Report Prototyping Assignment:
A public E-Bike Station with Roof Top Solar Panel

The Use Case

Due to climate change and other environmental aspects, solutions have to be found to fundamentally change the way mobility in as well as energy production for cities work. A system of public e-bikes with dedicated stations for parking and charging can contribute to the revolution of urban mobility. Stations will be equipped with and edge device at the station including sensors to communicate the availability and battery level of bikes to a cloud application. The cloud application will provide the possibility to reserve a bike shortly before usage.

Source: https://www.cleanenergyplanet.com/products/city-ls/

To make the e-bike stations more sustainable, we propose an additional roof for them, on top of which a large solar panel is installed to produce electricity. It includes a sensor for the current electricity production. When the produced amount does not suffice for charging purposes, the remaining energy will be obtained from a normal electricity contract. Excess electricity can be fed into the grid and sold on the electricity spot market to make extra profit. Additionally, the cloud application will communicate the current market prices to the station such that the edge device can compare them to the electricity contract price. With this information, the edge device can decide to sell, all produced electricity and rather pay the contract price for charging the e-bikes when this results in an overall profit. All data produced by the edge device is stored in the cloud to enable its later use for e.g. business model analysis.

Most Important Simplifications and Abstractions

Rules and regulations for feeding in to the electricity grid as well as the specifics of the spot market are not considered. The German electricity market provides the option to sell electricity in quarter hourly blocks but further flexibilizations are yet to be implemented on the market in the course of the energy transition. In our implementation, the amounts of electricity consumed by charging an E-Bike are abstracted to 1 kWh at all times which is completely unrealistic but this is a prototype. Also the production capacity of the solar unit is set to be just one kWh bigger than the number of spots at the station. Additionally, the market price for electricity is abstracted to be a random value between 0.27 and 0.68. Reservations are not created in a user interface or even a separated service, but more or less randomly requested by a function. This is due to the lack of time for creating a proper frontend.

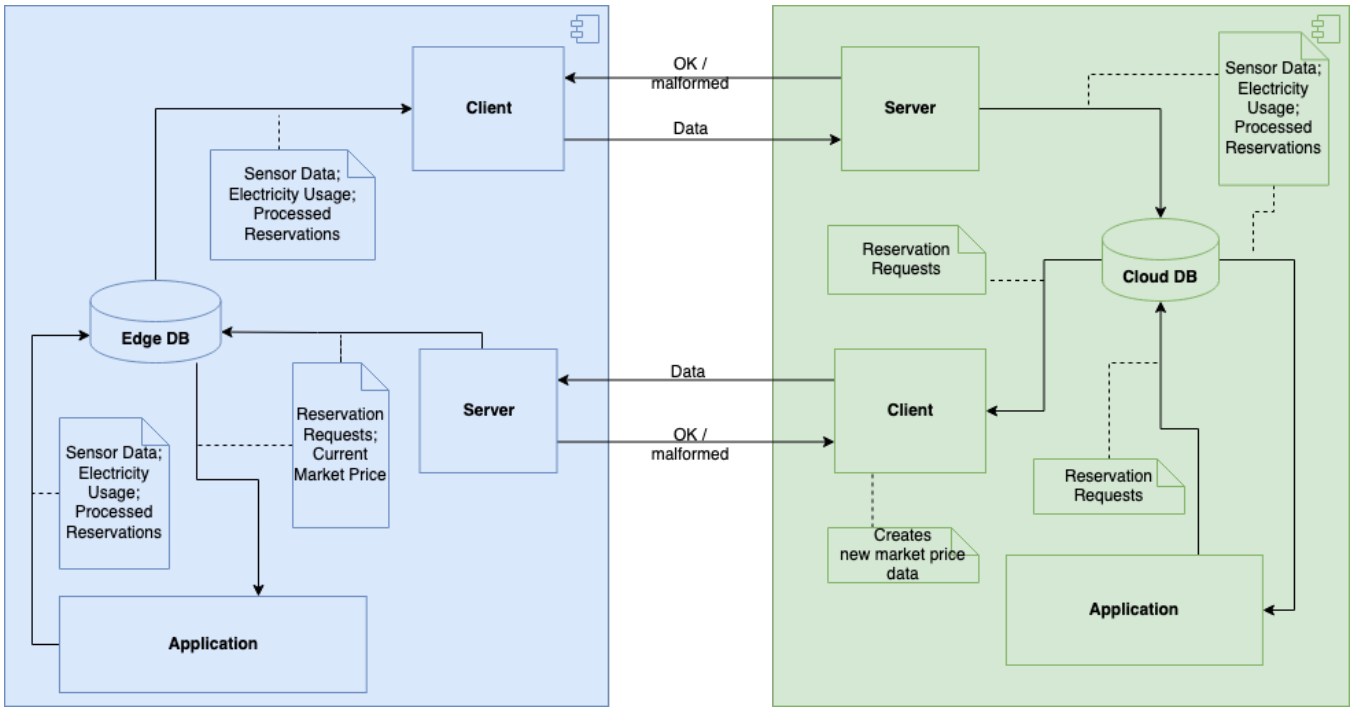
Implementation

Used Technologies

The Prototype is implemented in Python using the ZeroMQ library for messaging. For storage, SQLite databases are added and manipulated via the Sqlalchemy ORM python library. The cloud component is running in a Google Cloud VM instance.

Architecture/Design

Both components (edge and cloud component) each have three main modules: an application which generates and displays data, a client to send data and a server to receive data. This means, it is not a classical client-server architecture as servers do not answer requests but just send an OK reply when upon receiving data. The data gets then processed by the application and then sent back to the other component by the client. Additionally, they both have a database which fulfills two purposes. Firstly to produce data and secondly to queue data which has not been communicated successfully yet.



Data Generation

As stated above, data is generated and stored to the database by the respective component's applications.
The cloud application only creates reservation requests for spots it currently knows to be occupied and not reserved.
The edge application includes an instance of a Python class representing the station with all its spots. Additionally, the station class has a virtual sensor for the production of the solar panel and variables for data relevant for decision making.

```python
class BikeStation:
    """The complete bike station with all sensors."""

    feed_in = None
    self_consumption = None
    electricity_contract_price = 0.4
    current_market_price = 0.4

    def __init__(self, number_of_spots=5):
        self.number_of_spots = number_of_spots
        self.spots = dict((i, BikeSpot(i)) for i in range(0, number_of_spots))
        # solar panel's production capacity is abstracted to equal exactly the demand
        # of the fully occupied station, i.e. number_of_spots
        self.solar_panel_sensor = SolarPanelSensor(production_capacity=number_of_spots)
        self.decide_electricity_usage()
```

The spots are also represented as Python objects and have virtual sensors to get information on wether a spot is occupied and if so the battery level of the parked e-bike.

```python
class BikeSpot:
    """Virtual representation of one bike spot."""

    def __init__(self, spot_id):
        self.spot_id = spot_id
        self.occupied_sensor = SpotOccupiedSensor()
        self.bike_battery_sensor = BikeBatterySensor() if self.occupied_sensor.occupied else None
        self.reservation_state = ReservationState()
```

All sensors do produce more or less random values except for the bike battery sensor, which only gets assigned a random value when a new bike is put into a spot, but then increases the battery by a factor over time.

```python
class BikeBatterySensor:
    def __init__(self):
        # initial is battery level never 0% or 100%
        self.battery_level = round(random.uniform(0.01, 0.99), 4)
        self.level_increase_per_second = 0.0033
        self.last_sensed = datetime.datetime.utcnow()

    def _update_battery_level(self):
        """Increases battery level depending on seconds passed since last reading.
        No effect on battery level, if battery is fully charged
        """
        now = datetime.datetime.utcnow()
        seconds_passed = (now - self.last_sensed).total_seconds()
        self.last_sensed = now
        if self.battery_level == 1:
            return
        level_increase = round(seconds_passed * self.level_increase_per_second, 4)
        increased_level = self.battery_level + level_increase
        # battery doesn't go over 100%
        self.battery_level = increased_level if increased_level <= 1 else 1
```

Additionally, the edge application decides on the best strategy for electricity usage (feed-in or self-consumption) and sets the respective variables accordingly.

```python
    def decide_electricity_usage(self):
        """Set electricity usage attributes.
        Based upon own current consumption state and market price compared to contract price,
        decide whether to feed in and/or self-consume produced electricity.
        """
        # electricity demand is abstracted to equal the number of occupied spots
        self.current_market_price = models.Constant.get_real_value_by_name('current_market_price')
        current_demand = self.get_number_of_occupied_spots()
        current_production = self.solar_panel_sensor.current_production
        if (
            current_demand == 0
            or self.electricity_contract_price < self.current_market_price
        ):
            # Exclusive feed in
            # No occupied spots means no electricity usage so feed in if producing anything
            # A higher market price means profit when feeding in all production instead of self-consuming
            self.feed_in = current_production
            self.self_consumption = 0
        else:
            if current_production <= current_demand:
                # Exclusive self-consumption
                # Producing less than or equal to own demand, so do only self-consumption
                self.feed_in = 0
                self.self_consumption = current_production
            else:
                # Producing more than own demand, so feed in the remaining electricity to grid
                self.feed_in = current_production - current_demand
                self.self_consumption = current_demand
```

Reliable Messaging
As mentioned earlier, the SQLite database is used to store unsent messages. This means, the database is used as a way of queueing messages. This is done as follows.
Upon creation of new data, e.g. new readings from the virtual sensors, an entry to a database table is made that includes a field “sent_status”, initially set to “unprocessed”.

```python
class SpotSensorData(Base):
    __tablename__ = "spot_sensor_reading"
    read_id = Column(Integer, primary_key=True)
    read_timestamp = Column(DateTime(timezone=True), server_default=func.now(tz=pytz.utc))
    update_timestamp = Column(DateTime(timezone=True), onupdate=func.now(tz=pytz.utc))
    spot_id = Column(Integer, nullable=False)
    is_occupied = Column(Boolean, default=False)
    battery_level = Column(REAL)
    sent_status = Column(Enum(Status), default=Status.created)
```

The client fetches unsent data and tries to send it to the other component (there is a limit to how many data items should be sent to not overload one message).

```python
for sequence in itertools.count():
    # get oldest 10 sensor readings from db
    queued_readings = SpotSensorData.get_oldest_n_readings(10)
    # get oldest 10 electricity data items from db
    queued_electricity_data = ElectricityData.get_oldest_n_readings(10)

    # get processed reservations from db
    confirmed_reservations = Reservation.get_confirmed_reservation_requests()
    rejected_reservations = Reservation.get_rejected_reservation_requests()
```

It then sends it to the other component’s server. In case of an OK-reply, it sets all sent data items to have the sent_status “sent”. As long as there is no reply, the data will be resent. This ensures that all produced data will reach the other component eventually.

```python
while True:
    if (client.poll(REQUEST_TIMEOUT) & zmq.POLLIN) != 0:
        reply = client.recv()
        if int(reply) == 4:  # sanity check with length of sent object
            logging.info("Server replied OK")
            # status of sent records need to be set to "processed"
            if len(queued_readings) > 0:
                SpotSensorData.set_to_processed(queued_readings[len(queued_readings)-1].read_id)
            if len(queued_electricity_data) > 0:
                ElectricityData.set_to_processed(queued_electricity_data[len(queued_electricity_data)-1].data_item_id)
            for reservation in rejected_reservations:
                reservation.update_response_sent()
            for reservation in confirmed_reservations:
                reservation.update_response_sent()
            break
        else:
            logging.error("Malformed reply from server: %s", reply)
            continue
    # {REQUEST_TIMEOUT} seconds passed, but no results yet
    logging.warning("No response from server")
    # Socket is confused. Close and remove it.
    client.setsockopt(zmq.LINGER, 0)
    client.close()
    logging.info("Reconnecting to server…")
    # Create new connection
    client = context.socket(zmq.REQ)
    client.connect(server_url)
    logging.info("Resending sensor data, electricity info and reservation responses.")
    client.send(encoded)
```

Space Management

To avoid running out of storage space on the edge device, there are “cleaning” functions for all message data models which delete all entries that have a sent status of “sent”. Those functions can be called regularly, e.g. once per hour. This reduces the risk of running out of space.