KubeEdge Documentation

Release 0.1

KubeEdge

Mar 15, 2021

Contents

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

CHAPTER 1

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 open-sourced.

1.1 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**

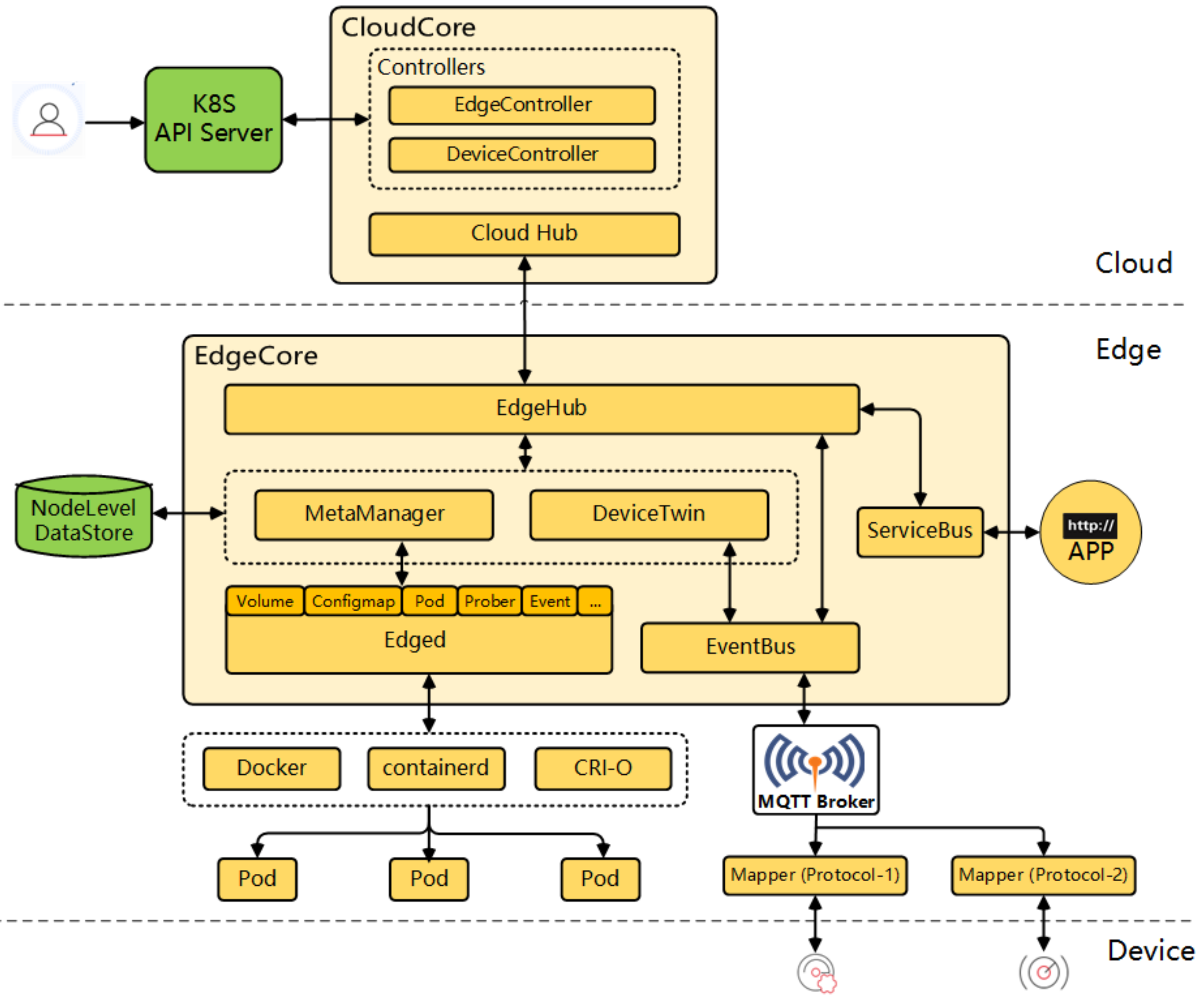
  It is easy to get and deploy existing complicated machine learning, image recognition, event processing and other high-level applications to the Edge.

1.2 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).

1.3 Architecture



KubeEdge Architecture

CHAPTER 2

Getting started

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

In this quick-start guide, we will explain:

- How to ask questions, build and contribute to KubeEdge.
- A few common ways of deploying KubeEdge.
- Links for further reading.

2.1 Dependencies

For cloud side, we need:

- Kubernetes cluster

For edge side, we need:

- Container runtimes, now we support:
- MQTT Server(Optional)

2.2 Get KubeEdge!

You can find the latest KubeEdge release here.

During release, we build tarballs for major platforms and release docker images in kubeedge dockerhub.

If you want to build KubeEdge from source, check this doc.

2.3 Deploying KubeEdge

Check setup docs.

2.4 Contributing

Contributions are very welcome! See our *CONTRIBUTING.md* for more information.

2.5 Community

KubeEdge is an open source project and we value and welcome new contributors and members of the community. Here are ways to get in touch with the community:

CHAPTER 3

Roadmap

This document defines a high level roadmap for KubeEdge development.

The milestones defined in GitHub represent the most up-to-date plans.

The roadmap below outlines new features that will be added to KubeEdge.

3.1 2021 H1

3.1.1 Core framework

Edge side list-watch

- Support list-watch interface at edge

Custom message transmission between cloud and edge

- Support transmission of custom message between cloud and edge

Support multi-instance cloudcore

Integration and verification of third-party CNI

- Flannel, Calico, etc.

Integration and verification of third-party CSI

- Rook, OpenEBS, etc.

Support managing clusters at edge from cloud (aka. EdgeSite)

Support ingress/gateway at edge.

3.1.2 Maintainability

Deployment optimization

- Easier deployment
- Admission controller automated deployment

Automatic configuration of edge application offline migration time

- Modify Default tolerationSeconds Automatically

3.1.3 IOT Device management

Device Mapper framework standard and framework generator

- Formulate mapper framework standard

Support mappers of more protocols

- OPC-UA mapper
- ONVIF mapper

3.1.4 Security

Complete security vulnerability scanning

3.1.5 Test

Improve the performance and e2e tests with more metrics and scenarios.

3.1.6 Edge-cloud synergy AI

Supports KubeFlow/ONNX/Pytorch/Mindspore

Edge-cloud synergy training and inference

3.1.7 MEC

Cross-edge cloud service discovery

5G network capability exposure

3.2 2021 H2

3.2.1 Core framework

Custom message transmission between cloud and edge

- Support CloudEvent protocol

Cross subnet communication of Data plane

- Edge-edge cross subnet
- Edge-cloud cross subnet

Unified Service Mesh support (Integrate with Istio/OSM etc.)

Cloud-edge synergy monitoring

- Provide support with prometheus push-gateway mode
- Data management with support for ingestion of telemetry data and analytics at the edge.

3.2.2 IOT Device management

Device Mapper framework standard and framework generator

- Develop mapper framework generator

Support mappers of more protocols

- GB/T 28181 mapper

3.2.3 Edge-cloud synergy AI

Intelligent edge benchmark

3.2.4 MEC

Cloud-network convergence

Service catalog

Cross-edge cloud application roaming

CHAPTER 4

Deploying using Keadm

Keadm is used to install the cloud and edge components of KubeEdge. It is not responsible for installing K8s and runtime, so check dependences section in this *doc* first.

Please refer kubernetes-compatibility to get **Kubernetes compatibility** and determine what version of Kubernetes would be installed.

4.1 Limitation

- Currently support of `keadm` is available for Ubuntu and CentOS OS. RaspberryPi supports is in-progress.
- Need super user rights (or root rights) to run.

4.2 Setup Cloud Side (KubeEdge Master Node)

By default ports `10000` and `10002` in your cloudcore needs to be accessible for your edge nodes.

Note: port `10002` only needed since 1.3 release.

`keadm init` will install cloudcore, generate the certs and install the CRDs. It also provides a flag by which a specific version can be set.

IMPORTANT NOTE:

1. At least one of kubeconfig or master must be configured correctly, so that it can be used to verify the version and other info of the k8s cluster.
2. Please make sure edge node can connect cloud node using local IP of cloud node, or you need to specify public IP of cloud node with `--advertise-address` flag.
3. `--advertise-address`(only work since 1.3 release) is the address exposed by the cloud side (will be added to the SANs of the CloudCore certificate), the default value is the local IP.

Example:

```
# keadm init --advertise-address="THE-EXPOSED-IP"(only work since 1.3 release)
```

Output:

```
Kubernetes version verification passed, KubeEdge installation will start...
...
KubeEdge cloudcore is running, For logs visit:  /var/log/kubeedge/cloudcore.log
```

4.3 (Only Needed in Pre 1.3 Release) Manually copy certs.tgz from cloud host to edge host(s)

Note: Since release 1.3, feature `EdgeNode auto TLS Bootstrapping` has been added and there is no need to manually copy certificate.

Now users still need to copy the certs to the edge nodes. In the future, it will support the use of tokens for authentication.

On edge host:

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

On cloud host:

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

On edge host untar the certs.tgz file

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

4.4 Setup Edge Side (KubeEdge Worker Node)

4.4.1 Get Token From Cloud Side

Run `keadm gettoken` in **cloud side** will return the token, which will be used when joining edge nodes.

```
# keadm gettoken
27a37ef16159f7d3be8fae95d588b79b3adaaf92727b72659eb89758c66ffda2.
↪eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1OTAyMTYwNzd9.
↪JBj8LLYWXwbbvHKffJBpPd5CyxqapRQYDIXtFZErgYE
```

4.4.2 Join Edge Node

`keadm join` will install edgecore and mqtt. It also provides a flag by which a specific version can be set.

Example:

```
# keadm join --cloudcore-ipport=192.168.20.50:10000 --
↪token=27a37ef16159f7d3be8fae95d588b79b3adaaf92727b72659eb89758c66ffda2.
↪eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1OTAyMTYwNzd9.
↪JBj8LLYWXwbbvHKffJBpPd5CyxqapRQYDIXtFZErgYE
```

(continues on next page)

(continued from previous page)

IMPORTANT NOTE:

1. `--cloudcore-ipport` flag is a mandatory flag.
2. If you want to apply certificate for edge node automatically, `--token` is needed.
3. The kubeEdge version used in cloud and edge side should be same.

Output:

```
Host has mosquit+ already installed and running. Hence skipping the installation␣
↪steps !!!
...
KubeEdge edgecore is running, For logs visit:  /var/log/kubeedge/edgecore.log
```

4.4.3 Enable `kubectl logs` Feature

Before metrics-server deployed, `kubectl logs` feature must be activated:

1. Make sure you can find the kubernetes `ca.crt` and `ca.key` files. If you set up your kubernetes cluster by `kubeadm` , those files will be in `/etc/kubernetes/pki/` dir.

   ```
   ls /etc/kubernetes/pki/
   ```

2. Set `CLOUDCOREIPS` env. The environment variable is set to specify the IP address of cloudcore, or a VIP if you have a highly available cluster.

   ```
   export CLOUDCOREIPS="192.168.0.139"
   ```

   (Warning: the same **terminal** is essential to continue the work, or it is necessary to type this command again.) Checking the environment variable with the following command:

   ```
   echo $CLOUDCOREIPS
   ```

3. Generate the certificates for **CloudStream** on cloud node, however, the generation file is not in the `/etc/kubeedge/`, we need to copy it from the repository which was git cloned from GitHub. Change user to root:

   ```
   sudo su
   ```

   Copy certificates generation file from original cloned repository:

   ```
   cp $GOPATH/src/github.com/kubeedge/kubeedge/build/tools/certgen.sh /etc/kubeedge/
   ```

   Change directory to the kubeedge directory:

   ```
   cd /etc/kubeedge/
   ```

   Generate certificates from **certgen.sh**

   ```
   /etc/kubeedge/certgen.sh stream
   ```

4. It is needed to set iptables on the host. (This command should be executed on every apiserver deployed node.)(In this case, this the master node, and execute this command by root.) Run the following command on the host on which each apiserver runs:

   Note: You need to set the cloudcoreips variable first

```
iptables -t nat -A OUTPUT -p tcp --dport 10350 -j DNAT --to $CLOUDCOREIPS:10003
```

> Port 10003 and 10350 are the default ports for the CloudStream and edgecore, use your own ports if you have changed them.

If you are not sure if you have setting of iptables, and you want to clean all of them. (If you set up iptables wrongly, it will block you out of your `kubectl logs` feature) The following command can be used to clean up iptables:

```
iptables -F && iptables -t nat -F && iptables -t mangle -F && iptables -X
```

5. Modify **both** `/etc/kubeedge/config/cloudcore.yaml` and `/etc/kubeedge/config/edgecore.yaml` on cloudcore and edgecore. Set up **cloudStream** and **edgeStream** to `enable:  true`. Change the server IP to the cloudcore IP (the same as $CLOUDCOREIPS).

   Open the YAML file in cloudcore:

   ```
   sudo nano /etc/kubeedge/config/cloudcore.yaml
   ```

   Modify the file in the following part (`enable:  true`):

   ```
   cloudStream:
     enable: true
     streamPort: 10003
     tlsStreamCAFile: /etc/kubeedge/ca/streamCA.crt
     tlsStreamCertFile: /etc/kubeedge/certs/stream.crt
     tlsStreamPrivateKeyFile: /etc/kubeedge/certs/stream.key
     tlsTunnelCAFile: /etc/kubeedge/ca/rootCA.crt
     tlsTunnelCertFile: /etc/kubeedge/certs/server.crt
     tlsTunnelPrivateKeyFile: /etc/kubeedge/certs/server.key
     tunnelPort: 10004
   ```

   Open the YAML file in edgecore:

   ```
   sudo nano /etc/kubeedge/config/edgecore.yaml
   ```

   Modify the file in the following part (`enable:  true`), (`server:  192.168.0.193:10004`):

   ```
   edgeStream:
     enable: true
     handshakeTimeout: 30
     readDeadline: 15
     server: 192.168.0.139:10004
     tlsTunnelCAFile: /etc/kubeedge/ca/rootCA.crt
     tlsTunnelCertFile: /etc/kubeedge/certs/server.crt
     tlsTunnelPrivateKeyFile: /etc/kubeedge/certs/server.key
     writeDeadline: 15
   ```

6. Restart all the cloudcore and edgecore.

   ```
   sudo su
   ```

   cloudCore:

   ```
   pkill cloudcore
   nohup cloudcore > cloudcore.log 2>&1 &
   ```

   edgeCore:

```
systemctl restart edgecore.service
```

If you fail to restart edgecore, check if that is because of `kube-proxy` and kill it. **kubeedge** reject it by default, we use a succedaneum called edgemesh

Note: the importance is to avoid `kube-proxy` being deployed on edgenode. There are two methods to solve it:

1. Add the following settings by calling `kubectl edit daemonsets.apps -n kube-system kube-proxy`:

```
affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/edge
                operator: DoesNotExist
```

1. If you still want to run `kube-proxy`, ask **edgecore** not to check the environment by adding the env variable in `edgecore.service` :

   ```
   sudo vi /etc/kubeedge/edgecore.service
   ```

   - Add the following line into the **edgecore.service** file:

   ```
   Environment="CHECK_EDGECORE_ENVIRONMENT=false"
   ```

   - The final file should look like this:

   ```
   Description=edgecore.service

   [Service]
   Type=simple
   ExecStart=/root/cmd/ke/edgecore --logtostderr=false --log-file=/root/cmd/ke/
   ↪edgecore.log
   Environment="CHECK_EDGECORE_ENVIRONMENT=false"

   [Install]
   WantedBy=multi-user.target
   ```

4.4.4 Support Metrics-server in Cloud

1. The realization of this function point reuses cloudstream and edgestream modules. So you also need to perform all steps of *Enable `kubectl logs` Feature*.

2. Since the kubelet ports of edge nodes and cloud nodes are not the same, the current release version of metrics-server(0.3.x) does not support automatic port identification (It is the 0.4.0 feature), so you need to manually compile the image from master branch yourself now.

   Git clone latest metrics server repository:

```
git clone https://github.com/kubernetes-sigs/metrics-server.git
```

   Go to the metrics server directory:

```
cd metrics-server
```

Make the docker image:

```
make container
```

Check if you have this docker image:

```
docker images
```

~~~
Make sure you change the tag of image by using its IMAGE ID to be compactable with
↪image name in yaml file.

```bash
docker tag a24f71249d69 metrics-server-kubeedge:latest
```
~~~

1. Apply the deployment yaml. For specific deployment documents, you can refer to https://github.com/kubernetes-sigs/metrics-server/tree/master/manifests.

   Note: those iptables below must be applyed on the machine (to be exactly network namespace, so metrics-server needs to run in hostnetwork mode also) metric-server runs on.

   ```
   iptables -t nat -A OUTPUT -p tcp --dport 10350 -j DNAT --to $CLOUDCOREIPS:10003
   ```

   (To direct the request for metric-data from edgecore:10250 through tunnel between cloudcore and edgecore, the iptables is vitally important.)

   Before you deploy metrics-server, you have to make sure that you deploy it on the node which has apiserver deployed on. In this case, that is the master node. As a consequence, it is needed to make master node schedulable by the following command:

   ```
   kubectl taint nodes --all node-role.kubernetes.io/master-
   ```

   Then, in the deployment.yaml file, it must be specified that metrics-server is deployed on master node. (The hostname is chosen as the marked label.) In **metrics-server-deployment.yaml**

   ```
       spec:
         affinity:
           nodeAffinity:
             requiredDuringSchedulingIgnoredDuringExecution:
               nodeSelectorTerms:
               - matchExpressions:
                 #Specify which label in [kubectl get nodes --show-labels] you want
   ↪to match
                 - key: kubernetes.io/hostname
                   operator: In
                   values:
                   #Specify the value in key
                   - charlie-latest
   ```

IMPORTANT NOTE:

1. Metrics-server needs to use hostnetwork network mode.
2. Use the image compiled by yourself and set imagePullPolicy to Never.
3. Enable the feature of –kubelet-use-node-status-port for Metrics-server

Those settings need to be written in deployment yaml (metrics-server-deployment.yaml) file like this:

```
      volumes:
      # mount in tmp so we can safely use from-scratch images and/or read-only containers
      - name: tmp-dir
        emptyDir: {}
      hostNetwork: true                          #Add this line to enable hostnetwork mode
      containers:
      - name: metrics-server
        image: metrics-server-kubeedge:latest    #Make sure that the REPOSITORY and TAG are correct
        # Modified args to include --kubelet-insecure-tls for Docker Desktop (don't use this flag with a real k8s cluster!!)
        imagePullPolicy: Never                   #Make sure that the deployment uses the image you built up
        args:
          - --cert-dir=/tmp
          - --secure-port=4443
          - --v=2
          - --kubelet-insecure-tls
          - --kubelet-preferred-address-types=InternalDNS,InternalIP,ExternalIP,Hostname
          - --kubelet-use-node-status-port       #Enable the feature of --kubelet-use-node-status-port for Metrics-server
        ports:
        - name: main-port
          containerPort: 4443
          protocol: TCP
```

4.5 Reset KubeEdge Master and Worker nodes

4.5.1 Master

`keadm reset` will stop `cloudcore` and delete KubeEdge related resources from Kubernetes master like `kubeedge` namespace. It doesn't uninstall/remove any of the pre-requisites.

It provides a flag for users to specify kubeconfig path, the default path is `/root/.kube/config`.

Example:

```
# keadm reset --kube-config=$HOME/.kube/config
```

4.5.2 Node

`keadm reset` will stop `edgecore` and it doesn't uninstall/remove any of the pre-requisites.

CHAPTER 5

Deploying Locally

Deploying KubeEdge locally is used to test, never use this way in production environment.

5.1 Limitation

- Need super user rights (or root rights) to run.

5.2 Setup Cloud Side (KubeEdge Master Node)

5.2.1 Create CRDs

```
kubectl apply -f https://raw.githubusercontent.com/kubeedge/kubeedge/master/build/
↪crds/devices/devices_v1alpha2_device.yaml
kubectl apply -f https://raw.githubusercontent.com/kubeedge/kubeedge/master/build/
↪crds/devices/devices_v1alpha2_devicemodel.yaml
kubectl apply -f https://raw.githubusercontent.com/kubeedge/kubeedge/master/build/
↪crds/reliablesyncs/cluster_objectsync_v1alpha1.yaml
kubectl apply -f https://raw.githubusercontent.com/kubeedge/kubeedge/master/build/
↪crds/reliablesyncs/objectsync_v1alpha1.yaml
```

5.2.2 Prepare config file

```
# cloudcore --minconfig > cloudcore.yaml
```

please refer to configuration for cloud for details.

5.2.3 Run

```
# cloudcore --config cloudcore.yaml
```

Run `cloudcore -h` to get help info and add options if needed.

5.3 Setup Edge Side (KubeEdge Worker Node)

5.3.1 Prepare config file

- generate config file

```
# edgecore --minconfig > edgecore.yaml
```

- get token value at cloud side:

```
# kubectl get secret -nkubeedge tokensecret -o=jsonpath='{.data.tokendata}' | base64 -
↪d
```

- update token value in edgecore config file:

```
# sed -i -e "s|token: .*|token: ${token}|g" edgecore.yaml
```

The `token` is what above step get.

please refer to configuration for edge for details.

5.3.2 Run

If you want to run cloudcore and edgecore at the same host, run following command first:

```
# export CHECK_EDGECORE_ENVIRONMENT="false"
```

Start edgecore:

```
# edgecore --config edgecore.yaml
```

Run `edgecore -h` to get help info and add options if needed.

CHAPTER 6

Upgrading KubeEdge

Please refer to following guide to upgrade your KubeEdge cluster.

6.1 Backup

6.1.1 Database

Backup edgecore database at each edge node:

```
$ mkdir -p /tmp/kubeedge_backup
$ cp /var/lib/kubeedge/edgecore.db /tmp/kubeedge_backup/
```

6.1.2 Config(Optional)

You can keep old config to save some custom changes as you wish.

Note:

After upgrading, some options may be deleted and some may be added, please don't use old config directly.

6.1.3 Device related(Optional)

If you upgrade from 1.3 to 1.4, please note that we upgrade device API from v1alpha1 to v1alpha2.

You need to install Device v1alpha2 and DeviceModel v1alpha2, and manually convert their existing custom resources from v1alpha1 to v1alpha2.

It's recommended to keep v1alpha1 CRD and custom resources in the cluster or exported somewhere, in case any rollback is needed.

6.2 Stop Processes

Stop edgecore processes one by one, after ensuring all edgecore processes are stopped, stop cloudcore.

The way to stop depends on how you deploy:

- for binary or "keadm": use `kill`
- for "systemd": use `systemctl`

6.3 Clean up

```
$ rm -rf /var/lib/kubeedge /etc/kubeedge
```

6.4 Restore Database

Restore database at each edge node:

```
$ mkdir -p /var/lib/kubeedge
$ mv /tmp/kubeedge_backup/edgecore.db /var/lib/kubeedge/
```

6.5 Deploy

Read the setup for deployment.

CHAPTER 7

KubeEdge Configuration

KubeEdge requires configuration on both *Cloud side (KubeEdge Master)* and *Edge side (KubeEdge Worker Node)*

7.1 Configuration Cloud side (KubeEdge Master)

Setting up cloud side requires two steps

1. *Modification of the configuration files*
2. Edge node will be auto registered by default. *Users can still choose to register manually*.

7.1.1 Modification of the configuration file

Cloudcore requires changes in `cloudcore.yaml` configuration file.

Create and set cloudcore config file

Create the `/etc/kubeedge/config` folder

```
# the default configuration file path is '/etc/kubeedge/config/cloudcore.yaml'
# also you can specify it anywhere with '--config'
mkdir -p /etc/kubeedge/config/
```

Either create a minimal configuration with command `~/kubeedge/cloudcore --minconfig`

```
~/kubeedge/cloudcore --minconfig > /etc/kubeedge/config/cloudcore.yaml
```

or a full configuration with command `~/kubeedge/cloudcore --defaultconfig`

```
~/kubeedge/cloudcore --defaultconfig > /etc/kubeedge/config/cloudcore.yaml
```

Edit the configuration file

```
vim /etc/kubeedge/config/cloudcore.yaml
```

Verify the configurations before running `cloudcore`

Modification in cloudcore.yaml

In the cloudcore.yaml, modify the below settings.

1. Either `kubeAPIConfig.kubeConfig` or `kubeAPIConfig.master` : This would be the path to your kubeconfig file. It might be either

   ```
   /root/.kube/config
   ```

   or

   ```
   /home/<your_username>/.kube/config
   ```

   depending on where you have setup your kubernetes by performing the below step:

   ```
   To start using your cluster, you need to run the following as a regular user:

   mkdir -p $HOME/.kube
   sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
   sudo chown $(id -u):$(id -g) $HOME/.kube/config
   ```

   By default, cloudcore use https connection to Kubernetes apiserver. If `master` and `kubeConfig` are both set, `master` will override any value in kubeconfig.

2. Before KubeEdge v1.3: check whether the cert files for `modules.cloudhub.tlsCAFile`, `modules.cloudhub.tlsCertFile`,`modules.cloudhub.tlsPrivateKeyFile` exists.

   From KubeEdge v1.3: just skip the above check. If you configure the CloudCore certificate manually, you must check if the path of certificate is right.

   Note: If your KubeEdge version is before the v1.3, then just skip the step 3.

3. Configure all the IP addresses of CloudCore which are exposed to the edge nodes(like floating IP) in the `advertiseAddress`, which will be added to SANs in cert of cloudcore.

   ```
   modules:
     cloudHub:
       advertiseAddress:
       - 10.1.11.85
   ```

7.1.2 Adding the edge nodes (KubeEdge Worker Node) on the Cloud side (KubeEdge Master)

Node registration can be completed in two ways:

1. Node - Automatic Registration
2. Node - Manual Registration

Node - Automatic Registration

Edge node can be registered automatically if the value of field `modules.edged.registerNode` in edgecore's *config* is set to true.

```
modules:
  edged:
    registerNode: true
```

Node - Manual Registration

Copy `$GOPATH/src/github.com/kubeedge/kubeedge/build/node.json` to your working directory and change `metadata.name` to the name of edge node

```
mkdir -p ~/kubeedge/yaml
cp $GOPATH/src/github.com/kubeedge/kubeedge/build/node.json ~/kubeedge/yaml
```

Node.json

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "edge-node",
    "labels": {
      "name": "edge-node",
      "node-role.kubernetes.io/edge": ""
    }
  }
}
```

Note:

1. the `metadata.name` must keep in line with edgecore's config `modules.edged.hostnameOverride`.
2. 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 `metadata.labels`. 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 edge node (you must run the command on cloud side)

```
kubectl apply -f ~/kubeedge/yaml/node.json
```

Check the existence of certificates (cloud side) (Required for pre 1.3 releases)

Note: From KubeEdge v1.3, just skip the follow steps of checking the existence of certificates. However, if you configure the cloudcore certificate manually, you must check if the path of certificate is right. And there is no need to transfer certificate file from the cloud side to edge side.

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.

cert/key should exist in /etc/kubeedge/ca and /etc/kubeedge/certs. Otherwise please refer to generate certs to generate them. You need to copy these files to the corresponding directory on edge side.

Create the `certs.tgz` by

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

Transfer certificate file from the cloud side to edge side

Transfer certificate files to the edge node, because `edgecore` uses these certificate files to connect to `cloudcore`

This can be done by utilising scp

```
cd /etc/kubeedge/
scp certs.tgz username@destination:/etc/kubeedge
```

Here, we are copying the certs.tgz from the cloud side to the edge node in the /etc/kubeedge directory. You may copy in any directory and then move the certs to /etc/kubeedge folder.

At this point we have completed all configuration changes related to cloudcore.

7.2 Configuration Edge side (KubeEdge Worker Node)

7.2.1 Manually copy certs.tgz from cloud host to edge host(s) (Required for pre 1.3 releases)

Note: From KubeEdge v1.3 just skip this step, the edgecore will apply for the certificate automatically from the cloudcore when starting. You can also configure the local certificate(The CA certificate in edge site must be the same with cloudcore now). Any directory is OK as long as you configure it in the edgecore.yaml below.

On edge host

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

On edge host untar the certs.tgz file

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

7.2.2 Create and set edgecore config file

Create the `/etc/kubeedge/config` folder

```
    # the default configration file path is '/etc/kubeedge/config/edgecore.yaml'
    # also you can specify it anywhere with '--config'
    mkdir -p /etc/kubeedge/config/
```

Either create a minimal configuration with command `~/kubeedge/edgecore --minconfig`

```
    ~/kubeedge/edgecore --minconfig > /etc/kubeedge/config/edgecore.yaml
```

or a full configuration with command `~/kubeedge/edgecore --defaultconfig`

```
~/kubeedge/edgecore --defaultconfig > /etc/kubeedge/config/edgecore.yaml
```

Edit the configuration file

```
    vim /etc/kubeedge/config/edgecore.yaml
```

Verify the configurations before running `edgecore`

Modification in edgecore.yaml

1. Check `modules.edged.podSandboxImage` : This is very important and must be set correctly.

   To check the architecture of your machine run the following

   ```
   getconf LONG_BIT
   ```

   - `kubeedge/pause-arm:3.1` for arm arch
   - `kubeedge/pause-arm64:3.1` for arm64 arch
   - `kubeedge/pause:3.1` for x86 arch

2. Before KubeEdge v1.3: check whether the cert files for `modules.edgehub.tlsCaFile` and `modules.edgehub.tlsCertFile` and `modules.edgehub.tlsPrivateKeyFile` exists. If those files not exist, you need to copy them from cloud side.

   From KubeEdge v1.3: just skip above check about cert files. However, if you configure the edgecore certificate manually, you must check if the path of certificate is right.

3. Update the IP address and port of the KubeEdge CloudCore in the `modules.edgehub.websocket.server` and `modules.edgehub.quic.server` field. You need set cloudcore ip address.

4. Configure the desired container runtime 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

   ```
   runtimeType: docker
   ```

   or

   ```
   runtimeType: remote
   ```

5. If your runtime-type is remote, follow this guide KubeEdge CRI Configuration to setup KubeEdge with the remote/CRI based runtimes.

   Note: If your KubeEdge version is before the v1.3, then just skip the steps 6-7.

6. Configure the IP address and port of the KubeEdge cloudcore in the `modules.edgehub.httpServer` which is used to apply for the certificate. For example:

   ```
   modules:
     edgeHub:
       httpServer: https://10.1.11.85:10002
   ```

7. Configure the token.

   ```
   kubectl get secret tokensecret -n kubeedge -oyaml
   ```

   Then you get it like this:

```
apiVersion: v1
data:
  tokendata:␣
↪ODEzNTZjY2MwODIzMmIxMTU0Y2ExYmI5MmRlZjY4YWQwMGQ3ZDcwOTIzYmU3YjcyZWZmOTVlMTdiZTk5MzdkNS5leUpoYk
kind: Secret
metadata:
  creationTimestamp: "2020-05-10T01:53:10Z"
  name: tokensecret
  namespace: kubeedge
  resourceVersion: "19124039"
  selfLink: /api/v1/namespaces/kubeedge/secrets/tokensecret
  uid: 48429ce1-2d5a-4f0e-9ff1-f0f1455a12b4
type: Opaque
```

Decode the tokendata field by base64:

```
echo␣
↪ODEzNTZjY2MwODIzMmIxMTU0Y2ExYmI5MmRlZjY4YWQwMGQ3ZDcwOTIzYmU3YjcyZWZmOTVlMTdiZTk5MzdkNS5leUpoYk
↪|base64 -d
# then we get:
81356ccc08232b1154ca1bb92def68ad00d7d70923be7b72eff95e17be9937d5.
↪eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1ODkxNjE5ODl9.
↪jq4CW6MV4yTVJU9gAS1j6DBtNjyXPOx18qyFq_9d8XY
```

Copy the decoded string to the edgecore.yaml just like follow:

```
modules:
  edgeHub:
    token: 81356ccc08232b1154ca1bb92def68ad00d7d70923be7b72eff95e17be9937d5.
↪eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1ODkxNjE5ODl9.
↪jq4CW6MV4yTVJU9gAS1j6DBtNjyXPOx18qyFq_9d8XY
```

Configuring MQTT mode

The Edge part of KubeEdge uses MQTT for communication between deviceTwin and devices. KubeEdge supports 3 MQTT modes (`internalMqttMode`, `bothMqttMode`, `externalMqttMode`), set `mqttMode` field in edgecore.yaml to the desired mode.

- internalMqttMode: internal mqtt broker is enabled (`mqttMode`=0).
- bothMqttMode: internal as well as external broker are enabled (`mqttMode`=1).
- externalMqttMode: only external broker is enabled (`mqttMode`=2).

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.

At this point we have completed all configuration changes related to edgecore.

CHAPTER 8

KubeEdge runtime configuration

8.1 containerd

Docker 18.09 and up ship with containerd, so you should not need to install it manually. If you do not have containerd, you may install it by running the following:

```
# Install containerd
apt-get update && apt-get install -y containerd.io

# Configure containerd
mkdir -p /etc/containerd
containerd config default > /etc/containerd/config.toml

# Restart containerd
systemctl restart containerd
```

When using containerd shipped with Docker, the cri plugin is disabled by default. You will need to update containerd's configuration to enable KubeEdge to use containerd as its runtime:

```
# Configure containerd
mkdir -p /etc/containerd
containerd config default > /etc/containerd/config.toml
```

Update the edgecore config file edgecore.yaml, specifying the following parameters for the containerd-based runtime:

```
remoteRuntimeEndpoint: unix:///var/run/containerd/containerd.sock
remoteImageEndpoint: unix:///var/run/containerd/containerd.sock
runtimeRequestTimeout: 2
podSandboxImage: k8s.gcr.io/pause:3.2
runtimeType: remote
```

By default, the cgroup driver of cri is configured as cgroupfs. If this is not the case, you can switch to systemd manually in edgecore.yaml:

```
modules:
  edged:
    cgroupDriver: systemd
```

Set `systemd_cgroup` to `true` in `containerd`'s configuration file (/etc/containerd/config.toml), and then restart `containerd`:

```
# /etc/containerd/config.toml
systemd_cgroup = true
```

```
# Restart containerd
systemctl restart containerd
```

Create the `nginx` application and check that the container is created with `containerd` on the edge side:

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

ctr --namespace=k8s.io container ls
CONTAINER                                                           IMAGE              ␣
↪                   RUNTIME
41c1a07fe7bf7425094a9b3be285c312127961c158f30fc308fd6a3b7376eab2    docker.io/library/
↪nginx:1.15.12    io.containerd.runtime.v1.linux
```

NOTE: since cri doesn't support multi-tenancy while `containerd` does, the namespace for containers are set to "k8s.io" by default. There is not a way to change that until support in cri has been implemented.

8.2 CRI-O

Follow the CRI-O install guide to setup CRI-O.

If your edge node is running on the ARM platform and your distro is ubuntu18.04, you might need to build the binaries from source and then install, since CRI-O packages are not available in the Kubic repository for this combination.

```
git clone https://github.com/cri-o/cri-o
cd cri-o
make
sudo make install
# generate and install configuration files
sudo make install.config
```

Set up CNI networking by following this guide: setup CNI. Update the edgecore config file, specifying the following parameters for the `CRI-O`-based runtime:

```
remoteRuntimeEndpoint: unix:///var/run/crio/crio.sock
remoteImageEndpoint: unix:////var/run/crio/crio.sock
runtimeRequestTimeout: 2
podSandboxImage: k8s.gcr.io/pause:3.2
runtimeType: remote
```

By default, `CRI-O` uses `cgroupfs` as a cgroup driver manager. If you want to switch to `systemd` instead, update the `CRI-O` config file (/etc/crio/crio.conf.d/00-default.conf):

```
# Cgroup management implementation used for the runtime.
cgroup_manager = "systemd"
```

NOTE: the `pause` *image should be updated if you are on ARM platform and the* `pause` *image you are using is not a multi-arch image. To set the pause image, update the* `CRI-O` *config file:*

```
pause_image = "k8s.gcr.io/pause-arm64:3.1"
```

Remember to update `edgecore.yaml` as well for your cgroup driver manager:

```
modules:
  edged:
    cgroupDriver: systemd
```

Start `CRI-O` and `edgecore` services (assume both services are taken care of by `systemd`),

```
sudo systemctl daemon-reload
sudo systemctl enable crio
sudo systemctl start crio
sudo systemctl start edgecore
```

Create the application and check that the container is created with `CRI-O` on the edge side:

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

# crictl ps
CONTAINER ID        IMAGE               CREATED             STATE               NAME ␣
↪               ATTEMPT             POD ID
41c1a07fe7bf7       f6d22dec9931b       2 days ago          Running             nginx␣
↪               0                   51f727498b06f
```

8.3 Kata Containers

Kata Containers is a container runtime created to address security challenges in the multi-tenant, untrusted cloud environment. However, multi-tenancy support is still in KubeEdge's backlog. If you have a downstream customized KubeEdge which supports multi-tenancy already then Kata Containers is a good option for a lightweight and secure container runtime.

Follow the install guide to install and configure containerd and Kata Containers.

If you have "kata-runtime" installed, run this command to check if your host system can run and create a Kata Container:

```
kata-runtime kata-check
```

`RuntimeClass` is a feature for selecting the container runtime configuration to use to run a pod's containers that is supported since `containerd` v1.2.0. If your `containerd` version is later than v1.2.0, you have two choices to configure `containerd` to use Kata Containers:

- Kata Containers as a RuntimeClass
- Kata Containers as a runtime for untrusted workloads

Suppose you have configured Kata Containers as the runtime for untrusted workloads. In order to verify whether it works on your edge node, you can run:

```
cat nginx-untrusted.yaml
apiVersion: v1
```

(continues on next page)

(continued from previous page)

```
kind: Pod
metadata:
  name: nginx-untrusted
  annotations:
    io.kubernetes.cri.untrusted-workload: "true"
spec:
  containers:
  - name: nginx
    image: nginx
```

```
kubectl create -f nginx-untrusted.yaml

# verify the container is running with qemu hypervisor on edge side,
ps aux | grep qemu
root       3941  3.0  1.0 2971576 174648 ?      Sl   17:38   0:02 /usr/bin/qemu-system-
↪aarch64

crictl pods
POD ID              CREATED             STATE               NAME                  ␣
↪NAMESPACE           ATTEMPT
b1c0911644cb9       About a minute ago  Ready               nginx-untrusted       ␣
↪default             0
```

8.4 Virtlet

Make sure no libvirt is running on the worker nodes.

8.4.1 Steps

1. **Install CNI plugin:**

   Download CNI plugin release and extract it:

```
$ wget https://github.com/containernetworking/plugins/releases/download/v0.8.2/
↪cni-plugins-linux-amd64-v0.8.2.tgz

# Extract the tarball
$ mkdir cni
$ tar -zxvf v0.2.0.tar.gz -C cni

$ mkdir -p /opt/cni/bin
$ cp ./cni/* /opt/cni/bin/
```

   Configure CNI plugin:

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

$ cat >/etc/cni/net.d/bridge.conf <<EOF
{
  "cniVersion": "0.3.1",
  "name": "containerd-net",
  "type": "bridge",
  "bridge": "cni0",
```

(continues on next page)

(continued from previous page)

```
  "isGateway": true,
  "ipMasq": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.88.0.0/16",
    "routes": [
      { "dst": "0.0.0.0/0" }
    ]
  }
}
EOF
```

2. **Setup VM runtime:** Use the script `hack/setup-vmruntime.sh` to set up a VM runtime. It makes use of the Arktos Runtime release to start three containers:

```
vmruntime_vms
vmruntime_libvirt
vmruntime_virtlet
```

CHAPTER 9

KubeEdge Volume Support

Consider use case at edge side, we only support following volume types, all of those are same as Kubernetes:

If you want to want more volume types support, please file an issue and comment use case, we will support it if necessary.

CHAPTER 10

Beehive

10.1 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

10.2 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{})

10.3 Register Module

1. On starting edgecore, 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.

10.4 Channel Context Structure Fields

10.4.1 (*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 be 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

10.5 Module Operations

10.5.1 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")
```

10.5.2 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”)
```

10.5.3 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”)
```

10.6 Message Operations

10.6.1 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)
```

10.6.2 Send to a Group

1. SendToGroup 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.SendToGroup(“edged”,message) message will be sent to all modules in edged␣
↪group.
```

10.6.3 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")
```

10.6.4 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)
```

10.6.5 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.SendToGroupSync("edged",message,60*time.Second)
```

10.6.6 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 anon-Channels.
4. If channel exists, message(response) is sent on that channel.
5. Or else error is logged.

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

CHAPTER 11

Edge Controller

11.1 Edge Controller Overview

EdgeController is the bridge between Kubernetes Api-Server and edgecore

11.2 Operations Performed By Edge Controller

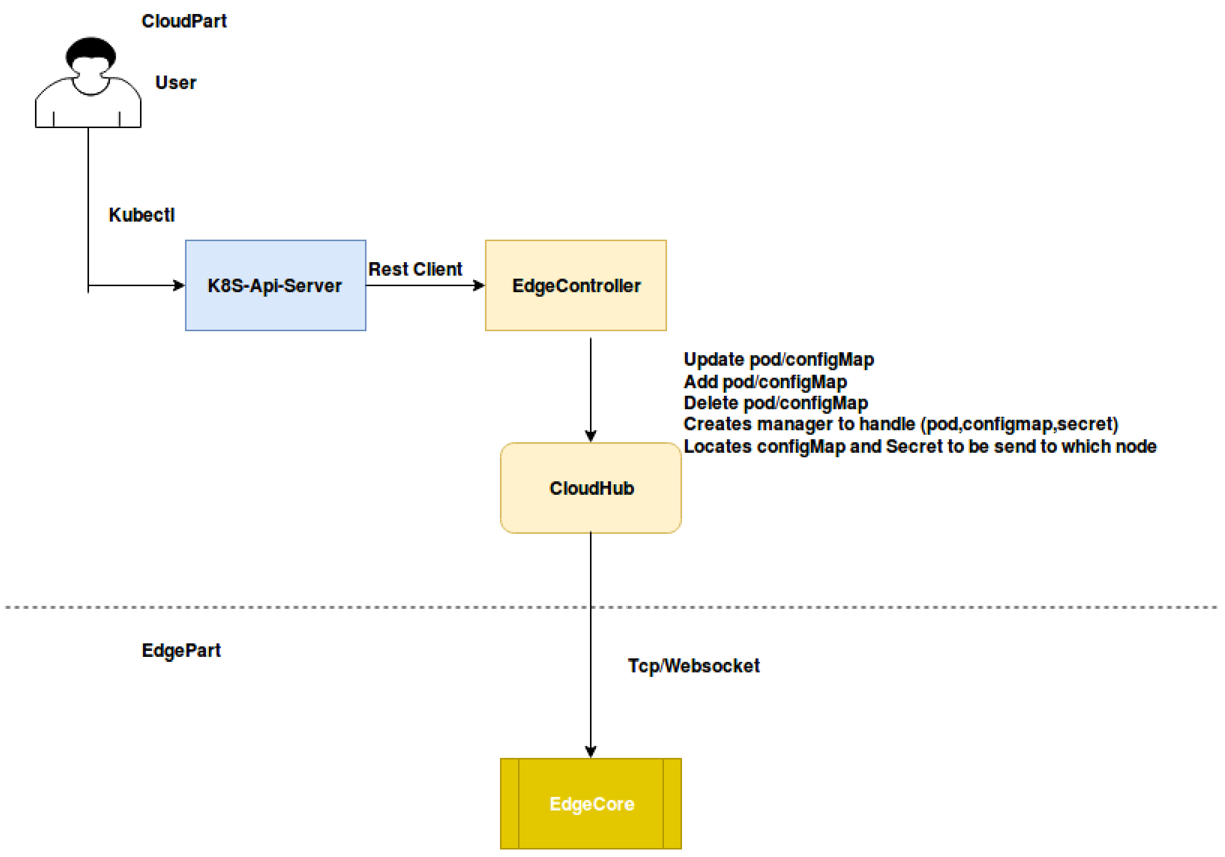
The following are the functions performed by Edge controller :-

- 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

11.3 Downstream Controller:

11.3.1 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



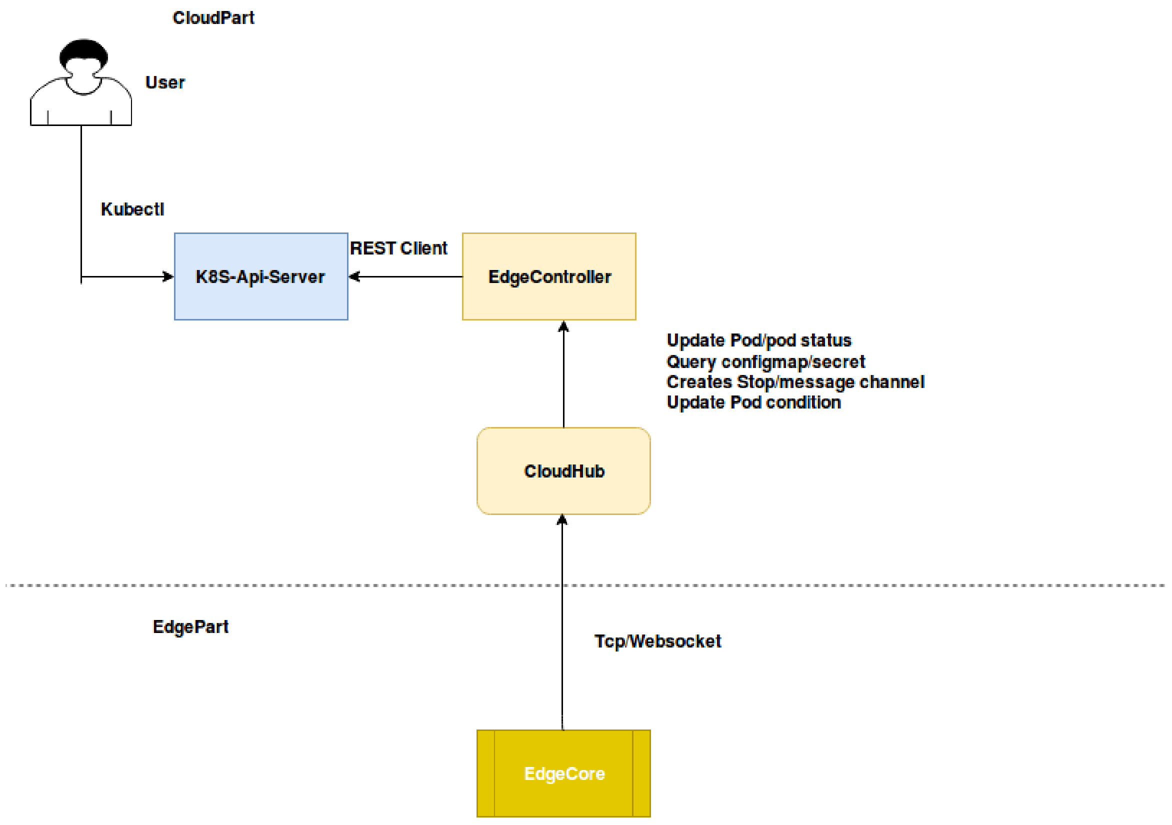
Downstream Controller

11.4 Upstream Controller:

11.4.1 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 successfully
  - **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

11.5 Controller Manager:

11.5.1 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 12

CloudHub

12.1 CloudHub Overview

CloudHub is one module of cloudcore and is the mediator between Controllers and the Edge side. It supports both websocket 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 Controllers.

The connection to the edge(through EdgeHub module) is done through the HTTP over websocket connection. For internal communication it directly communicates with the Controllers. All the requests sent 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

12.1.1 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.

12.1.2 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 serves all the incoming connections

12.1.3 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 Controllers

12.1.4 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

12.1.5 Publish Message to Controllers:

- 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.

12.2 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 13

Device Controller

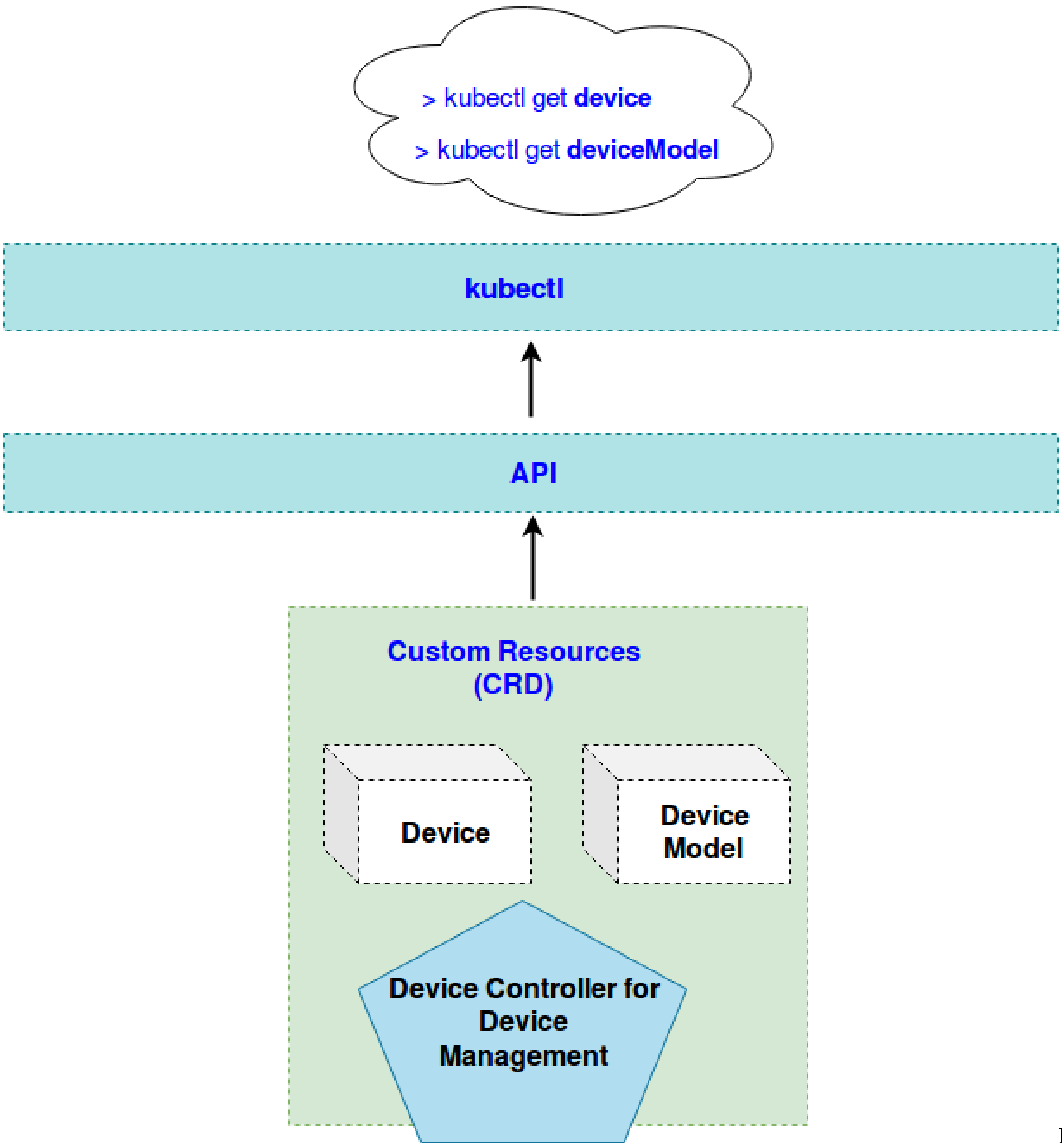
13.1 Device Controller Overview

The device controller is the cloud component of KubeEdge which is responsible for device management. Device management in 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

13.2 Operations Performed By Device Controller

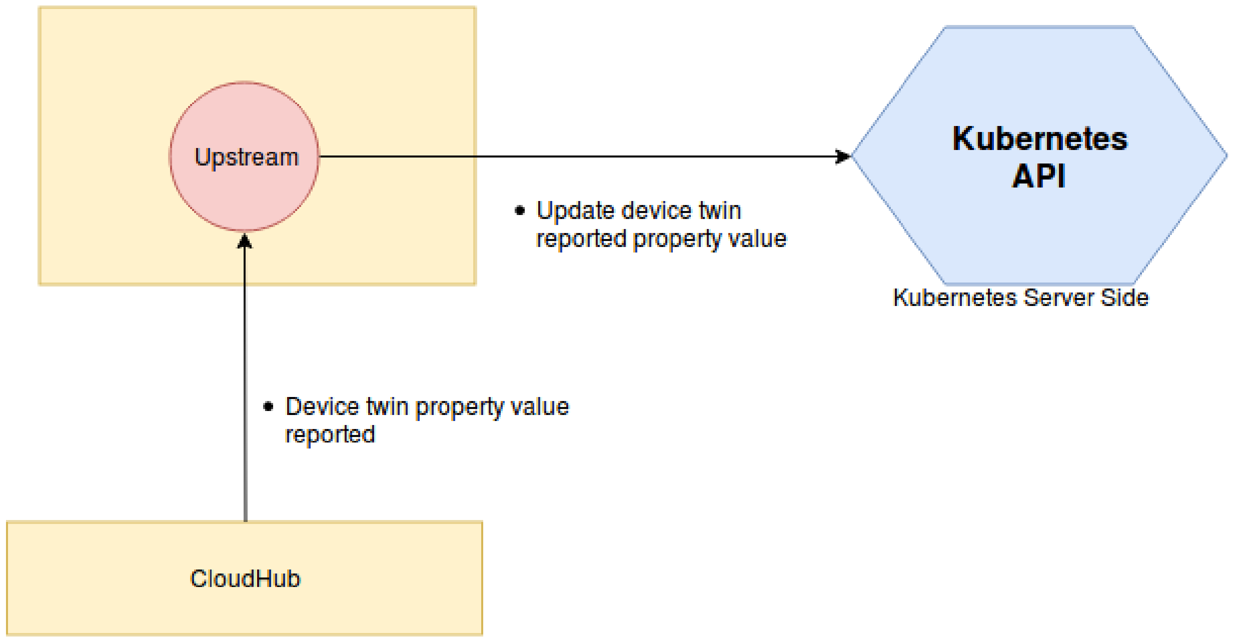
The following are the functions performed by the device controller :-

- **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

13.3 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:

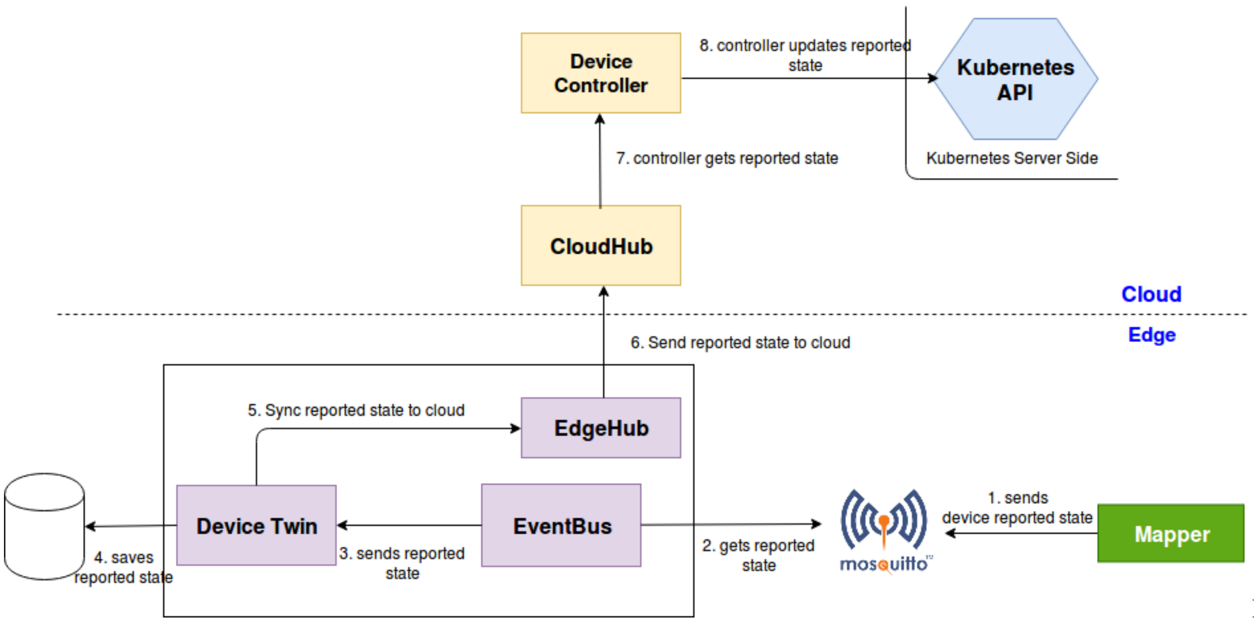

Upstream
Kubernetes API
Update device twin reported property value
Kubernetes Server Side
Device twin property value reported
CloudHub

Device

Upstream Controller

13.3.1 Syncing Reported Device Twin Property Update From Edge To Cloud

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.

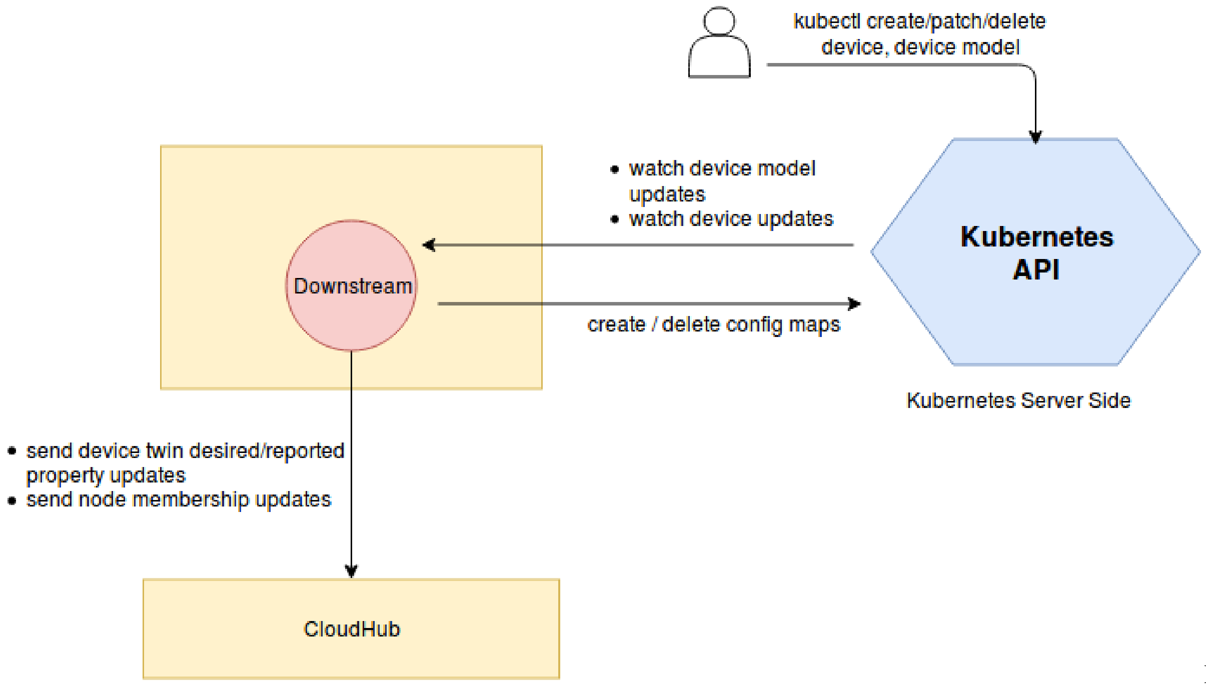


Device

Updates Edge To Cloud

13.4 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:

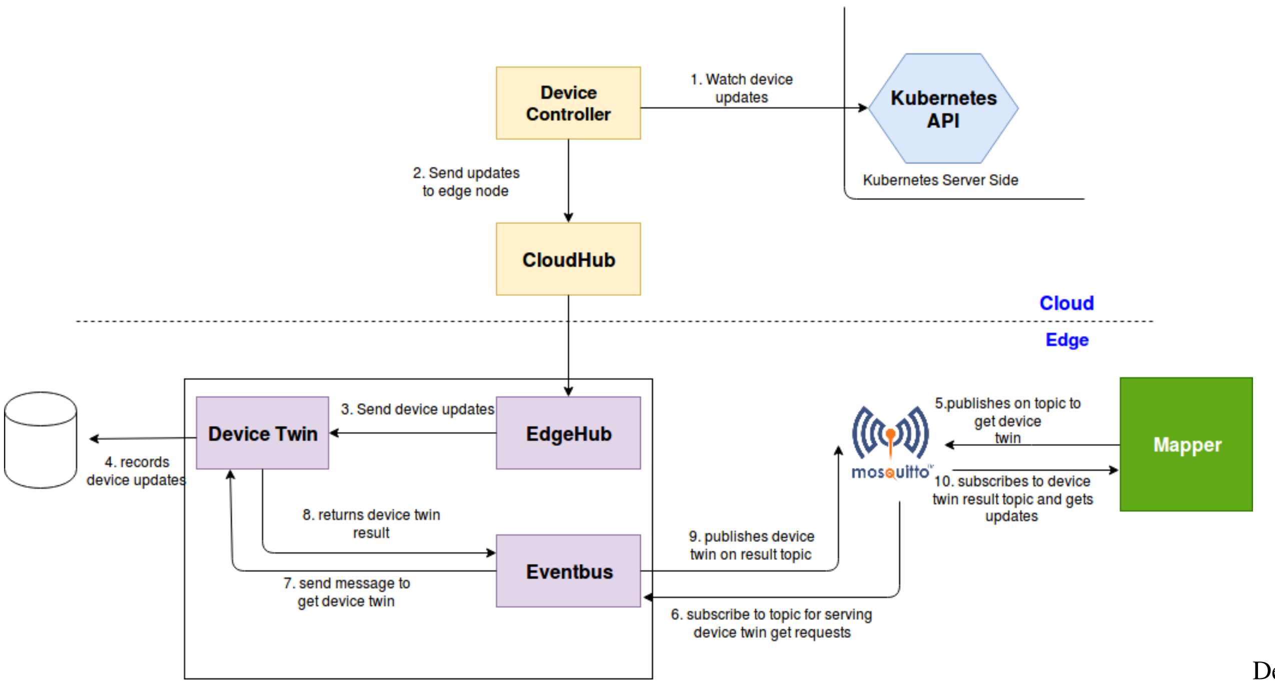

kubectl create/patch/delete device, device model
watch device model updates
watch device updates
Kubernetes API
Downstream
create / delete config maps
Kubernetes Server Side
send device twin desired/reported property updates
send node membership updates
CloudHub

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 instance. 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.

13.4.1 Syncing Desired Device Twin Property Update From Cloud To Edge


Device Controller
1. Watch device updates
Kubernetes API
Kubernetes Server Side
2. Send updates to edge node
CloudHub
Cloud
Edge
Device Twin
3. Send device updates
EdgeHub
4. records device updates
8. returns device twin result
Eventbus
7. send message to get device twin
9. publishes device twin on result topic
6. subscribe to topic for serving device twin get requests
mosquitto
5.publishes on topic to get device twin
10. subscribes to device twin result topic and gets updates
Mapper

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 14

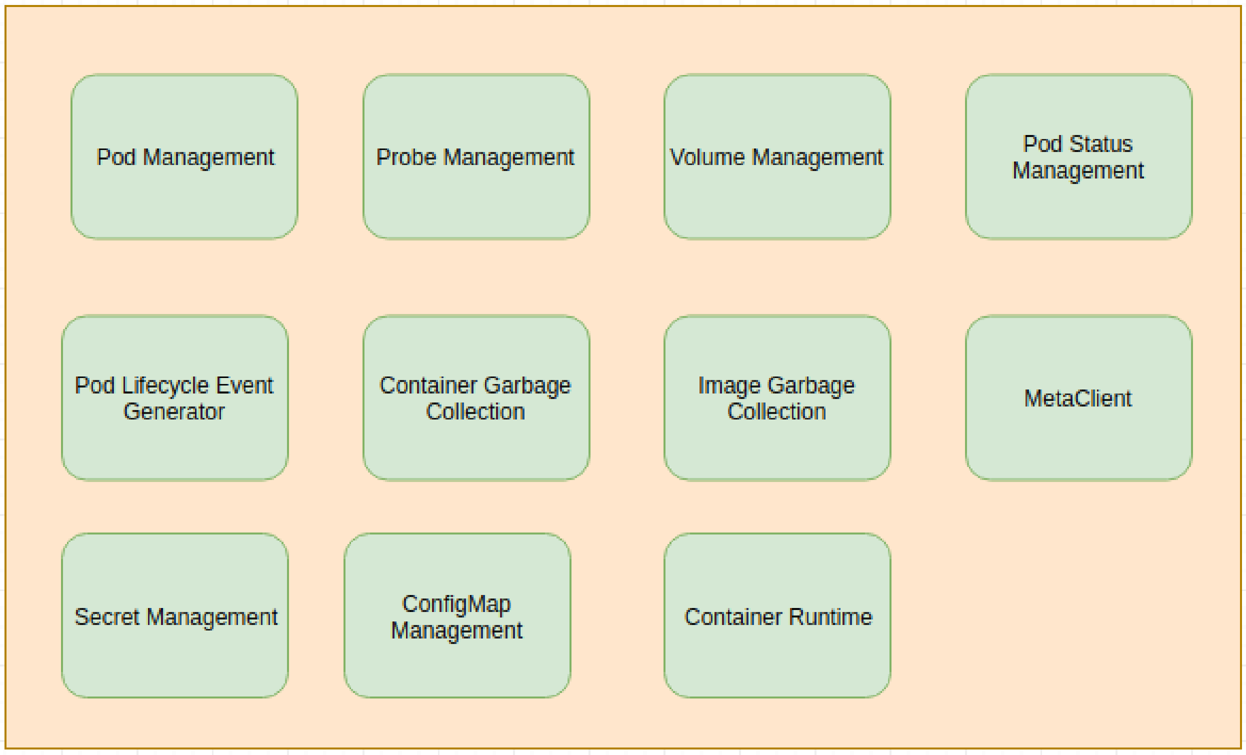
Edged

14.1 Overview

EdgeD is an edge node module which manages pod lifecycle. It helps users 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, users can issue commands to launch the workloads.

Several OCI-compliant runtimes are supported through the Container Runtime Interface (CRI). See *KubeEdge runtime configuration* for more information on how to configure edged to make use of other runtimes.

There are many modules which work in tandem to achieve edged's functionalities.



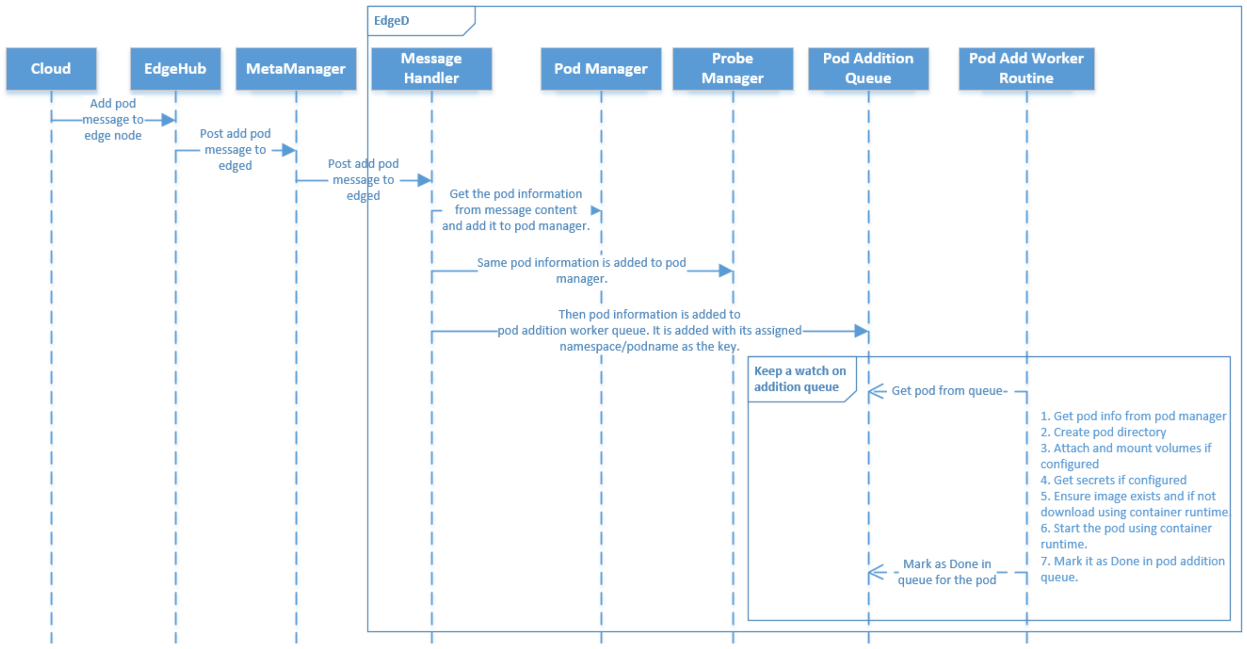
EdgeD Overall

Fig 1: EdgeD Functionalities

14.2 Pod Management

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

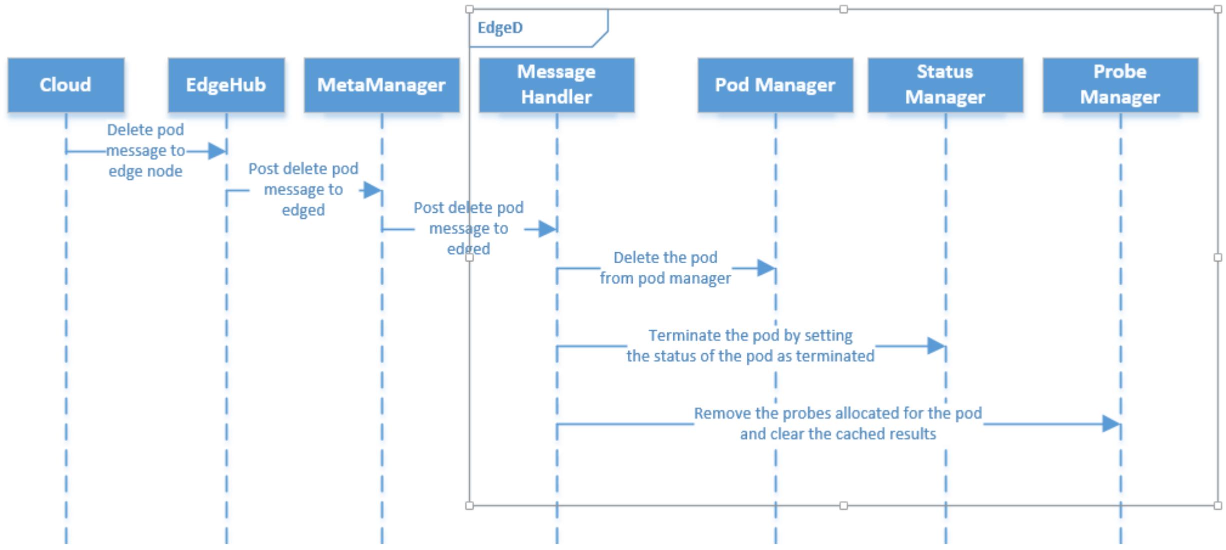
- Receives and handles pod addition/deletion/modification messages from metamanager.
- Handles separate worker queues for pod addition and deletion.
- Handles worker routines to check worker queues to do pod operations.
- Keeps separate cache for config map and secrets respectively.
- Regular cleanup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


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

Pod

Addition Flow

Fig 2: Pod Addition Flow


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

Pod

Deletion Flow

Fig 3: Pod Deletion Flow

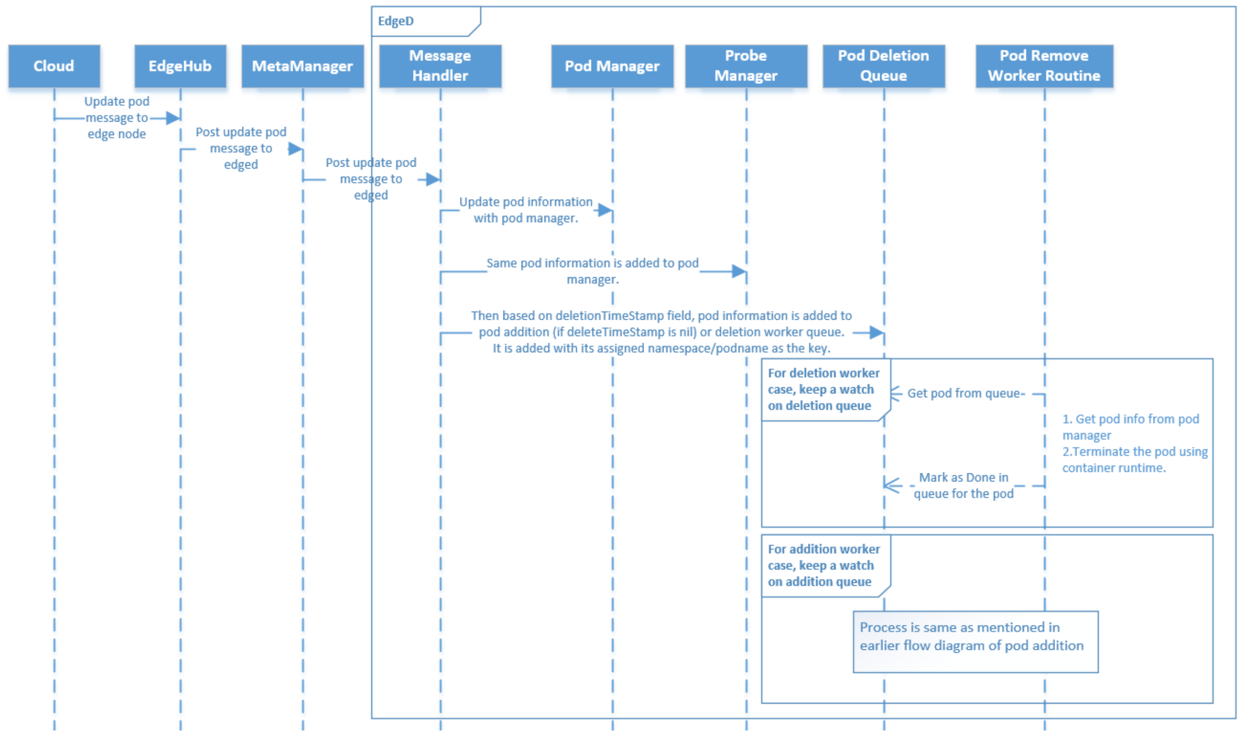

EdgeD
Cloud
EdgeHub
MetaManager
Message Handler
Pod Manager
Probe Manager
Pod Deletion Queue
Pod Remove Worker Routine
Update pod message to edge node
Post update pod message to edged
Post update pod message to edged
Update pod information with pod manager.
Same pod information is added to pod manager.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

Fig 4: Pod Updation Flow

14.3 Pod Lifecycle Event Generator

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

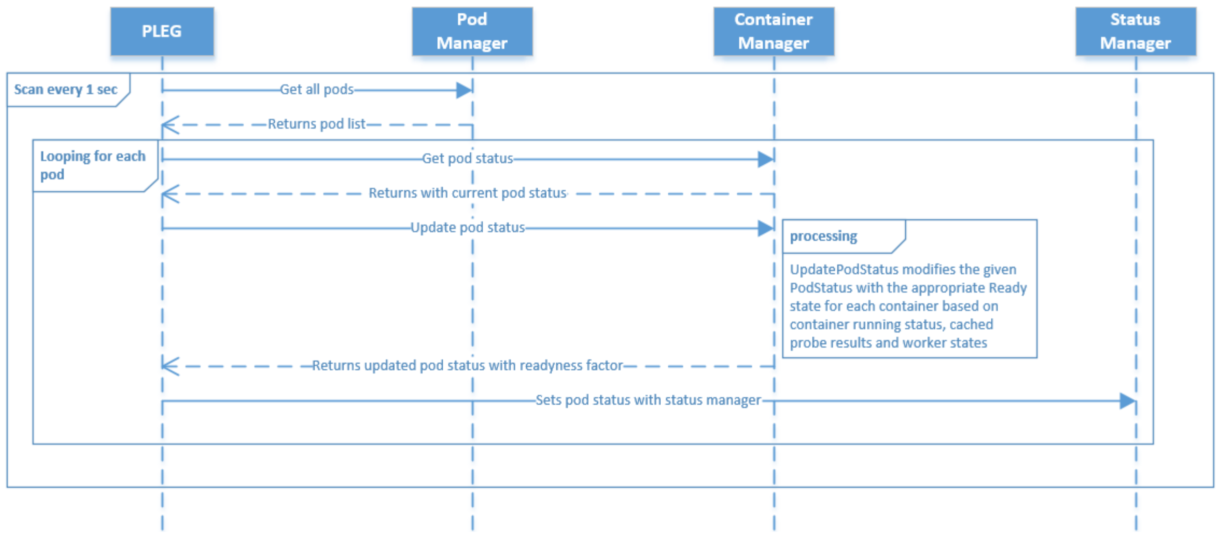

PLEG
Pod Manager
Container Manager
Status Manager
Scan every 1 sec
Get all pods
Returns pod list
Looping for each pod
Get pod status
Returns with current pod status
Update pod status
processing
UpdatePodStatus modifies the given PodStatus with the appropriate Ready state for each container based on container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

Fig 5: PLEG at EdgeD

14.4 CRI for edged

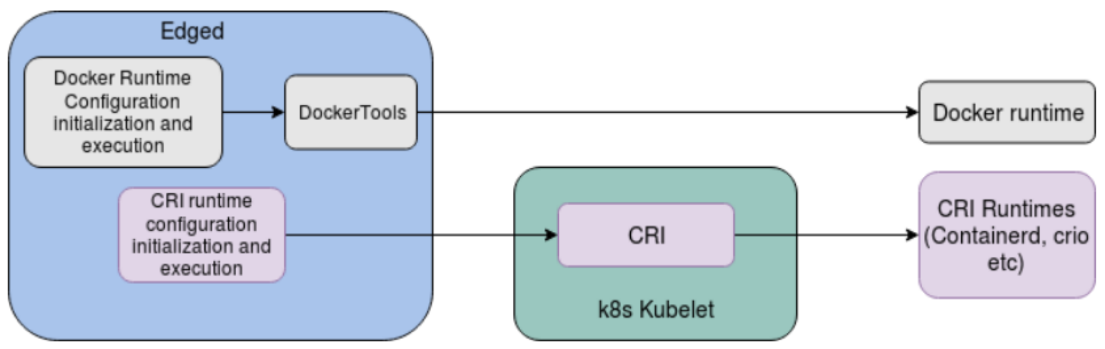
Container Runtime Interface (CRI) – a plugin interface which enables edged to use a wide variety of container runtimes like Docker, containerd, CRI-O, etc., without the need to recompile. For more on how to configure KubeEdge for container runtimes, see *KubeEdge runtime configuration*.

14.4.1 Why CRI for edged?

CRI support for multiple container runtimes in edged is needed in order to:

- Support light-weight container runtimes on resource-constrained edge nodes which are unable to run the existing Docker runtime.
- Support multiple container runtimes like Docker, containerd, CRI-O, etc., on edge nodes.

Support for corresponding CNI with pause container and IP will be considered later.


Edged
Docker Runtime Configuration initialization and execution
DockerTools
Docker runtime
CRI runtime configuration initialization and execution
CRI
k8s Kubelet
CRI Runtimes (Containerd, crio etc)

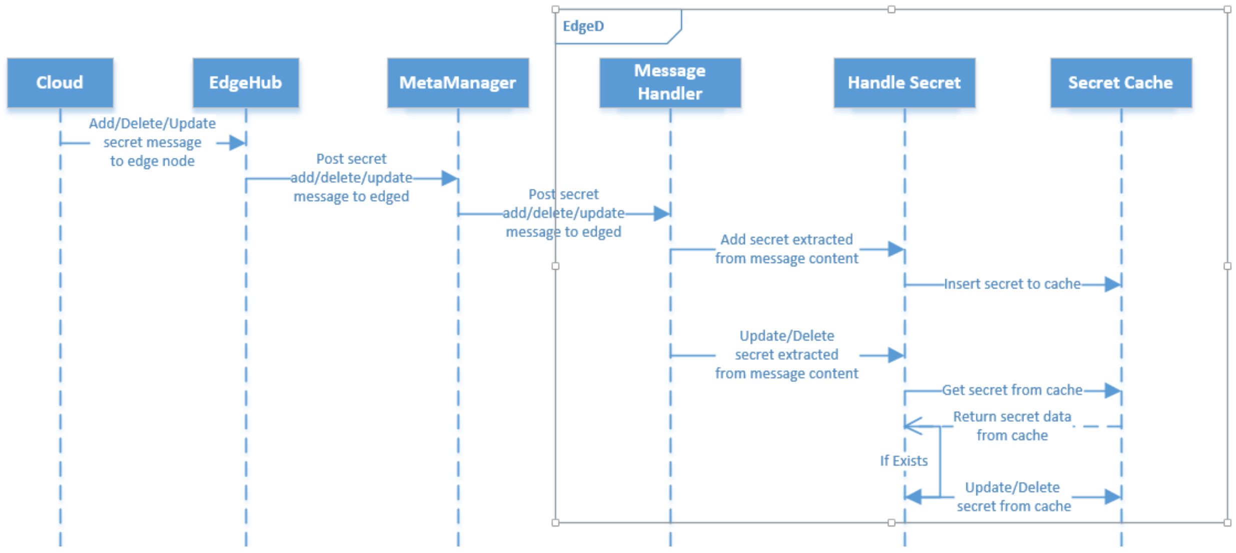
CRI

Design

Fig 6: CRI at EdgeD

14.5 Secret Management

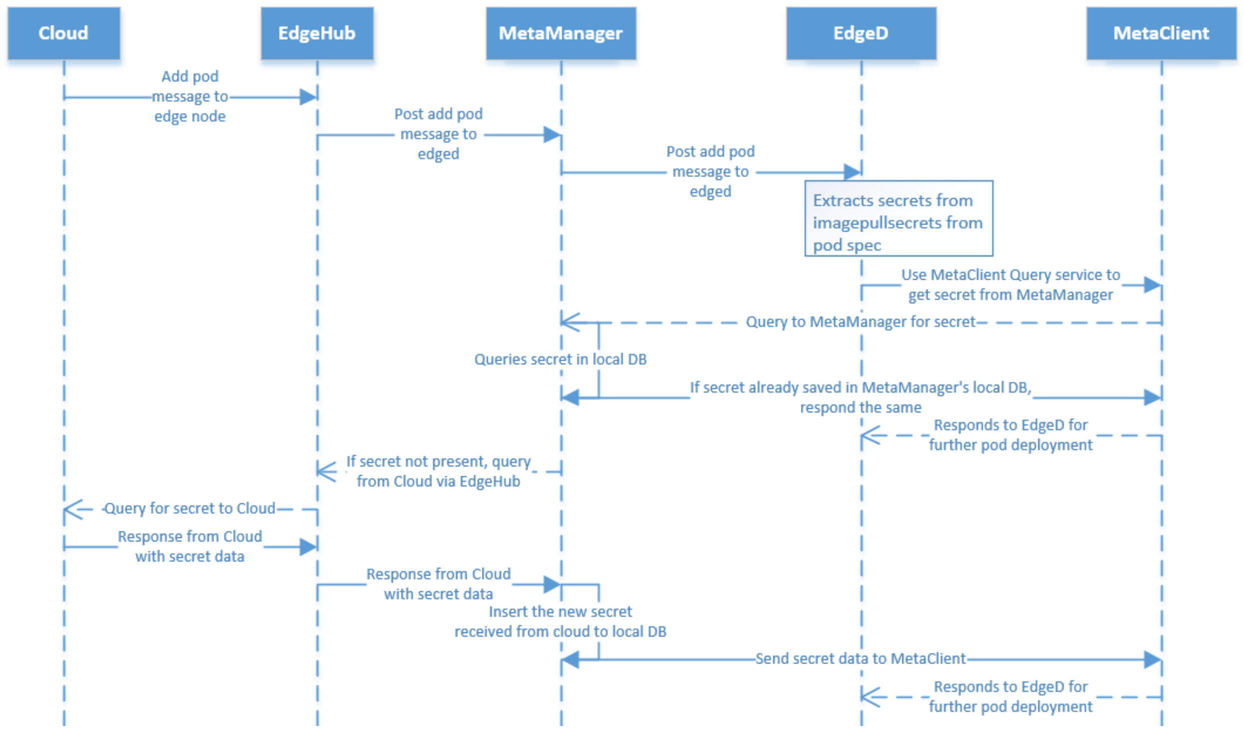
In edged, Secrets are handled separately. For operations like addition, deletion and modification, there are separate sets of config messages and interfaces. Using these interfaces, secrets are updated in cache store. The flow diagram below explains the message flow.


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

Secret Message Handling

Fig 7: Secret Message Handling at EdgeD

Edged uses the MetaClient module to fetch secrets from MetaManager. If edged queries for a new secret which is not yet stored in MetaManager, the request is forwarded to the Cloud. Before sending the response containing the secret, MetaManager stores it in a local database. Subsequent queries for the same secret key will be retrieved from the database, reducing latency. The flow diagram below shows how a secret is fetched from MetaManager and the Cloud. It also describes how the secret is stored in MetaManager.


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

Query Secret

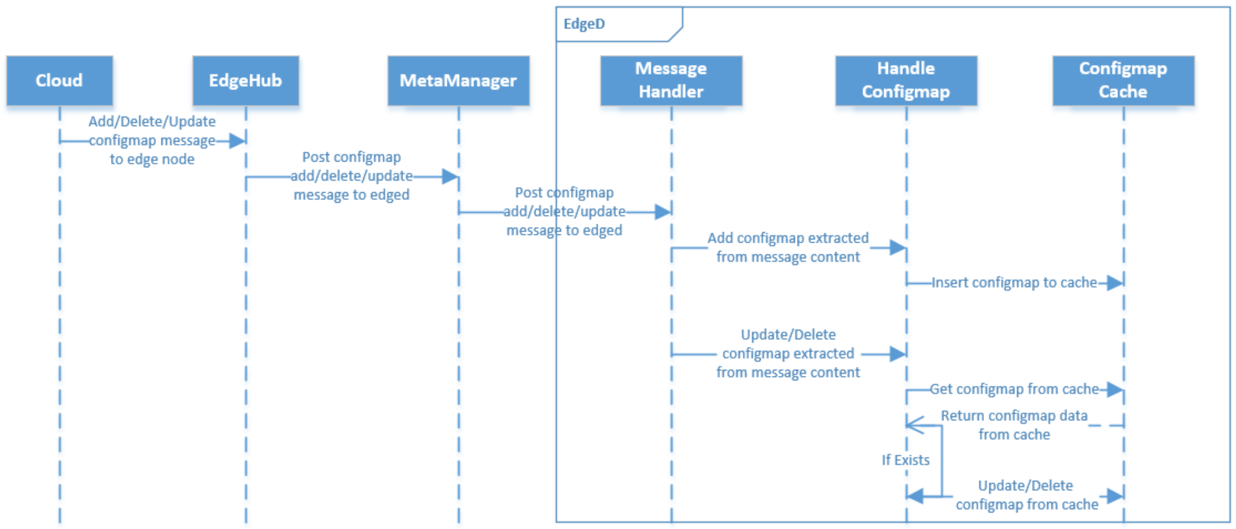
Fig 8: Query Secret by EdgeD

14.6 Probe Management

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

14.7 ConfigMap Management

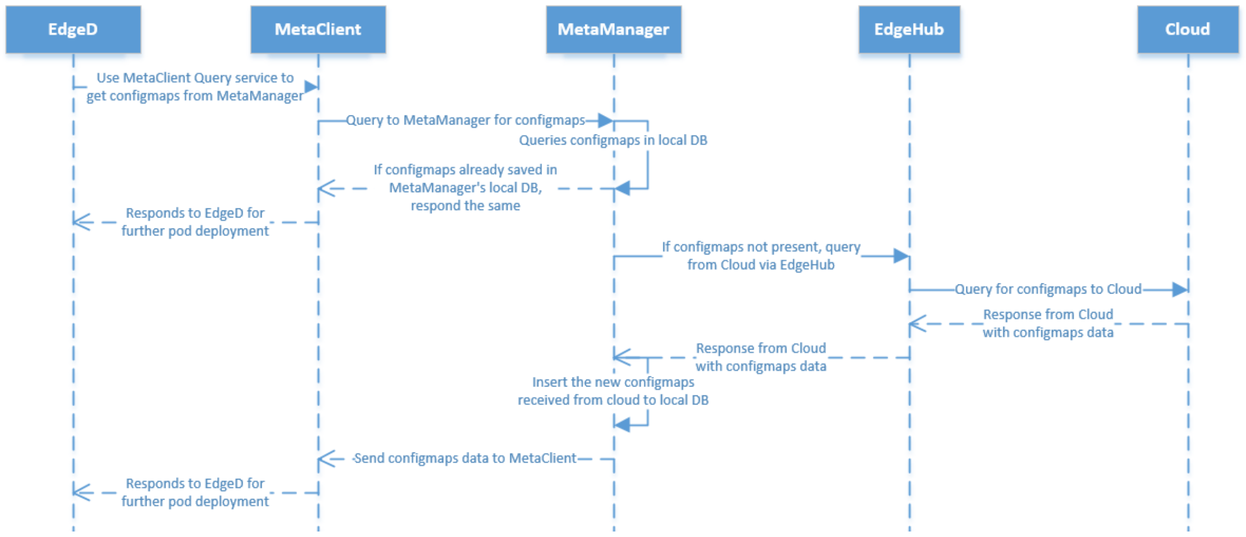
In edged, ConfigMaps are also handled separately. For operations like addition, deletion and modification, there are separate sets of config messages and interfaces. Using these interfaces, ConfigMaps are updated in cache store. The flow diagram below explains the message flow.


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

ConfigMap Message Handling

Fig 9: ConfigMap Message Handling at EdgeD

Edged uses the MetaClient module to fetch ConfigMaps from MetaManager. If edged queries for a new ConfigMap which is not yet stored in MetaManager, the request is forwarded to the Cloud. Before sending the response containing the ConfigMap, MetaManager stores it in a local database. Subsequent queries for the same ConfigMap key will be retrieved from the database, reducing latency. The flow diagram below shows how ConfigMaps are fetched from MetaManager and the Cloud. It also describes how ConfigMaps are stored in MetaManager.


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

Query Configmaps

Fig 10: Query Configmaps by EdgeD

14.8 Container GC

The 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 is determined by three variables, which can be user-defined:

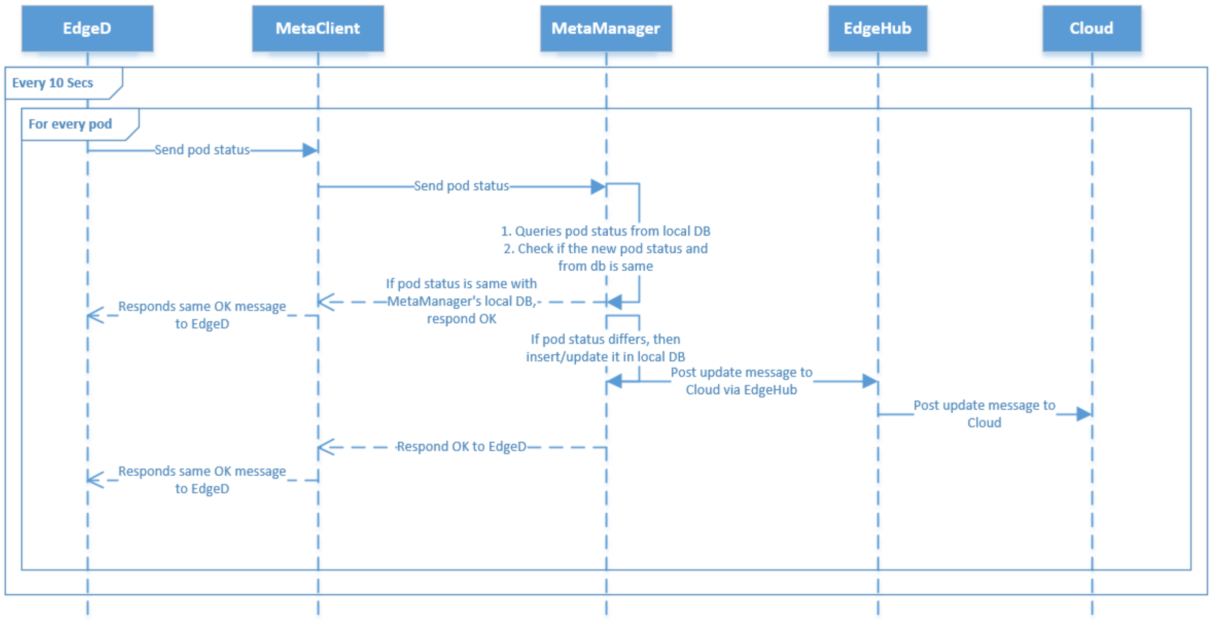
- `MinAge` is the minimum age at which a container can be garbage collected, zero for no limit.
- `MaxPerPodContainer` is the maximum number of dead containers that any single pod (UID, container name) pair is allowed to have, less than zero for no limit.
- `MaxContainers` is the maximum number of total dead containers, less than zero for no limit. Generally, the oldest containers are removed first.

14.9 Image GC

The image garbage collector is an edged routine which wakes up every 5 secs, and collects information about disk usage based on the policy used. The policy for garbage collecting images 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.

14.10 Status Manager

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


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

Status Manager Flow

Fig 11: Status Manager Flow

14.11 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 its other operations.

14.12 MetaClient

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

CHAPTER 15

EventBus

15.1 Overview

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

It supports 3 kinds of mode:

- internalMqttMode
- externalMqttMode
- bothMqttMode

15.2 Topic

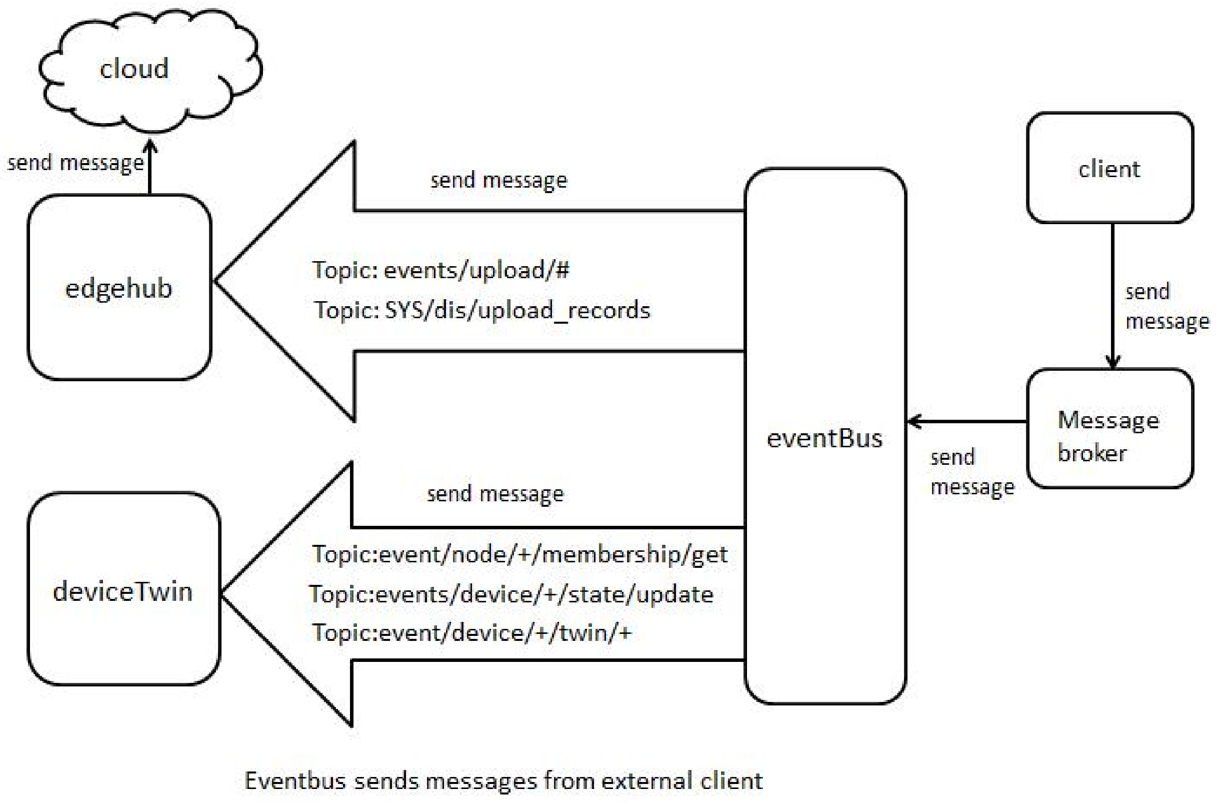
eventbus subscribes to the following topics:

```
- $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

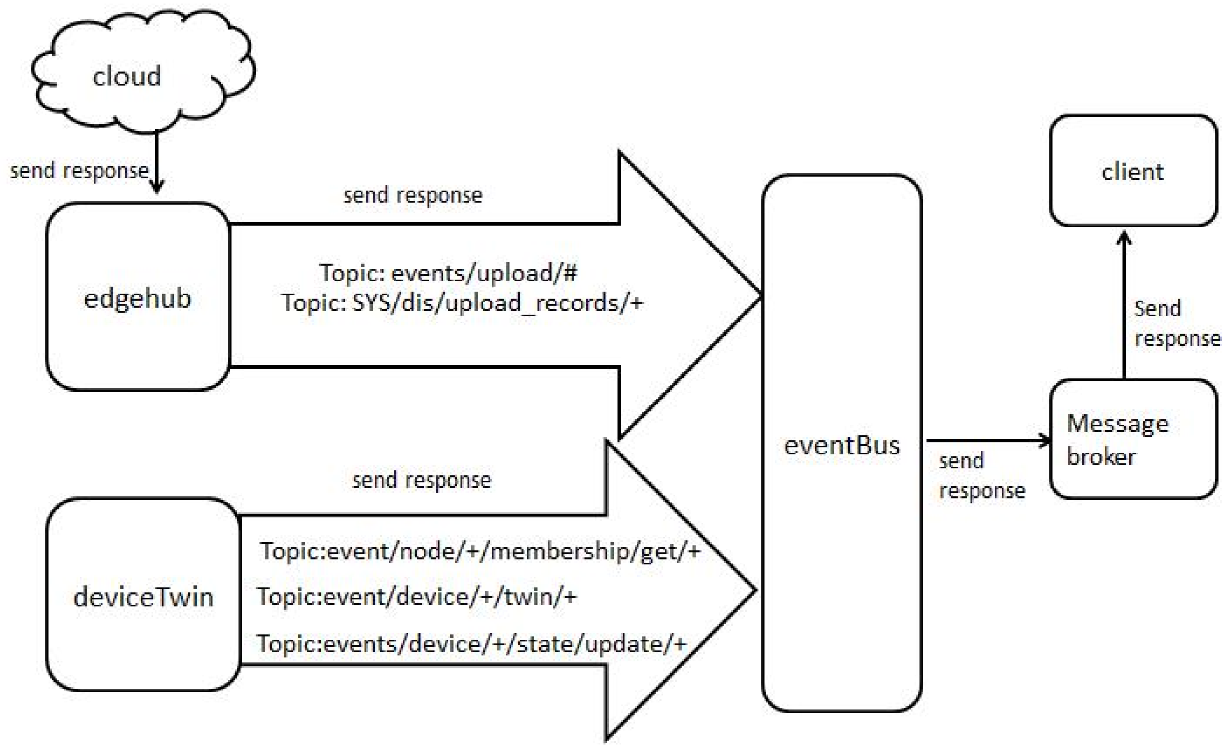
15.3 Flow chart

15.3.1 1. eventbus sends messages from external client


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

eventbus sends messages from external client

15.3.2 2. eventbus sends response messages to external client



Eventbus sends response messages to external client

eventbus sends response messages to external client

The flow is almost the same in internal mode except the eventbus is as message broker itself.

CHAPTER 16

MetaManager

16.1 Overview

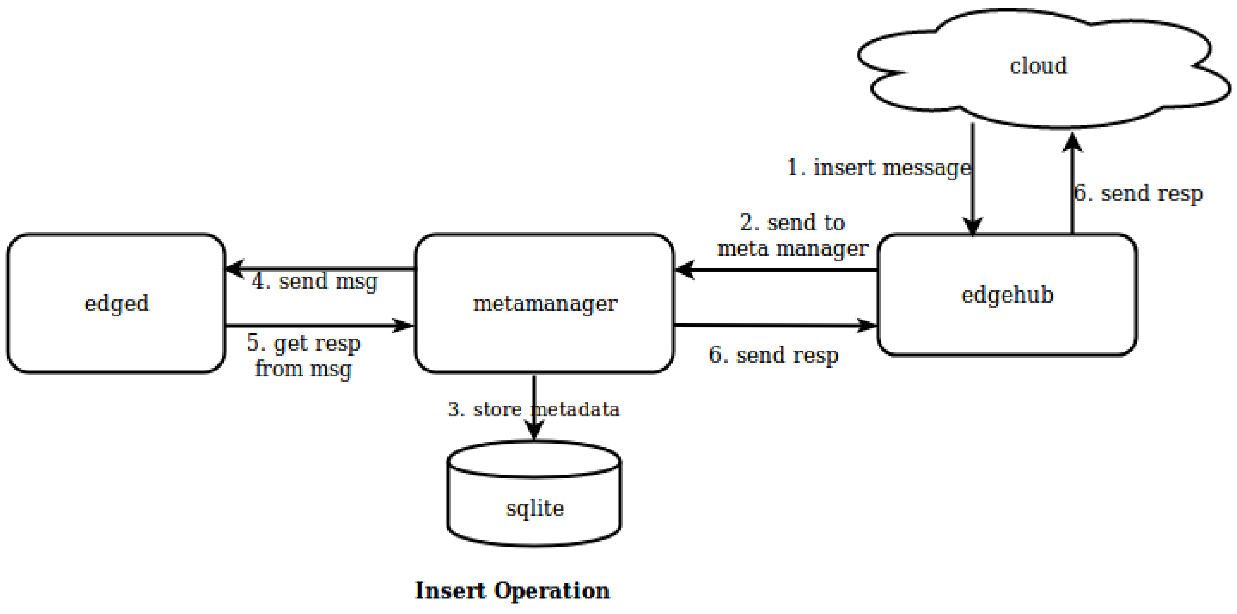
MetaManager is the message processor between edged and edgehub. It's also responsible for storing/retrieving metadata to/from a lightweight database(SQLite).

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

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

16.2 Insert Operation

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



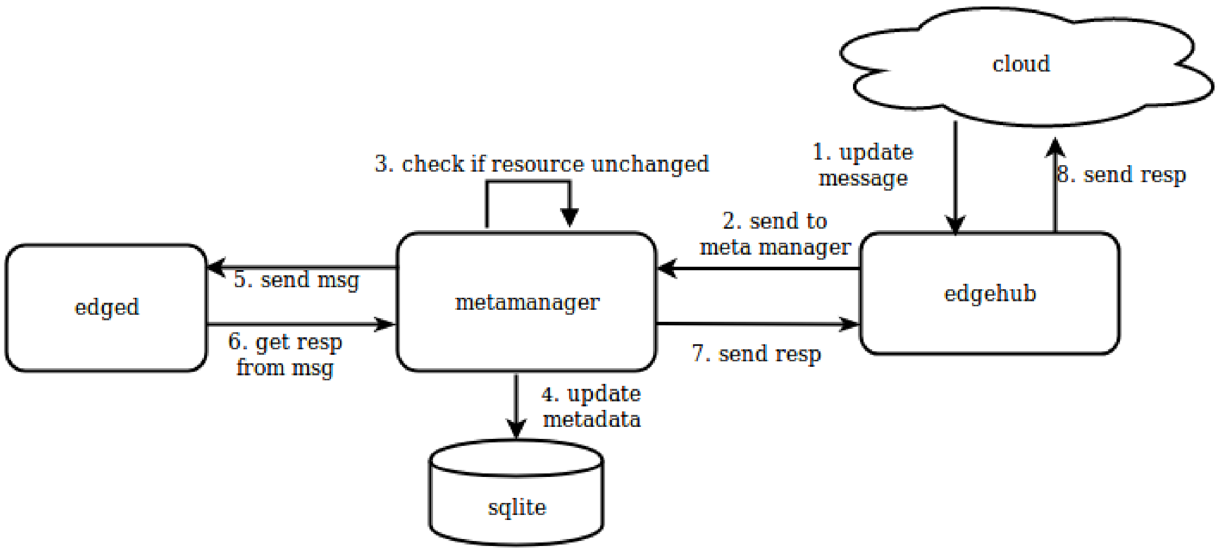
Insert Operation

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

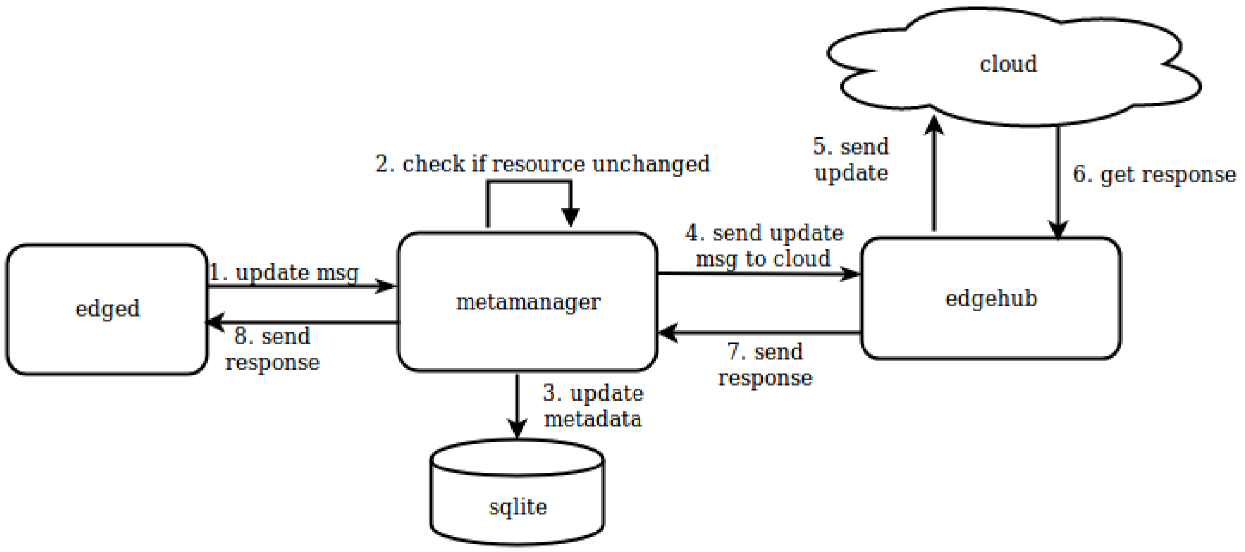
16.3 Update Operation

`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

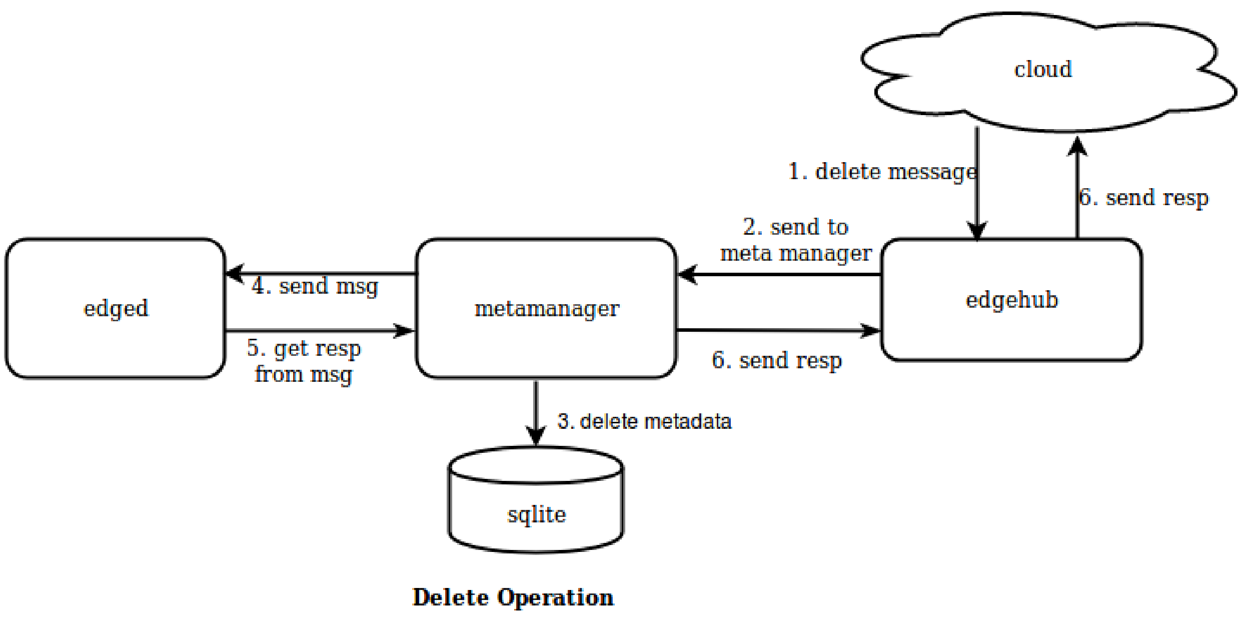


Update From Edge To Cloud

Update Operation

16.4 Delete Operation

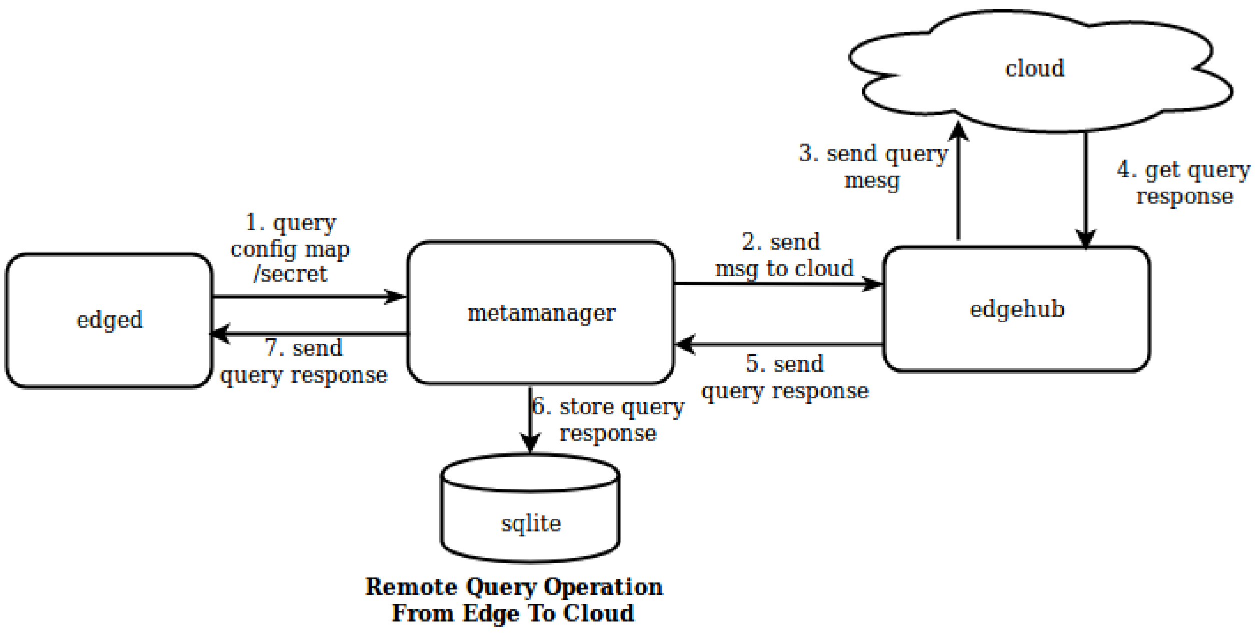
`Delete` operations are triggered when objects like pods are deleted from the cloud.

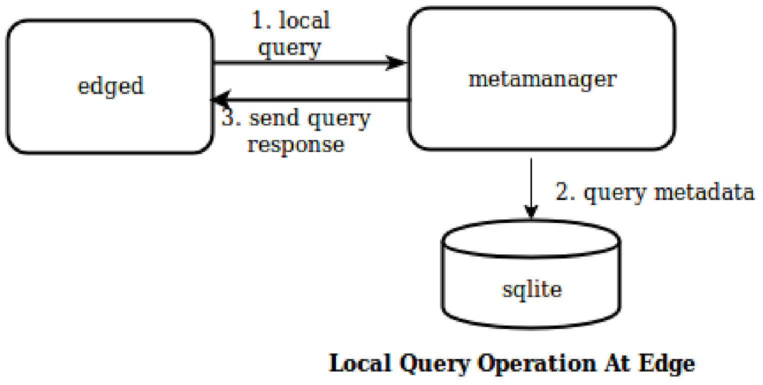


Delete Operation

16.5 Query Operation

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





Query Operation

16.6 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.

16.7 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.

16.8 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 17

EdgeHub

17.1 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 :-

- Keep Alive
- Publish Client Info
- Route to Cloud
- Route to Edge

17.2 Keep Alive

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

17.3 Publish Client Info

- The main responsibility of publishing 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.

17.4 Route To Cloud

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

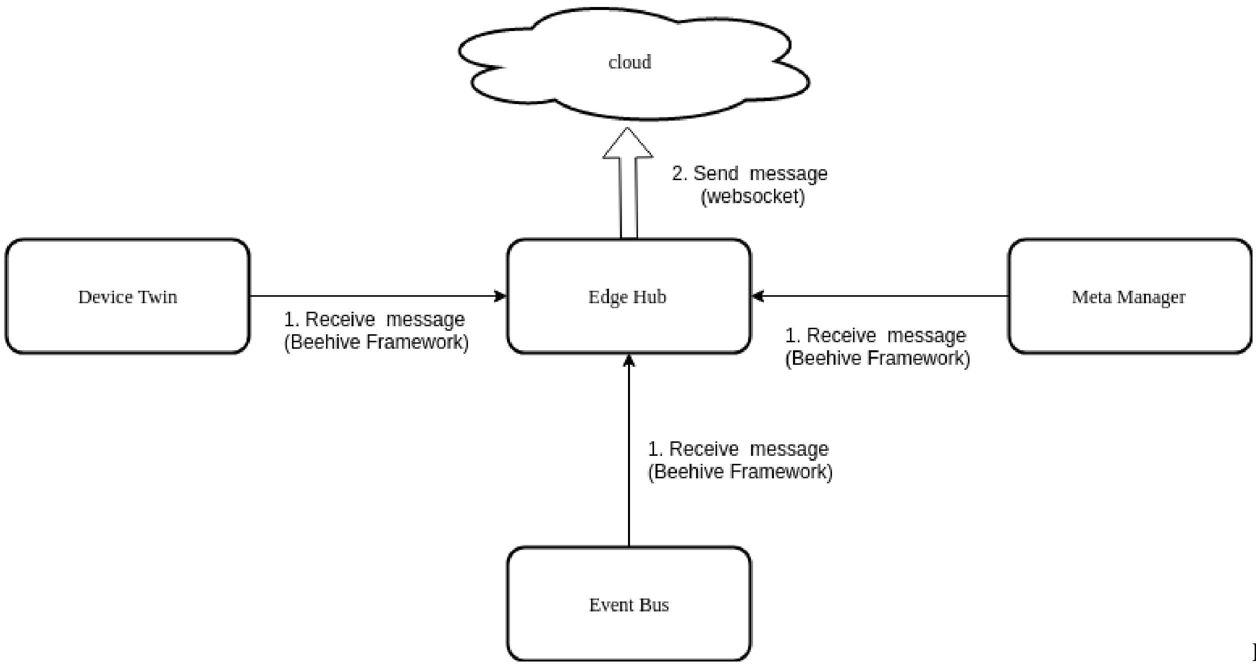
The major steps involved in this process are as follows :-

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

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

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

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


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

Route to Cloud

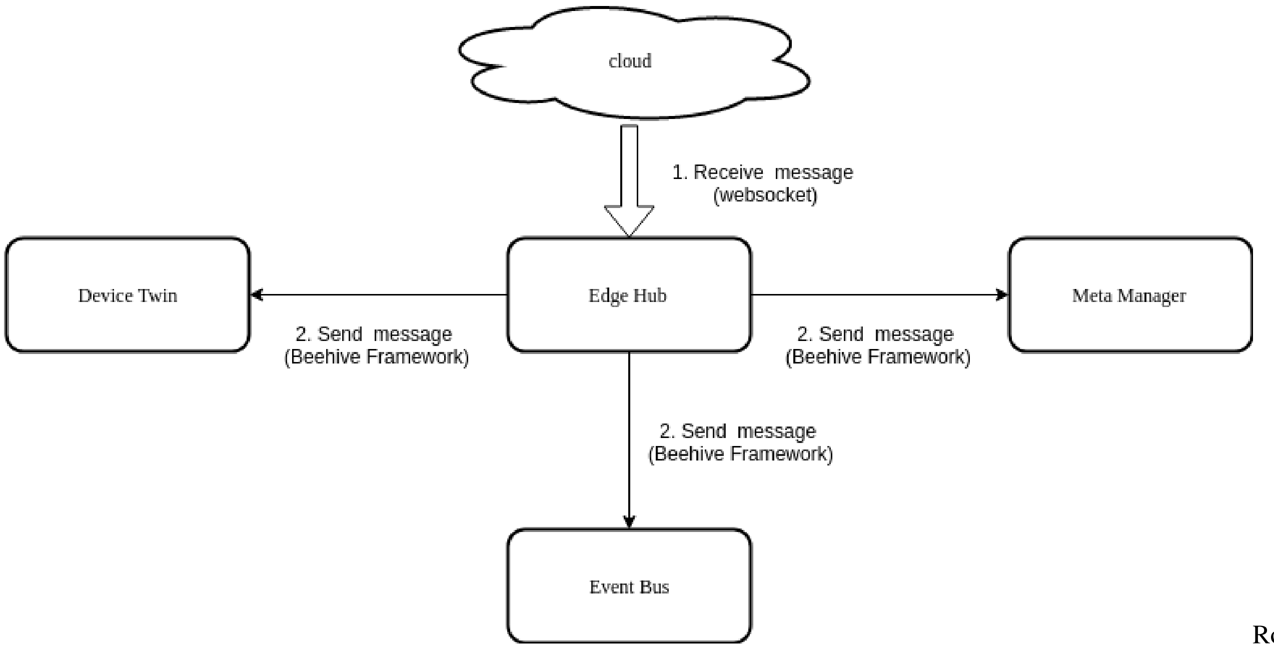
17.5 Route To Edge

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

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

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

- If it is a response message then the message is sent to the syncKeep channel


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

Route to Edge

17.6 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 18

DeviceTwin

18.1 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.

18.2 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

18.2.1 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.

18.2.2 Register and Start Sub Modules

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

18.2.3 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

18.2.4 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.

18.3 Modules

DeviceTwin consists of four modules, namely :-

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

18.3.1 Membership Module

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

The major functions performed by this module are:-

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

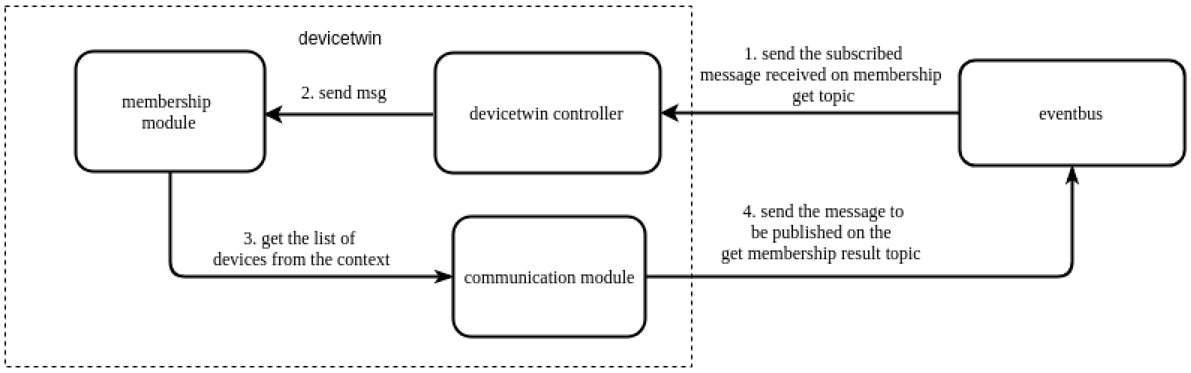
The following are the action callbacks which can be performed by the membership module :-

- dealMembershipGet

- dealMembershipUpdated
- dealMembershipDetail

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

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

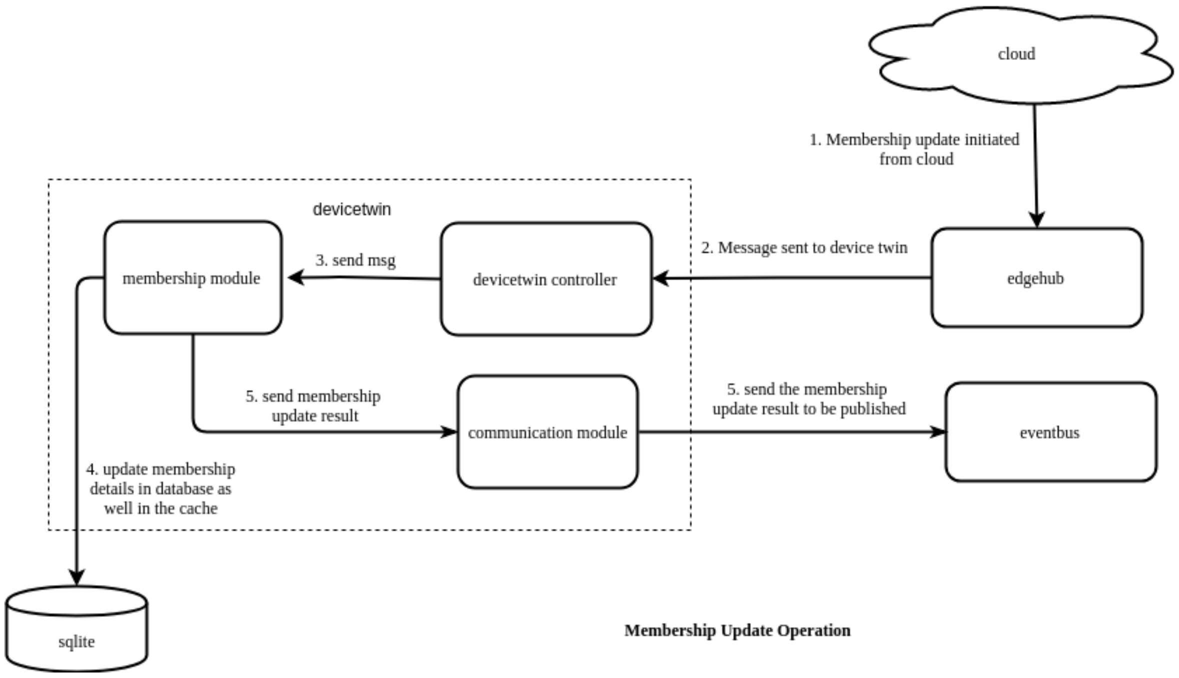

devicetwin
membership module
2. send msg
devicetwin controller
1. send the subscribed message received on membership get topic
eventbus
3. get the list of devices from the context
communication module
4. send the message to be published on the get membership result topic

Memebership Get Operation

Membership Get()

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

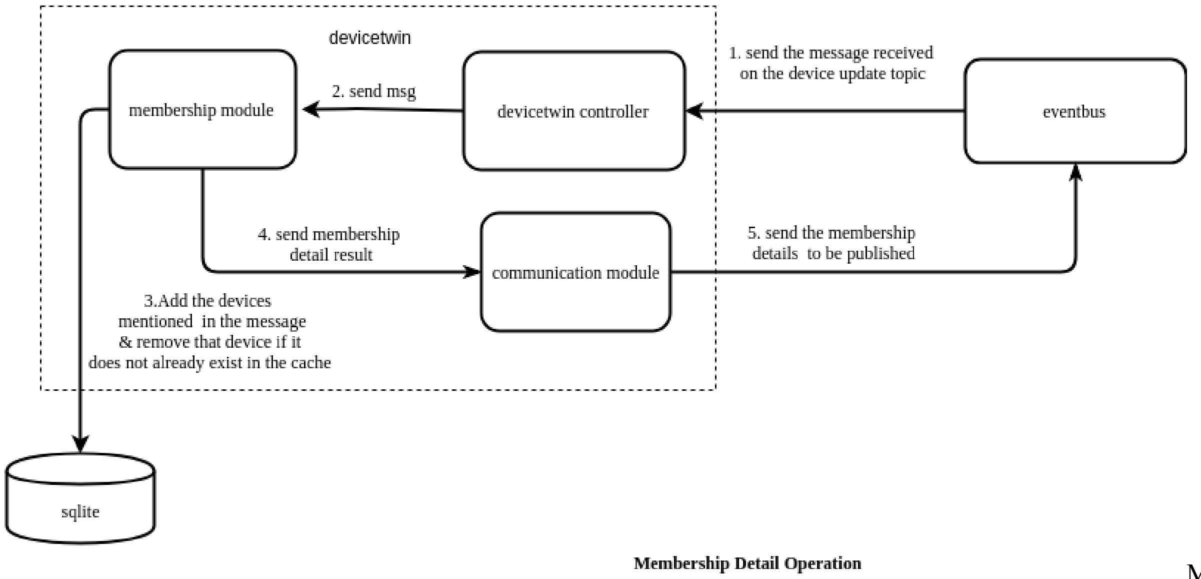
- The edgehub module receives the membership update message from the cloud and forwards the message to devicetwin controller which further forwards it to the membership module.
- The membership module adds devices that are newly added, removes devices that have been recently deleted and also updates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


cloud
1. Membership update initiated from cloud
devicetwin
membership module
3. send msg
devicetwin controller
2. Message sent to device twin
edgehub
5. send membership update result
communication module
5. send the membership update result to be published
eventbus
4. update membership details in database as well in the cache
sqlite
Membership Update Operation

Membership Update

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

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


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

Membership Detail

18.3.2 Twin Module

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

The major functions performed by this module are:-

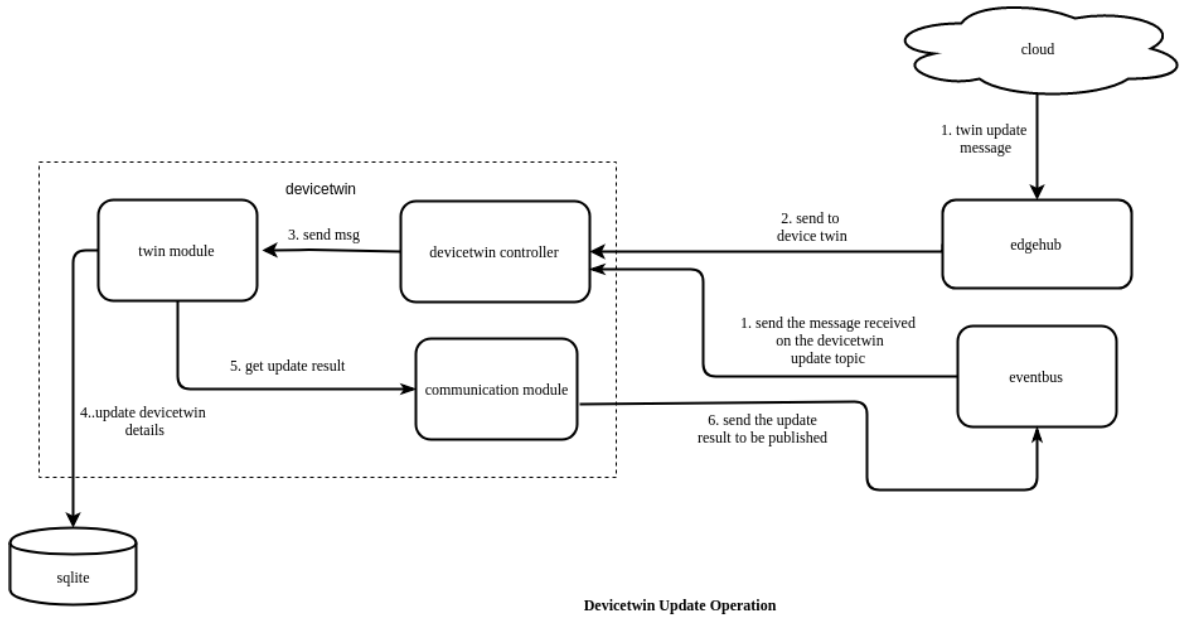
1. Initialize action callback map (which is a map of action(string) to the callback function that performs the requested action)
2. Receive the messages sent to twin module
3. For each message the action message is read and the corresponding function is called
4. Receive heartbeat from the heartbeat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

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

- dealTwinUpdate
- dealTwinGet
- dealTwinSync

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

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

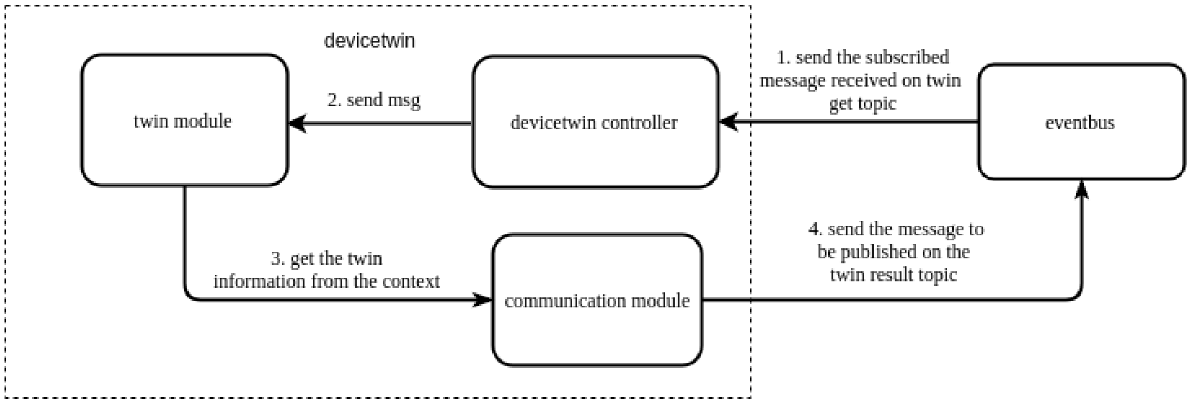


Device Twin Update

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

- The eventbus component receives the message that arrives on the subscribed twin get topic and forwards the message to devicetwin controller, which further sends the message to twin module.

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

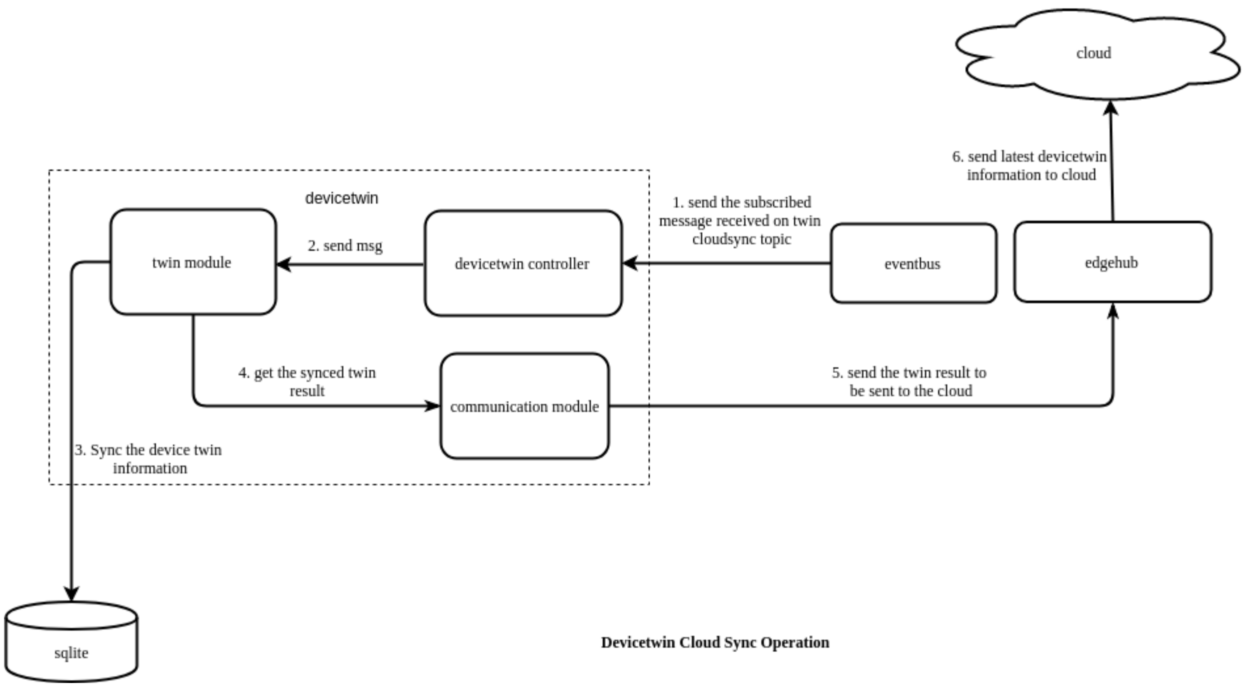

devicetwin
twin module
2. send msg
devicetwin controller
1. send the subscribed message received on twin get topic
eventbus
3. get the twin information from the context
communication module
4. send the message to be published on the twin result topic

Devicetwin Get Operation

Device Twin Get

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

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


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

Sync to Cloud

18.3.3 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.

18.3.4 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 :-

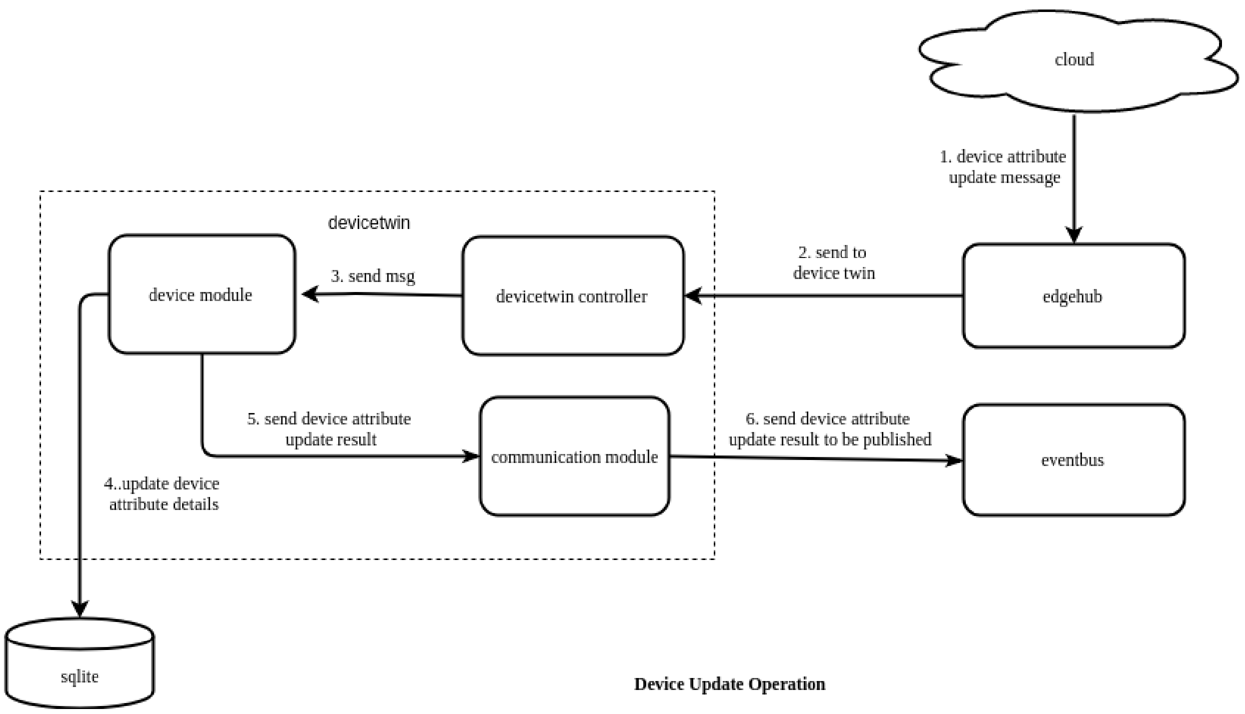
1. Initialize action callback map (which is a map of action(string) to the callback function that performs the requested action)
2. Receive the messages sent to device module
3. For each message the action message is read and the corresponding function is called
4. Receive heartbeat from the heartbeat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

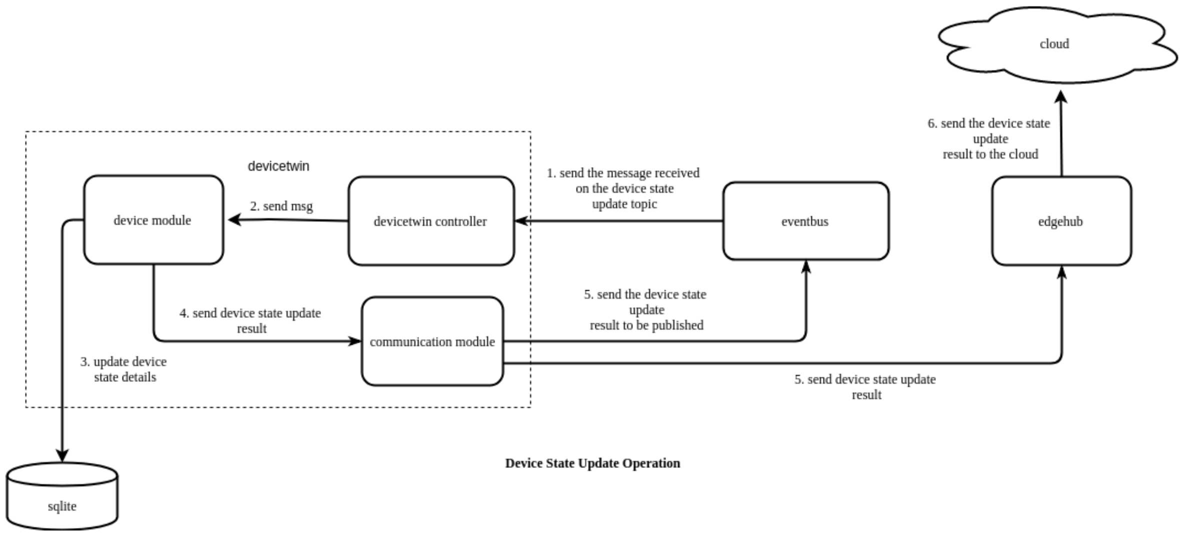
- The device attribute updation is initiated from the cloud, which sends the update to edgehub.
- The edgehub component sends the message to the device twin controller which forwards the message to the device module.
- The device module updates the device attribute details into the database after which,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


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

Device Update

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

- 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.


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

Device State Update

18.4 Tables

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

- Device Table
- Device Attribute Table
- Device Twin Table

18.4.1 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

18.4.2 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.

18.4.3 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 19

EdgeSite: Standalone Cluster at edge

19.1 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.

19.2 Motivation

There are scenarios users need to run a standalone Kubernetes cluster at edge to get full control and improve the offline scheduling capability. There are two scenarios users 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 applications but also the schedule capabilities. So that the CDN edge is manageable regardless the connection to one center.

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

  In some IOT scenarios, users 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.

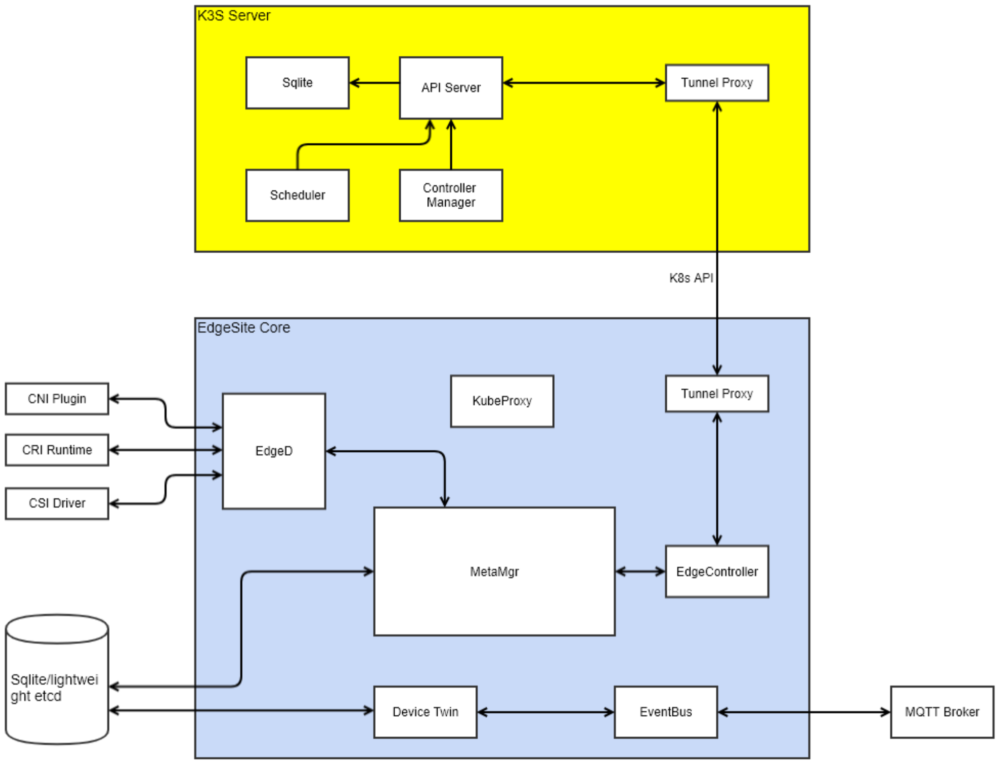
19.3 Assumptions

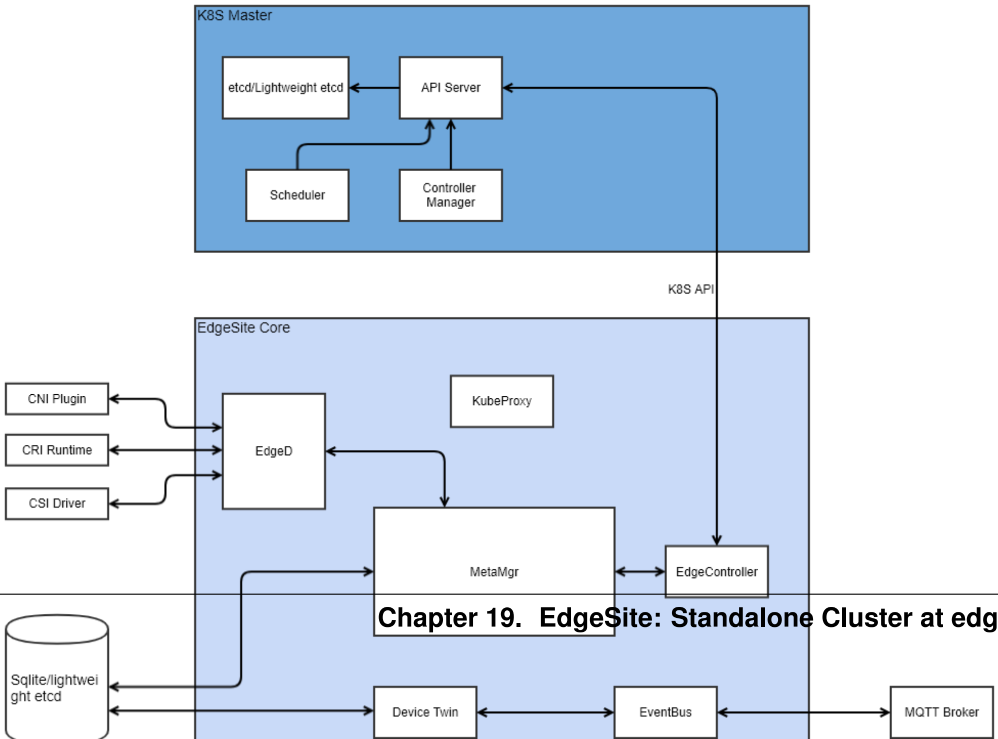
Here we assume a cluster is deployed at edge location including the management control plane. For the management control plane to manage some scale of edge worker nodes, the hosting master node needs to have sufficient resources.

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

19.4 Architecture Design





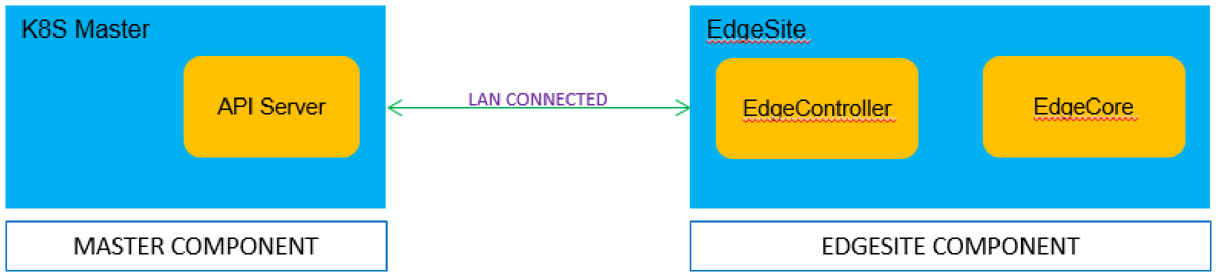
19.5 Advantages

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.

19.6 Getting Started

19.6.1 Setup



EdgeSite Setup

19.6.2 Steps for K8S (API server) Cluster

- Install docker
- Install kubeadm/kubectl
- Creating cluster with kubeadm
- KubeEdge supports https connection to Kubernetes apiserver.

  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
```

- (Optional) KubeEdge also supports insecure http connection to Kubernetes apiserver for testing, debugging cases. 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
```

  Enter the master address in controller.yaml

```
controller:
  kube:
    ...
    master: "http://127.0.0.1:8080" #Note if master and kubeconfig are both set,␣
↪master will override any value in kubeconfig.
```

19.6.3 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
make all WHAT=edgesite
```

Download Release packages

Click here and download.

Configuring EdgeSite

Generate edgesite config by `edgesite --minconfig` and update:

- 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 `edge-node` with an unique edge id/name in below fields :

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

```
controller:
  kube:
    ...
    node-id: edge-node
    node-name: edge-node
    ...
```

(continues on next page)

(continued from previous page)

```
  edged:
    ...
    hostname-override: edge-node
    ...
```

- 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 --config /path/to/edgesite/config > edgesite.log 2>&1 &
```

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

19.6.4 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 `edge-node` 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": "edge-node",
  }
}
```

- 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

19.6.5 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 20

Bluetooth Mapper

20.1 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.

20.2 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.

20.2.1 Building the bluetooth mapper

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/mappers/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 following␣
↪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 bluetooth mapper application

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

20.3 Modules

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

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

20.3.1 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 manager 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.

20.3.2 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 through 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.

20.3.3 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.

20.3.4 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 manager 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>
```

20.4 Usage

20.4.1 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/mappers/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>
        - ......
      - ......
```

20.4.2 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/mappers/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/mappers/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/mappers/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/mappers/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/mappers/bluetooth-mapper/< deviceID >/scheduler/delete

Message:

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

CHAPTER 21

Modbus Mapper

21.1 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.

21.2 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.

21.2.1 Building the modbus mapper

```
cd $GOPATH/src/github.com/kubeedge/kubeedge/mappers/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 following␣
↪command to sign in
```

(continues on next page)

(continued from previous page)

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

21.2.2 Deploying modbus mapper application

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

21.3 Modules

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

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

21.3.1 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.

21.3.2 Modbus Manager

Modbus Manager is a component which can perform a 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 according to the dpl configuration.

21.3.3 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.

21.3.4 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 22

Contributing

22.1 Code of Conduct

Please make sure to read and observe our Code of Conduct.

22.2 Community Expectations

KubeEdge is a community project driven by its community which strives to promote a healthy, friendly and productive environment. The goal of the community is to develop a cloud native edge computing platform built on top of Kubernetes to manage edge nodes and devices at scale and demonstrate resiliency, reliability in offline scenarios. To build a platform at such scale requires the support of a community with similar aspirations.

- See Community Membership for a list of various community roles. With gradual contributions, one can move up in the chain.

22.3 Preparation

- Choose matched golang version and install:
- Fork the repository on GitHub
- Download the repository
- Read this for more details

CHAPTER 23

Governance

The governance model adopted here is heavily influenced by a set of CNCF projects, especially drawing reference from Kubernetes governance. *For similar structures some of the same wordings from kubernetes governance are borrowed to adhere to the originally construed meaning.*

23.1 Principles

- **Open**: KubeEdge is open source community.
- **Welcoming and respectful**: See Code of Conduct.
- **Transparent and accessible**: Work and collaboration should be done in public. Changes to the KubeEdge organization, KubeEdge code repositories, and CNCF related activities (e.g. level, involvement, etc) are done in public.
- **Merit**: Ideas and contributions are accepted according to their technical merit and alignment with project objectives, scope and design principles.

23.2 Code of Conduct

KubeEdge follows the CNCF Code of Conduct. Here is an excerpt:

> As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

23.3 Community Membership

See community membership

23.4 Community Groups

23.4.1 Special Interest Groups (SIGs)

The KubeEdge project is organized primarily into Special Interest Groups, or SIGs. Each SIG is comprised of individuals from multiple companies and organizations, with a common purpose of advancing the project with respect to a specific topic.

The goal is to enable a distributed decision structure and code ownership, as well as providing focused forums for getting work done, making decisions, and on-boarding new Contributors. Every identifiable part of the project (e.g. repository, subdirectory, API, test, issue, PR) is intended to be owned by some SIG.

SIG Chairs

SIGs must have at least one, and may have up to two SIG chairs at any given time. SIG chairs are intended to be organizers and facilitators, responsible for the operation of the SIG and for communication and coordination with the other SIGs, and the broader community.

SIG Charter

Each SIG must have a charter that specifies its scope (topics, sub-systems, code repos and directories), responsibilities, and areas of authority, how members and roles of authority/leadership are selected/granted, how decisions are made, and how conflicts are resolved.

SIGs should be relatively free to customize or change how they operate, within some broad guidelines and constraints imposed by cross-SIG processes (e.g., the release process) and assets (e.g., the kubeedge repo).

A primary reason that SIGs exist is as forums for collaboration. Much work in a SIG should stay local within that SIG. However, SIGs must communicate in the open, ensure other SIGs and community members can find meeting notes, discussions, designs, and decisions, and periodically communicate a high-level summary of the SIG's work to the community. SIGs are also responsible to:

- Meet regularly, at least monthly
- Keep up-to-date meeting notes, linked from the SIG's page in the community repo
- Announce meeting agenda and minutes after each meeting, on the KubeEdge mailing list and/or slack or other channel.
- Ensure the SIG's decision making is archived somewhere public
- Report activity in overall community meetings
- Participate in release planning meetings, retrospective, etc (if relevant)
- Actively triage issues, PRs, test failures, etc. related to code and tests owned by the SIG
- Use the above forums as the primary means of working, communicating, and collaborating, as opposed to private emails and meetings.

23.5 CLA

All contributors must sign the CNCF CLA, as described here.

23.6 Credits

Sections of this documents have been borrowed from Kubernetes governance.

CHAPTER 24

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.

24.1 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.

24.1.1 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

24.1.2 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

24.2 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.

24.2.1 Requirements

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

24.2.2 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

24.3 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.

24.3.1 Requirements

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

24.3.2 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

24.4 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

24.4.1 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

24.4.2 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 25

Feature Lifecycle

This document is to clarify definitions and differences between features and corresponding APIs during different development stages (versions).

Each version has different level of stability, support time, and requires different graduation criteria moving to next level:

- *Alpha*
- *Beta*
- *GA*

25.1 Alpha

The feature may be changed/upgraded in incompatible ways in the later versions.

The source code will be available in the release branch/tag as well as in the binaries.

Support for the feature can be stopped any time without notice.

The feature may have bugs.

The feature may also induce bugs in other APIs/Features if enabled.

The feature may not be completely implemented.

The API version names will be like v1alpha1, v1alpha2, etc. The suffixed number will be incremented by 1 in each upgrade.

25.1.1 Graduation Criteria

- Each feature will start at alpha level.
- Should not break the functioning of other APIs/Features.

25.2 Beta

The feature may not be changed/upgraded in incompatible ways in later versions, but if changed in incompatible ways then upgrade strategy will be provided.

The source code will be available in the release branch/tag as well as in the binaries.

Support for the feature will not be stopped without 2 minor releases notice and will be present in at least next 2 minor releases.

The feature will have very less bugs.

The feature will not induce bugs in other APIs/Features if enabled.

The feature will be completely implemented.

The API version names will be like v1beta1, v1beta2, etc. The suffixed number will be incremented by 1 in each upgrade.

25.2.1 Graduation Criteria

- Should have at least 50% coverage in e2e tests.
- Project agrees to support this feature for at least next 2 minor releases and notice of at least 2 minor releases will be given before stopping the support.
- Feature Owner should commit to ensure backward/forward compatibility in the later versions.

25.3 GA

The feature will not be changed/upgraded in incompatible ways in the next couple of versions.

The source code will be available in the release branch/tag as well as in the binaries.

Support for the feature will not be stopped without 4 minor releases notice and will be present in at least next 4 minor releases.

The feature will not have major bugs as it will be tested completely as well as have e2e tests.

The feature will not induce bugs in other APIs/Features if enabled.

The feature will be completely implemented.

The API version names will be like v1, v2, etc.

25.3.1 Graduation Criteria

- Should have complete e2e tests.
- Code is thoroughly tested and is reported to be very stable.
- Project will support this feature for at least next 4 minor releases and notice of at least 4 minor releases will be given before stopping support.
- Feature Owner should commit to ensure backward/forward compatibility in the later versions.
- Consensus from KubeEdge Maintainers as well as Feature/API Owners who use/interact with the Feature/API.

CHAPTER 26

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.

26.1 Notice

Device Management features are updated from v1alpha1 to v1alpha2 in release v1.4. It is **not** compatible for v1alpha1 and v1alpha2. Details can be found device-management-enhance

26.2 Device Model

A `device model` describes the device properties such as 'temperature' or 'pressure'. 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.

26.2.1 Device Model Sample

A sample device model like below,

```
apiVersion: devices.kubeedge.io/v1alpha2
kind: DeviceModel
metadata:
 name: sensor-tag-model
 namespace: default
spec:
 properties:
  - name: temperature
    description: temperature in degree celsius
```

(continues on next page)

(continued from previous page)

```yaml
    type:
     int:
      accessMode: ReadWrite
      maximum: 100
      unit: degree celsius
  - name: temperature-enable
    description: enable data collection of temperature sensor
    type:
      string:
        accessMode: ReadWrite
        defaultValue: 'OFF'
```

26.3 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 which exposed by property visitors to access. 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.

26.3.1 Device Instance Sample

A sample device instance like below,

```yaml
apiVersion: devices.kubeedge.io/v1alpha2
kind: Device
metadata:
  name: sensor-tag-instance-01
  labels:
    description: TISimplelinkSensorTag
    manufacturer: TexasInstruments
    model: CC2650
spec:
  deviceModelRef:
    name: sensor-tag-model
  protocol:
    modbus:
      slaveID: 1
    common:
      com:
        serialPort: '1'
        baudRate: 115200
        dataBits: 8
        parity: even
        stopBits: 1
  nodeSelector:
    nodeSelectorTerms:
    - matchExpressions:
      - key: ''
        operator: In
        values:
        - node1
```

(continues on next page)

(continued from previous page)

```
  propertyVisitors:
    - propertyName: temperature
      modbus:
        register: CoilRegister
        offset: 2
        limit: 1
        scale: 1
        isSwap: true
        isRegisterSwap: true
    - propertyName: temperature-enable
      modbus:
        register: DiscreteInputRegister
        offset: 3
        limit: 1
        scale: 1.0
        isSwap: true
        isRegisterSwap: true
status:
  twins:
    - propertyName: temperature
      reported:
        metadata:
          timestamp: '1550049403598'
          type: int
        value: '10'
      desired:
        metadata:
          timestamp: '1550049403598'
          type: int
        value: '15'
```

26.3.2 Customized Protocols and Customized Settings

From KubeEdge v1.4, we can support customized protocols and customized settings, samples like below

- customized protocols

```
  propertyVisitors:
    - propertyName: temperature
      collectCycle: 500000000
      reportCycle: 1000000000
      customizedProtocol:
        protocolName: MY-TEST-PROTOCOL
        configData:
          def1: def1-val
          def2: def2-val
          def3:
            innerDef1: idef-val
```

- customized values

```
  protocol:
    common:
      ...
      customizedValues:
```

(continues on next page)

(continued from previous page)

```
        def1: def1-val
        def2: def2-val
```

26.3.3 Data Topic

From KubeEdge v1.4, we add data section defined in device spec. Data section describe a list of time-series properties which will be reported by mappers to edge MQTT broker and should be processed in edge.

```
apiVersion: devices.kubeedge.io/v1alpha1
kind: Device
metadata:
    ...
spec:
  deviceModelRef:
    ...
  protocol:
    ...
  nodeSelector:
    ...
  propertyVisitors:
    ...
  data:
    dataTopic: "$ke/events/device/+/data/update"
    dataProperties:
      - propertyName: pressure
        metadata:
          type: int
      - propertyName: temperature
        metadata:
          type: int
```

26.4 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

26.5 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 updates 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 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"
6. "$ke/events/+/device/data/update"
```

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.

4. `"$ke/events/device/+/data/update"` This topic is add in KubeEdge v1.4, and used for delivering time-serial data. This topic is not processed by edgecore, instead, they should be processed by third-party component on edge node such as EMQ Kuiper.

The content of data topic should conform to following format

```
{
        "event_id": "123e4567-e89b-12d3-a456-426655440000",
        "timestamp": 1597213444,
        "data": {
                "propertyName1": {
                        "value": "123",
                        "metadata": {
                                "timestamp": 1597213444, //+optional
                                "type": "int"
                        }
                },
                "propertyName2": {
                        "value": "456",
                        "metadata": {
                                "timestamp": 1597213444,
                                "type": "int"
                        }
                }
        }
}
```

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

Bluetooth Mapper End to End Test Setup Guide

The test setup required for running the end to end test of bluetooth mapper requires two separate machines in bluetooth range. The paypal/gatt package used for bluetooth mapper makes use of hci interface for bluetooth communication. Out of two machines specified, one is used for running bluetooth mapper and other is used for running a test server which publishes data that the mapper use for processing. The test server created here is also using the paypal/gatt package.

29.1 Steps for running E2E tests

1. Turn ON bluetooth service of both machines
2. Run server on first machine. Follow steps given below for running the test server.
3. For running mapper tests on second machine, clone kubbedge code and follow steps 4,5 and 6.
4. Update "dockerhubusername" and "dockerhubpassword" in tests/e2e/scripts/fast_test.sh with your credentials.
5. Compile bluetooth mapper e2e by executing the following command in $GOPATH/src/github.com/kubeedge/kubeedge. `bash -x tests/e2e/scripts/compile.sh bluetooth`
6. Run bluetooth mapper e2e by executing the following command in $GOPATH/src/github.com/kubeedge/kubeedge. `bash -x tests/e2e/scripts/execute.sh bluetooth`

29.1.1 Test Server Creation

1. Copy devices folder in tests/e2e/stubs and keep it in path TESTSERVER/src/github.com in first machine.
2. Update the following in devices/mockserver.go
    1. package devices -> package main
    2. import "github.com/kubeedge/kubeedge/tests/stubs/devices/services" to "github.com/devices/services"

3. Build the binary using `go build mockserver.go`
4. Run the server using `sudo ./mockserver -logtostderr -duration=<specify duration for which test server should be running>`

sudo is required for getting hci control of the machine.

This runs your test server which publishes data for the mapper to process.

CHAPTER 30

EdgeMesh guide

In case network issue between cloud and edge side, we intergrate EdgeMesh to support DNS visit at any time.

Currently we only support HTTP1.x, more protocols like HTTPS and gRPC coming later.

EdgeMesh is enabled as default.

30.1 Limitation

- Ensure network interface “docker0” exists, which means that EdgeMesh only works for Docker CRI.

30.2 Environment Check

Before run examples, please check environment first.

30.2.1 DNS Order

Modify `/etc/nsswitch.conf`, make sure `dns` is first order, like below:

```
$ grep hosts /etc/nsswitch.conf
hosts:          dns file mdns4_minimal [NOTFOUND=return]
```

30.2.2 IP Forward Setting

Enable ip forward:

```
$ sudo echo "net.ipv4.ip_forward = 1" >> /etc/sysctl.conf
$ sudo sysctl -p
```

Then check it:

```
$ sudo sysctl -p | grep ip_forward
net.ipv4.ip_forward = 1
```

30.3 Usage

Assume we have two edge nodes in ready state, we call them edge node “a” and “b”:

```
$ kubectl get nodes
NAME           STATUS     ROLES    AGE   VERSION
edge-node-a    Ready      edge     25m   v1.15.3-kubeedge-v1.1.0-beta.0.
↪358+0b7ac7172442b5-dirty
edge-node-b    Ready      edge     25m   v1.15.3-kubeedge-v1.1.0-beta.0.
↪358+0b7ac7172442b5-dirty
master         NotReady   master   8d    v1.15.0
```

Deploy a sample pod from Cloud Side:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubeedge/kubeedge/master/build/
↪deployment.yaml
deployment.apps/nginx-deployment created
```

Check the pod is up and is running state, as we could see the pod is running on edge node b:

```
$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP           ␣
↪NODE           NOMINATED NODE   READINESS GATES
nginx-deployment-54bf9847f8-sxk94   1/1     Running   0          14m   172.17.0.2   ␣
↪edge-node-b    <none>           <none>
```

Check it works:

```
$ curl 172.17.0.2
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
```

(continues on next page)

(continued from previous page)

```
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

`172.17.0.2` is the IP of deployment and the output may be different since the version of nginx image.

Then create a service for it:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
  namespace: default
spec:
  clusterIP: None
  selector:
    app: nginx
  ports:
    - name: http-0
      port: 12345
      protocol: TCP
      targetPort: 80
EOF
```

- For L4/L7 proxy, specify what protocol a port would use by the port's "name". First HTTP port should be named "http-0" and the second one should be called "http-1", etc.

Check the service and endpoints:

```
$ kubectl get service
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)     AGE
nginx-svc    ClusterIP   None         <none>        12345/TCP   77m
$ kubectl get endpoints
NAME         ENDPOINTS            AGE
nginx-svc    172.17.0.2:80        81m
```

To request a server, use url like this: `<service_name>.<service_namespace>.svc.<cluster>.<local>:<port>`

In our case, from edge node a or b, run following command:

```
$ curl http://nginx-svc.default.svc.cluster.local:12345
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

(continues on next page)

(continued from previous page)

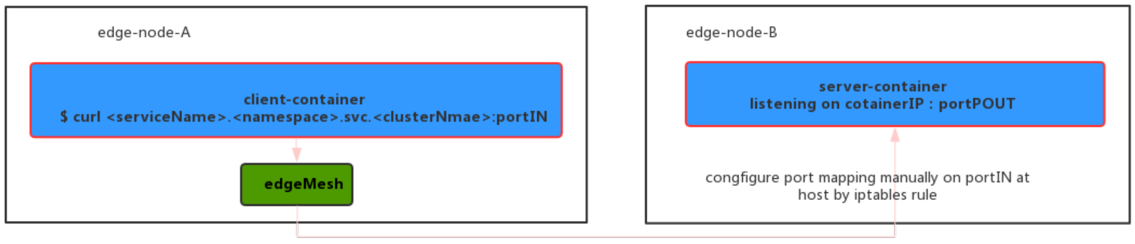
```
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

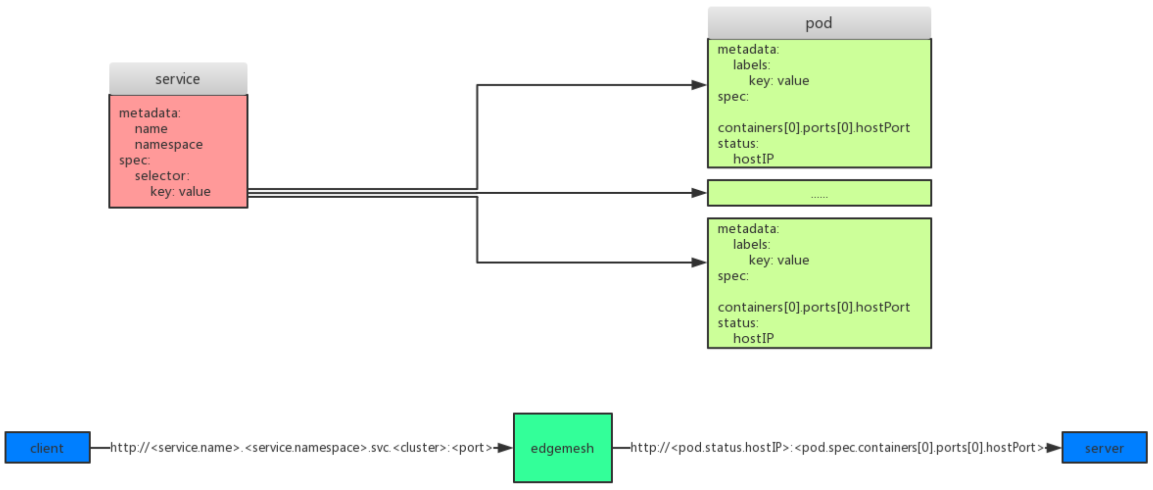
- EdgeMesh supports both Host Networking and Container Networking
- If you ever used EdgeMesh of old version, check your iptables rules. It might affect your test result.

30.4 Sample


edge-node-A
client-container
$ curl <serviceName>.<namespace>.svc.<clusterNmae>:portIN
edgeMesh
edge-node-B
server-container
listening on cotainerIP : portPOUT
congfigure port mapping manually on portIN at
host by iptables rule

sample

30.5 Model

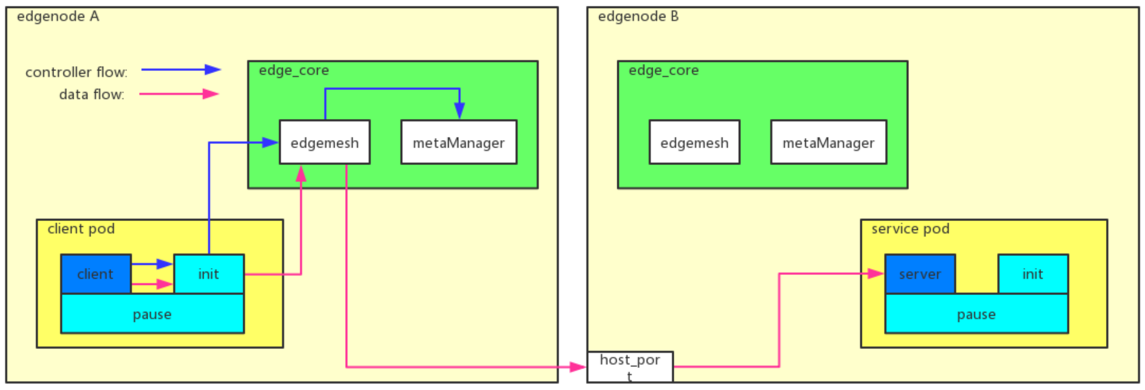


service
metadata:
name
namespace
spec:
selector:
key: value
pod
metadata:
labels:
key: value
spec:
containers[0].ports[0].hostPort
status:
hostIP
......
client
http://<service.name>.<service.namespace>.svc.<cluster>:<port>
edgemesh
http://<pod.status.hostIP>:<pod.spec.containers[0].ports[0].hostPort>
server

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. to request a server, use: `<service_name>.<service_namespace>.svc.<cluster>:<port>`:
   1. get the service's name and namespace from domain name
   2. query all the backend pods from MetaManager by service's namespace and name
   3. LoadBalance returns the real backend containers' hostIP and hostPort

30.6 Flow



flow

1. client requests to server by server's domain name
2. DNS being hijacked to EdgeMesh by iptables rules, then a fake ip returned

3. request hijacked to EdgeMesh by iptables rules
4. EdgeMesh resolves request, gets domain name, protocol, request and so on
5. EdgeMesh load balances:
    1. get the service's name and namespace from the domain name
    2. query backend pods of the service from MetaManager
    3. choose a backend based on strategy
6. EdgeMesh transports request to server, wait for server's response and then sends response back to client

CHAPTER 31

Measuring memory footprint of EdgeCore

31.1 Why measure memory footprint?

- This platform is designed for a light-weight edge computing deployment, capable of running on devices with few resources (for example, 256MB RAM)
- It is important to know when deploying many pods that it showcases as little memory footprint as possible

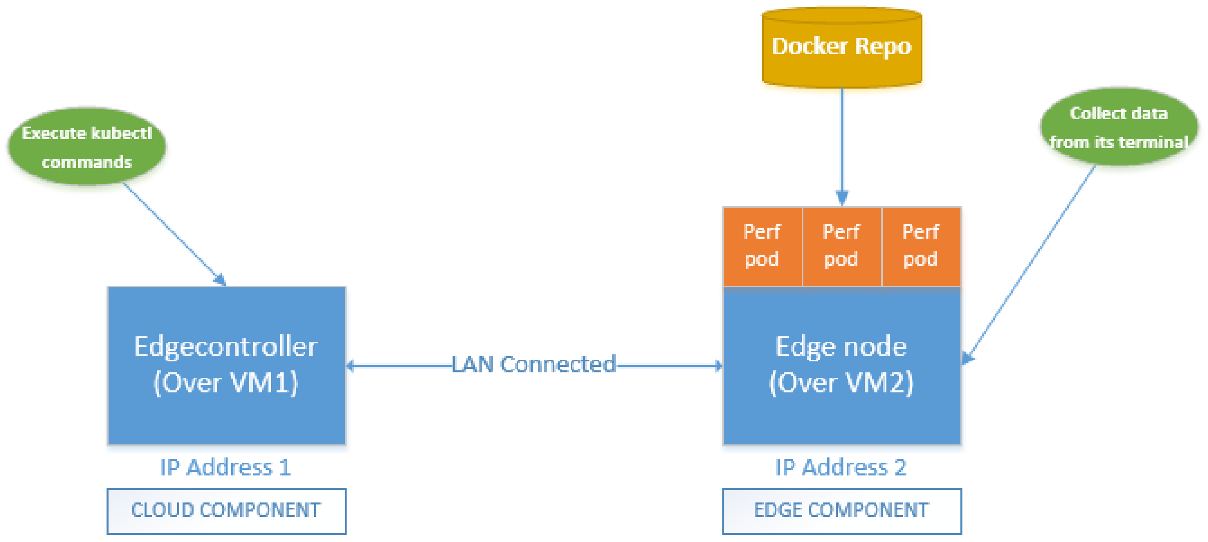
31.2 KPI's measured

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

31.3 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.

31.3.1 Test setup



KubeEdge

Test Setup

Fig 1: KubeEdge Test Setup

31.3.2 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]

31.3.3 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 edgecore`. 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 ./edgecore
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 32

FAQs

This page contains a few commonly occurring questions.

32.1 keadm init failed to download release

If you have issue about connection to github, please follow below guide with proxy before do setup, take `v1.4.0` as example:

- download release pkgs from release page
- download crds yamls matches the release version you downloaded, links as below:
- put them under `/etc/kubeedge` as below:

```
$ tree -L 3
.
├── crds
│   ├── devices
│   │   ├── devices_v1alpha2_devicemodel.yaml
│   │   └── devices_v1alpha2_device.yaml
│   └── reliablesyncs
│       ├── cluster_objectsync_v1alpha1.yaml
│       └── objectsync_v1alpha1.yaml
└── kubeedge-v1.4.0-linux-amd64.tar.gz

3 directories, 5 files
```

Then you can do setup without any network issue, `keadm` would detect them and not download again(make sure you specify `v1.4.0` with option `--kubeedge-version 1.4.0`).

32.2 Container keeps pending/ terminating

1. Check the output of `kubectl get nodes`, whether the node is running healthy. Note that nodes in unreachable, offline status cannot complete graceful/soft pod deletion until they come back to normal.
2. Check the output of `kubectl describe pod <your-pod>`, whether the pod is scheduled successfully.
3. Check the `edgecore` logs for any errors.
4. Check the architecture of the node running `edgecore` and make sure that container image you are trying to run is of the same architecture. For example, if you are running `edgecore` on Raspberry Pi 4, which is of `arm64v8` architecture, the nginx image to be executed would be `arm64v8/nginx` from the docker hub.
5. Also, check that the `podSandboxImage` is correctly set as defined in Modification in edgecore.yaml.
6. If all of the above is correctly set, login manually to your edge node and run your docker image manually by

   ```
   docker run <your-container-image>
   ```

7. If the docker container image is not pulled from the docker hub, please check that there is enough space on the edge node.

32.3 Where do we find cloudcore/edgecore logs

This depends on the how cloudcore/ edgecore has been executed.

1. If `systemd` was used to start the cloudcore/ edgecore? then use `journalctl --unit <name of the service probably edgecore.service>` to view the logs.
2. If `nohup` was used to start the cloudcore/ edgecore, either a path would have been added where the log is located, Otherwise, if the log file wasn't provided, the logs would be written to stdout.

32.4 Where do we find the pod logs

Connect to the edge node and then either

- use the log file located in `/var/log/pods` or
- use commands like `docker logs <container id>`

kubectl logs is not yet supported by KubeEdge.