Edge Application

The idea behind our application was to create an automatic greenhouse. The greenhouse would be equipped with uv, humidity, and temperature sensors. The sensor data are then used to generate commands for things such as the windows and ac in the greenhouse. Since the network between the edge and the cloud is unreliable, our edge components do not require the cloud to function. However, the sensor data are augmented by weather data from an external weather API. This weather data is gathered and processed by the cloud. The cloud component generates an optimal command for the greenhouse, which is then relayed back to the edge component via our reliable messaging implementation.

Reliable messaging implementation.

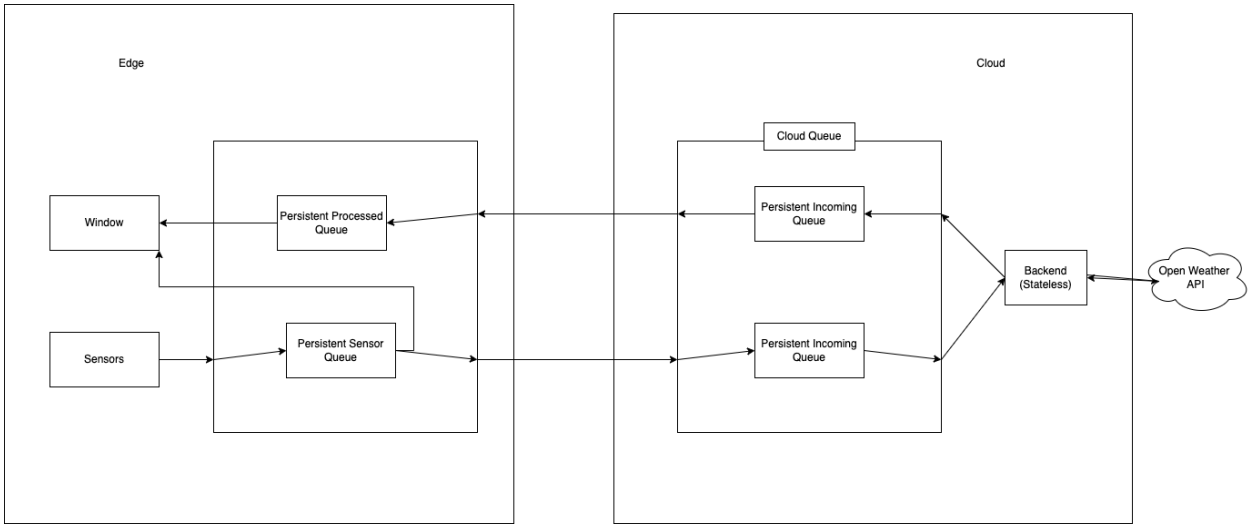
Our reliable messaging implementation consists of two separate queues. One on the edge device and the other in the cloud. In the following two paragraphs, we will explain how each of the functions.

The edge messaging queue is implemented using shelve. All the updates in this map are protected using a mutex. There are two queues:

- sensor-queue: Data received from the sensor is enqueued here. Data is dequeued when the processed-queue receives processed data with the same id. Data is also dequeued if the controller reads directly from the sensor queue when the processed-queue is empty.
- processed-queue: processed data from the cloud-broker is enqueued here. The queue is dequeued when the controller reads from it.

The cloud queue is implemented as a threaded flask app. The application will listen for incoming requests and write them to the incoming queue. A separate thread will attempt to hand the request on to the backend application. The generated command from the backend is placed in the outgoing queue of the cloud queue system. Once the edge device is able to reach the cloud, the generated command can be read from the queue.

The queue used internally by the cloud queue system is an extension of the standard thread-safe queue implementation of python. The queue is used in tandem with pickle to preserve the state even if the backend crashes. On start up the last state of both the incoming and outgoing queue is restored.



This image shows the basic setup of and flow of messages in our application. We did not put the controller and the sensors in one application because we figured that this would not be guaranteed in a realistic edge application. In the demo, the two cloud components run in a single google cloud VM and controller, sensor and edge-broker in a local machine.

The sensor application reads data from two bricklet sensors, the humidity and UV sensors (the UV sensor supplies both UV light and temperature conditions), every 10 seconds. It also has two HTTP endpoints. One to set the location “GET or POST /location” and one to set the targets “GET or POST /targets”. Both of these values are persistent. The targets are used to define the values that the greenhouse will attempt to reach.

```
ipcon = IPConnection() # Create IP connection
h = BrickletHumidityV2(UID_H, ipcon) # Create humidity bricklet object
uvl = BrickletUVLight(UID_UV, ipcon) # Create uv bricklet object
ipcon.connect(HOST, PORT)

# Executes the checkValues function every x sencdons/minutes.
# Important: If the current checkValues execution is not finished the next one will be skipped.
# Maximum number of running instances = 1
scheduler = BlockingScheduler()
scheduler.add_job(checkValues, 'interval', seconds=10)
scheduler.start()

@api.resource("/targets")
class Command(Resource):

    def __init__(self):
        self.reqparse = reqparse.RequestParser()
        self.reqparse.add_argument( "temprature_target", required=True, type = float)
        self.reqparse.add_argument( "humidity_target", required=True, type = float)
        self.reqparse.add_argument( "uv_target", required=True, type = float)
        super( Command, self).__init__()

    def get(self):
        return getTargets(), 200

    def post(self):
        self.args = self.reqparse.parse_args()
        db = shelve.open("targets", writeback=False)
        db["temprature_target"] = self.args["temprature_target"]
        db["humidity_target"] = self.args["humidity_target"]
        db["uv_target"] = self.args["uv_target"]
        db.close()
        return "ok", 200
```

The window controller reads data from the processed queue and if it gets an empty reply, it reads directly from the sensor queue. The processed queue contains more information in the message as the control information is enhanced with weather data.

```
async function getNewStatus(){
    try{
       const response =  await axios.get("http://"+EDGE_BROKER+"/processedQueue");
       if (response.status == 200){
          windowStatus = response.data;
          console.log("New status from broker's processed queue:",windowStatus);
       }else{
          console.log("Unexpected response code:",response.status);
       }
    }catch(error:any){
        if (error.response){
            if (error.response.status === 404){
                console.log("Processed Queue is empty");
                await new Promise(r => setTimeout(r, 10000));
                readDirectlyFromSensor();
            }
        }else if(error.request){
            console.error("Error. Cannot connect to edge broker!");
        }else{
            console.error("Unknown Error:", error.message)
        }
    }
}
let windowStatus:Status = {};

async function subscriber(){
    await getNewStatus();
    setTimeout(subscriber, 5000);
}
subscriber();
```

The cloud-broker: reads and writes data from the cloud broker using HTTP. It also has HTTP methods to read from the sensor-queue and processed-queue for the controller. The images below show excerpts from the implementation.

```python
def send_to_cloud_broker():
    try:
        if not sensor_data.empty():
            data = sensor_data.get_not_sent()
            if data != None:
                print("http://"+CLOUD_BROKER+"/queue")
                response = requests.post("http://"+CLOUD_BROKER+"/queue", json=data)
            if response.status_code == 200:
                sensor_data.mark_as_sent(data["timestamp"])
            elif response.status_code == 409:
                sensor_data.delete(data['timestamp'])
            else:
                print(f'HTTP Error while writing to cloud broker: {response.status_code} \n {response.text}')
    except Exception as ex:
        print(f'Error while writing to cloud broker: {ex}')

def read_from_cloud_broker():
    try:
        response = requests.get("http://"+CLOUD_BROKER+"/queue")
        if response.status_code == 200:
            json = response.json()
            sensor_data.delete(json["timestamp"])
            processed_data.insert(json["timestamp"], json)
        elif response.status_code != 404:
            print(f'HTTP Error while reading from cloud broker: {response.status_code}')

    except Exception as ex:
        print(f'Error while reading from cloud broker: {ex}')
```

```python
    def insert(self, id, object):
        with self.mutex:
            if self.can_be_inserted(id) == False:
                return
            id_list = self.queue['ids']
            id_list.append(id)
            self.queue['ids'] = id_list
            object['sent'] = False
            self.queue[id] = object
            if len(self.queue['ids']) > self.max_size:
                self.pop(False)
    def delete(self,id):
        with self.mutex:
            ids = self.queue['ids']
            if id in ids:
                ids.remove(id)
                self.queue['ids'] = ids
            if id in self.queue:
                del self.queue[id]
    def pop(self, withLock=True):
        if withLock:
            self.mutex.acquire()
        retVal = None
        ids = self.queue['ids']
        if len(ids) > 0:
            id = ids.pop(0)
            self.queue['ids']=ids
            if id in self.queue:
                retVal = self.queue[id]
```

Example of how decisions are made by the cloud backend. The targets, outside conditions, and measured sensor values are considered.

```
# converts temprature data to temprature command
def tempratureSetting(self):
    """
    Use outside, measured, and target temprature to respond with optimised command.
    Aritificial Reulgation:
        Target > Temp:
            Turn on heater
        Target < Temp:
            Turn on AC
        Else:
            Turn off AC and heater
    Wind Alert == False:
        Window Regulation:
            Target > Temp && Out_Temp > Target:
                Turn off heater & ac
                Open window
            Target < Temp && Out_Temp < Target:
                Turn off ac & heater
                Open window
            Else:
                Do nothing
    Return heater, ac, window
    """
    # Artificial regulation
    heater = False
    ac = False
    window = False
    if self.args["temprature_target"] > self.args["temprature"]:
        heater = True
    elif self.args["temprature_target"] < self.args["temprature"]:
```

Implementation of the persistent queue in the cloud queue system.

```
class PersistentQueue(queue.Queue):

    def __init__(self, name, maxsize):
        self.maxsize = maxsize
        self.__Lock = Lock()
        self.persistentFile = f"./pickle/{name}.pkl"
        super(PersistentQueue, self).__init__(maxsize)
        try:
            self.restore()
        except:
            pass

    def put(self, item):
        with self.__Lock:
            super(PersistentQueue, self).put(item)
            self.save()

    def get(self):
        with self.__Lock:
            item = super(PersistentQueue, self).get()
            self.save()
        return item

    def peek(self):
        with self.__Lock:
            item = super(PersistentQueue, self).get()
            self.restore()
        return item

    def save(self):
        with open(self.persistentFile, 'wb') as f:
                pickle.dump(self.queue, f)

    def restore(self):
        with open(self.persistentFile, 'rb') as f:
                self.queue = pickle.load(f)
```