubeedge/cloud/
```

(continues on next page)

(continued from previous page)

```
kubeedge/cloud/edgecontroller
kubeedge/cloud/conf/
kubeedge/cloud/conf/controller.yaml
kubeedge/cloud/conf/modules.yaml
kubeedge/cloud/conf/logging.yaml
kubeedge/version

KubeEdge Edge Node: testing123 successfully add to kube-apiserver, with operation␣
↪status: 201 Created
Content {"kind":"Node","apiVersion":"v1","metadata":{"name":"testing123","selfLink":"/
↪api/v1/nodes/testing123","uid":"87d8d7a3-7acd-11e9-b86b-286ed488c645",
↪"resourceVersion":"3864","creationTimestamp":"2019-05-20T07:04:37Z","labels":{"name
↪":"edge-node"}},"spec":{"taints":[{"key":"node.kubernetes.io/not-ready","effect":
↪"NoSchedule"}]},"status":{"daemonEndpoints":{"kubeletEndpoint":{"Port":0}},"nodeInfo
↪":{"machineID":"","systemUUID":"","bootID":"","kernelVersion":"","osImage":"",
↪"containerRuntimeVersion":"","kubeletVersion":"","kubeProxyVersion":"",
↪"operatingSystem":"","architecture":""}}}

KubeEdge edge core is running, For logs visit /etc/kubeedge/kubeedge/edge/
#
```

Note:Cloud IP refers to IP generated ,from the step 1 as highlighted

23.7.4 4. Edge node status on edgeController (master node) console

On cloud host run,

```
kubectl get nodes

NAME          STATUS      ROLES     AGE      VERSION
testing123    Ready       <none>    6s       0.3.0-beta.0
```

Check if the edge node is in ready state

23.7.5 5.Deploy a sample pod from Cloud VM

https://github.com/kubeedge/kubeedge/blob/master/build/deployment.yaml

Copy the deployment.yaml from the above link in cloud host,run

```
kubectl create -f deployment.yaml
deployment.apps/nginx-deployment created
```

23.7.6 6.Pod status

Check the pod is up and is running state

```
kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-d86dfb797-scfzz    1/1     Running   0          44s
```

Check the deployment is up and is running state

```
kubectl get deployments

NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   1/1     1            1           63s
```

23.8 Errata

1.If GPG key for docker repo fail to fetch from key server. Please refer Docker GPG error fix

2.After kubeadm init, if you face any errors regarding swap memory and preflight checks please refer Kubernetes preflight error fix

CHAPTER 24

Cross Compiling KubeEdge

24.1 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 25

Measuring memory footprint of EdgeCore

25.1 Why measuring memory footprint

- This platform is also tagged for a light weighted edge computing deployment
- To be able to be deployed over devices with less resources (for example, 256MB RAM)
- It is required to know by deploying as many as possible pods, it showcases as much as less possible memory footprint

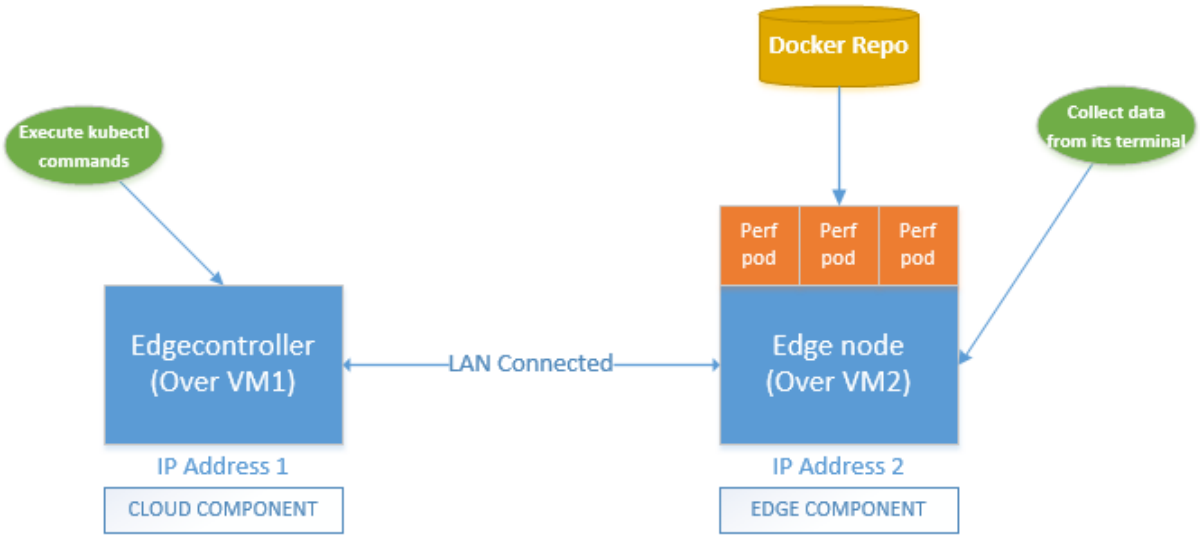
25.2 KPI's measured

- %CPU
- %Memory
- Resident Set Size (RSS)

25.3 How to test

After deployment and provisioning of KubeEdge cloud and edge components in 2 VM's (supported and tested over Ubuntu 16.04) respectively, start deploying pods from 0 to 100 in steps of 5. Keep capturing above KPI's using standard linux `ps` commands, after each step.

25.3.1 Test setup



KubeEdge Test Setup

Fig 1: KubeEdge Test Setup

25.3.2 Creating a setup

Requirements

- Host machine's or VM's resource requirements can mirror the edge device of your choice
- Resources used for above setup are 4 CPU, 8GB RAM and 200 GB Disk space. OS is Ubuntu 16.04.
- Docker image used to deploy the pods in edge, needs to be created. The steps are:
  1. Go to github.com/kubeedge/kubeedge/edge/hack/memfootprint-test/
  2. Using the Dockerfile available here and create docker image (`perftestimg:v1`).
  3. Execute the docker command `sudo docker build --tag "perftestimg:v1" .`, to get the image.

Installation

- For KubeEdge Cloud and Edge:

  Please follow steps mentioned in KubeEdge README.md
- For docker image:
- Deploy docker registry to either edge on any VM or host which is reachable to edge. Follow the steps mentioned here: https://docs.docker.com/registry/deploying/
- Create `perftestimg:v1` docker image on the above mentioned host
- Then push this image to docker registry using `docker tag` and `docker push` commands (Refer: Same docker registry url mentioned above) [Use this image's metadata in pod deployment yaml]

25.3.3 Steps

1. Check edge node is connected to cloud. In cloud console/terminal, execute the below command

```
root@ubuntu:~/edge/pod_yamls# kubectl get nodes
NAME                                  STATUS     ROLES    AGE     VERSION
192.168.20.31                         Unknown    <none>   11s
ubuntu                                NotReady   master   5m22s   v1.14.0
```

1. On cloud, modify deployment yaml (github.com/kubeedge/kubeedge/edge/hack/memfootprint-test/perftestimg.yaml), set the image name and set spec.replica as 5
2. Execute `sudo kubectl create -f ./perftestimg.yaml` to deploy the first of 5 pods in edge node
3. Execute `sudo kubectl get pods | grep Running | wc` to check if all the pods come to Running state. Once all pods come to running state, go to edge VM
4. On Edge console, execute `ps -aux | grep edge_core`. The output shall be something like:

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      102452  1.0  0.5 871704 42784 pts/0    Sl+  17:56   0:00 ./edge_core
root      102779  0.0  0.0  14224   936 pts/2    S+   17:56   0:00 grep --color=auto␣
↪edge
```

1. Collect %CPU, %MEM and RSS from respective columns and record
2. Repeat step 2 and this time increase the replica by 5
3. This time execute `sudo kubectl apply -f <PATH>/perftestimg.yaml`
4. Repeat steps from **4 to 6**.
5. Now **repeat steps from 7 to 9**, till the replica count reaches 100

CHAPTER 26

Try KubeEdge with HuaweiCloud (IEF)

26.1 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 27

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.

27.1 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 28

Unit Test Guide

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

28.1 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.

28.2 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.

28.3 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.

28.4 Writing UT using GoMock

28.4.1 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 29

Device Management User Guide

KubeEdge supports device management with the help of Kubernetes CRDs and a Device Mapper (explained below) corresponding to the device being used. We currently manage devices from the cloud and synchronize the device updates between edge nodes and cloud, with the help of device controller and device twin modules.

29.1 Device Model

A `device model` describes the device properties exposed by the device and property visitors to access these properties. A device model is like a reusable template using which many devices can be created and managed.

Details on device model definition can be found here.

A sample device model can be found here

29.2 Device Instance

A `device` instance represents an actual device object. It is like an instantiation of the `device model` and references properties defined in the model. The device spec is static while the device status contains dynamically changing data like the desired state of a device property and the state reported by the device.

Details on device instance definition can be found here.

A sample device model can be found here.

29.3 Device Mapper

Mapper is an application that is used to connect and and control devices. Following are the responsibilities of mapper:

1. Scan and connect to the device.
2. Report the actual state of twin-attributes of device.

3. Map the expected state of device-twin to actual state of device-twin.
4. Collect telemetry data from device.
5. Convert readings from device to format accepted by KubeEdge.
6. Schedule actions on the device.
7. Check health of the device.

Mapper can be specific to a protocol where standards are defined i.e Bluetooth, Zigbee, etc or specific to a device if it a custom protocol.

Mapper design details can be found here

An example of a mapper application created to support bluetooth protocol can be found here

29.4 Usage of Device CRD

The following are the steps to

1. Create a device model in the cloud node.

```
kubectl apply -f <path to device model yaml>
```

2. Create a device instance in the cloud node.

```
kubectl apply -f <path to device instance yaml>
```

   Note: Creation of device instance will also lead to the creation of a config map which will contain information about the devices which are required by the mapper applications The name of the config map will be as follows: device-profile-config-< edge node name >. The updation of the config map is handled internally by the device controller.

3. Run the mapper application corresponding to your protocol.
4. Edit the status section of the device instance yaml created in step 2 and apply the yaml to change the state of device twin. This change will be reflected at the edge, through the device controller and device twin modules. Based on the updated value of device twin at the edge the mapper will be able to perform its operation on the device.
5. The reported values of the device twin are updated by the mapper application at the edge and this data is synced back to the cloud by the device controller. User can view the update at the cloud by checking his device instance object.

```
Note: Sample device model and device instance for a few protocols can be found at
↪$GOPATH/src/github.com/kubeedge/kubeedge/build/crd-samples/devices
```

CHAPTER 30

Edgemesh test env config guide

30.1 install Containerd

- please refer to the following link to install

  https://kubernetes.io/docs/setup/production-environment/container-runtimes/#containerd

30.2 install CNI plugin

- get cni plugin and these five version are sopported (0.1.0, 0.2.0, 0.3.0, 0.3.1, 0.4.0)
- and then use 'tar -zxvf' to extract to /opt/cni/bin
- configure cni plugin

```
$ mkdir -p /etc/cni/net.d/
```

- please Make sure docker0 does not exist !!
- field "bridge" must be "docker0"
- field "isGateway" must be true

```
$ cat >/etc/cni/net.d/10-mynet.conf <<EOF
{
        "cniVersion": "0.2.0",
        "name": "mynet",
        "type": "bridge",
        "bridge": "docker0",
        "isGateway": true,
        "ipMasq": true,
        "ipam": {
                "type": "host-local",
                "subnet": "10.22.0.0/16",
```

(continues on next page)

(continued from previous page)

```
                "routes": [
                        { "dst": "0.0.0.0/0" }
                ]
        }
}
EOF
```

30.3 Configure port mapping manually on node on which server is running

can see the examples in the next section.

- step1. execute iptables command as follows

```
$ iptables -t nat -N PORT-MAP
$ iptables -t nat -A PORT-MAP -i docker0 -j RETURN
$ iptables -t nat -A PREROUTING -p tcp -m addrtype --dst-type LOCAL -j PORT-MAP
$ iptables -t nat -A OUTPUT ! -d 127.0.0.0/8 -p tcp -m addrtype --dst-type LOCAL -j␣
↪PORT-MAP
$ iptables -P FORWARD ACCEPT
```

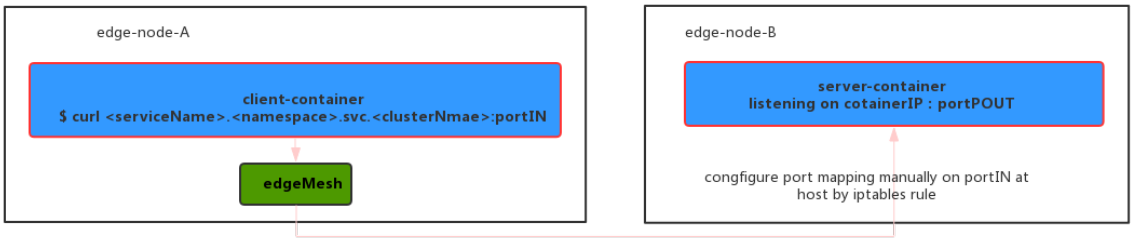
- step2. execute iptables command as follows
  - **portIN** is the service map at the host
  - **containerIP** is the IP in the container. Can be find out on master by **kubectl get pod -o wide**
  - **portOUT** is the port that monitored In-container

```
$ iptables -t nat -A PORT-MAP ! -i docker0 -p tcp -m tcp --dport portIN -j DNAT --to-
↪destination containerIP:portOUT
```

- by the way, If you redeployed the service,you can use the command as follows to delete the rule, and perform the second step again.

```
 $ iptables -t nat -D PORT-MAP 2
```

30.4 Example for Edgemesh test env

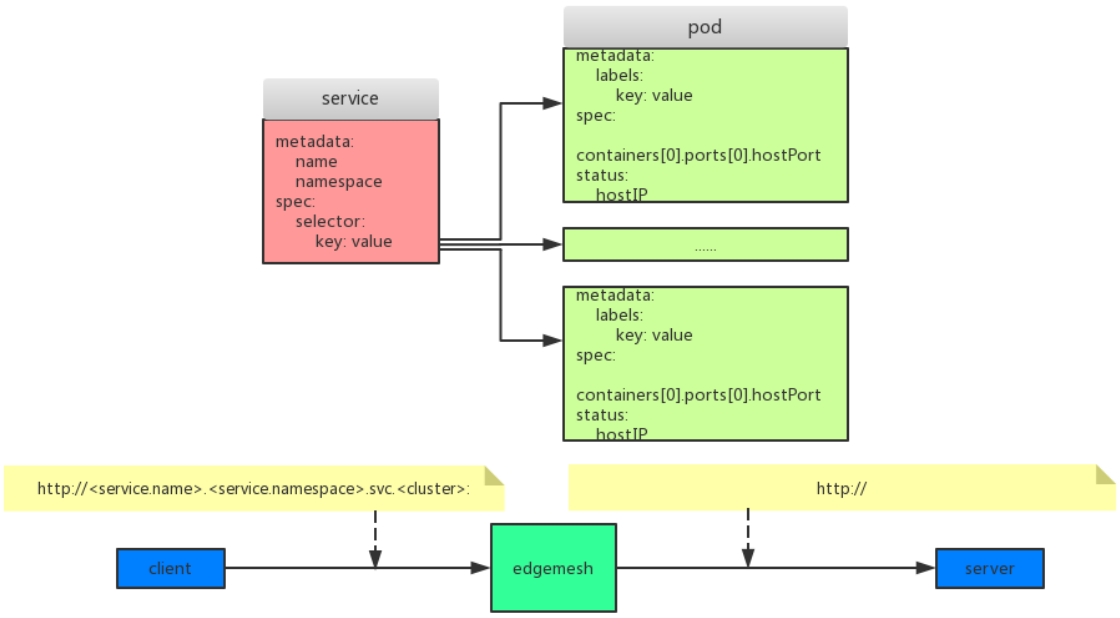


edgemesh

test env example

30.5 Edgemesh end to end test guide

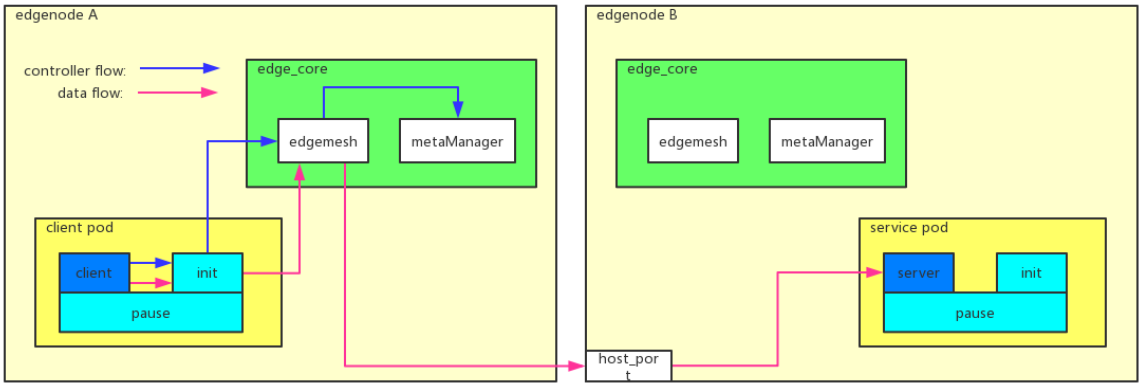
30.5.1 model



model

1. a headless service(a service with selector but ClusterIP is None)
2. one or more pods' labels match the headless service's selector
3. so when request a server: `<service_name>.<service_namespace>.svc.<cluster>`:
    1. get the service's name and namespace from domain name
    2. query the backend pods from metaManager by service's namespace and name
    3. load balance return the real backend container's hostIP and hostPort

30.5.2 flow from client to server



flow

1. client request to server's domain name
2. DNS request hijacked to edgemesh by iptables, return a fake ip
3. request hijacked to edgemesh by iptables
4. edgemesh resolve request, get domain name, protocol, request and so on
5. edgemesh load balance:
    1. get the service name and namespace from the domain name
    2. query backend pod of the service from metaManager
    3. choose a backend based on strategy
6. edgemesh transport request to server wait server response and then response to client

30.5.3 how to test end to end

- create a headless service(**no need specify port**):

```
apiVersion: v1
kind: Service
metadata:
  name: test-headless
  namespace: default
spec:
  clusterIP: None
  selector:
    app: whatapp
```

- create server deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-deployment
  labels:
```

(continues on next page)

(continued from previous page)

```
    app: whatapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: whatapp
  template:
    metadata:
      labels:
        app: whatapp
    spec:
      nodeSelector:
        name: ${label of the node server run}
      containers:
      - name: whatapp
        image: docker.io/cloudnativelabs/whats-my-ip:latest
        ports:
        - containerPort: 8080
          hostPort: 8080
      initContainers:
      - args:
        - -p
        - "8080"
        - -i
        - "192.168.1.2/24,156.43.2.1/26"
        - -t
        - "12345,5432,8080"
        - -c
        - "9292"
        name: init1
        image: docker.io/ytsobd/edgemesh_init:v1.0
        securityContext:
          privileged: true
```

- create client deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client-deployment
  labels:
    app: whatapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app: client
  template:
    metadata:
      labels:
        app: client
    spec:
      nodeSelector:
        name: ${label of the node server run}
      containers:
      - name: whatapp
        image: docker.io/cloudnativelabs/whats-my-ip:latest
```

(continues on next page)

(continued from previous page)

```
      initContainers:
      - args:
        - -p
        - "8080"
        - -i
        - "192.168.1.2/24,156.43.2.1/26"
        - -t
        - "12345,5432,8080"
        - -c
        - "9292"
        name: init1
        image: docker.io/ytsobd/edgemesh_init:v1.0
        securityContext:
          privileged: true
```

note: -p: whitelist, only port in whitelist can go out from client to edgemesh then to server

- client request server: exec into client container and then run command: `curl http://test-headless.default.svc.cluster:8080`, will get the response from server like: `HOSTNAME:test-app-686c6dbf98-6hrdq IP:10.11.0.4`
- **there is two ways to exec the 'curl' command to access your service**
- 1st: use 'ctr' command attach in the container and make sure there is 'curl' command in the container

```
$ ctr -n k8s.io t exec --exec-id 123 <containerID> sh
```

- 2nd: switch the network namespace.(Recommended Use)

```
# first step get the id,this command will return a id start with 'cni-xx'. and make␣
↪sure the 'xx' is related to the pod which you can get from 'kubectl describe
↪<podName>'
$ ip netns
# and the use this id to switch the net namespace. And the you can exec curl to␣
↪access the service
$ ip netns exec <id> bash
```

CHAPTER 31

FAQs

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