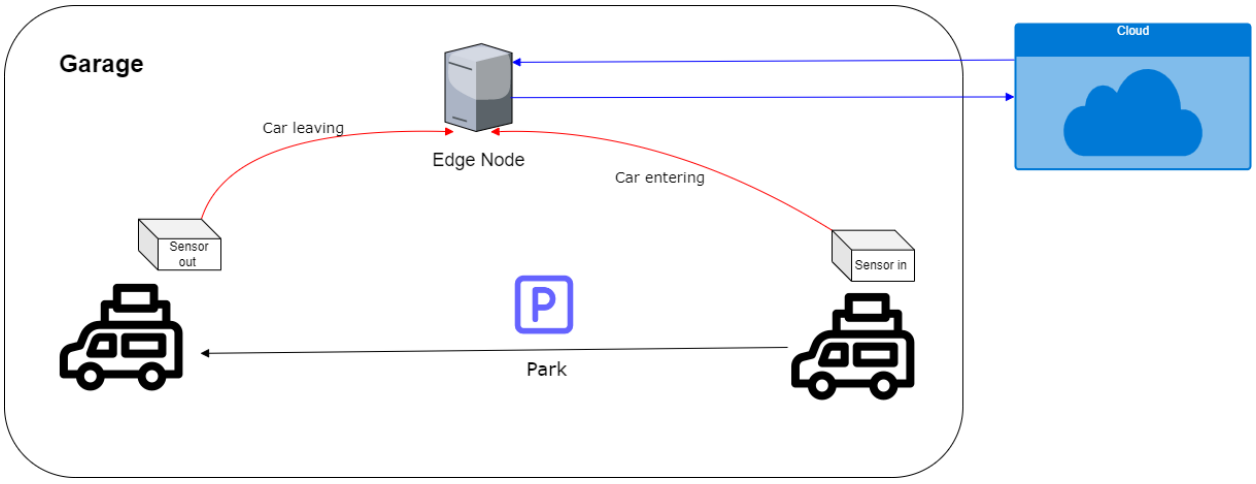
Prototype Use Case: United Parking Garages (simulated)

GitHub: https://github.com/marcplustwo/FC_reliable_messaging

Author: Marc Radau, Tianle Luo

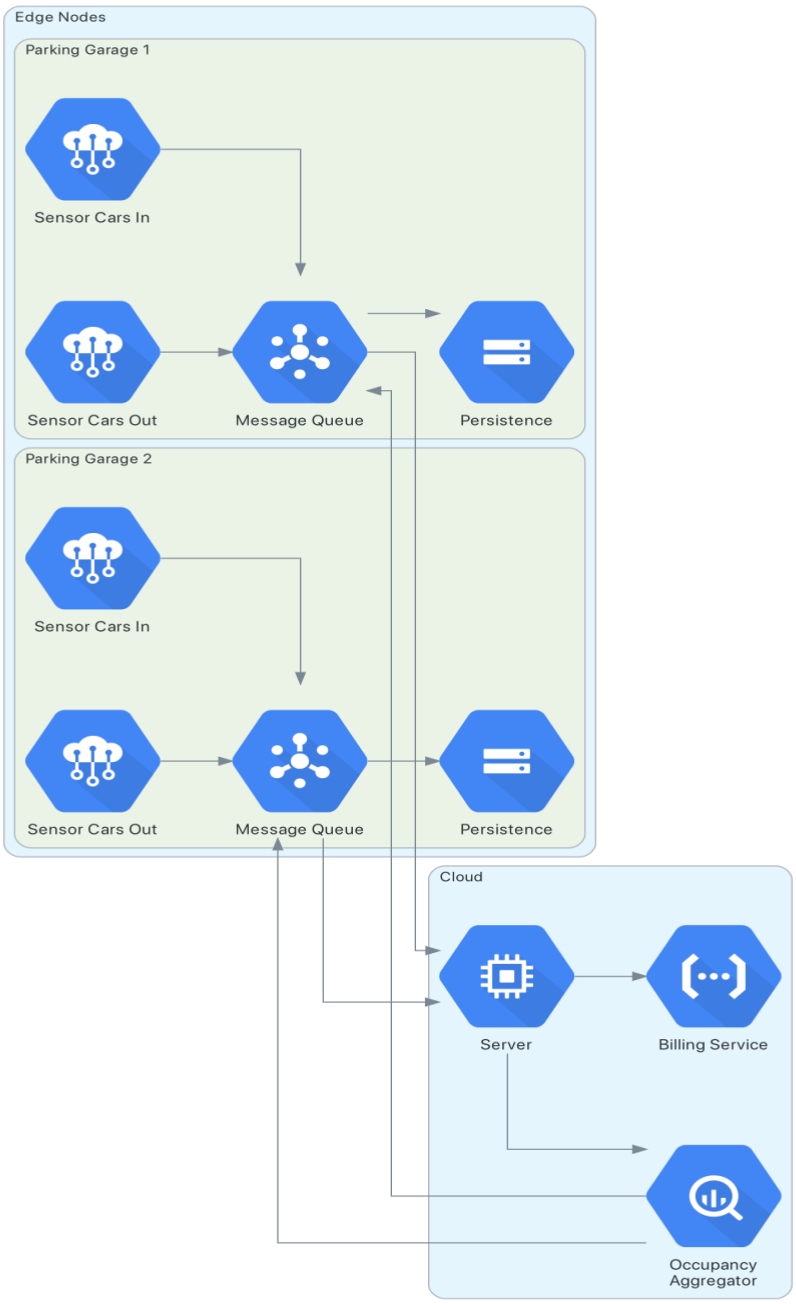


We simulate a network of parking garages across a city that are linked to a central cloud server to **a)** aggregate real-time occupancy data and **b)** handle billing based on the stay duration for each car.

TWO SENSORS: Car IN and Car OUT

There is an assumption that the cars parking at the garages are registered by their license plates which are mimicked as well in the project. The “virtual sensors IN and OUT” monitor the cars entering or leaving the specific garage(Edge Node), then the edge can calculate its own occupancy. Occupancy for all parking garages can be requested from the cloud server and displayed at each parking garage to show to the potential drivers.

Below there is a schematic overview of the system's architecture, with a simplified example of 2 edge nodes.



Reliable Messaging Prototype: Parking Garage

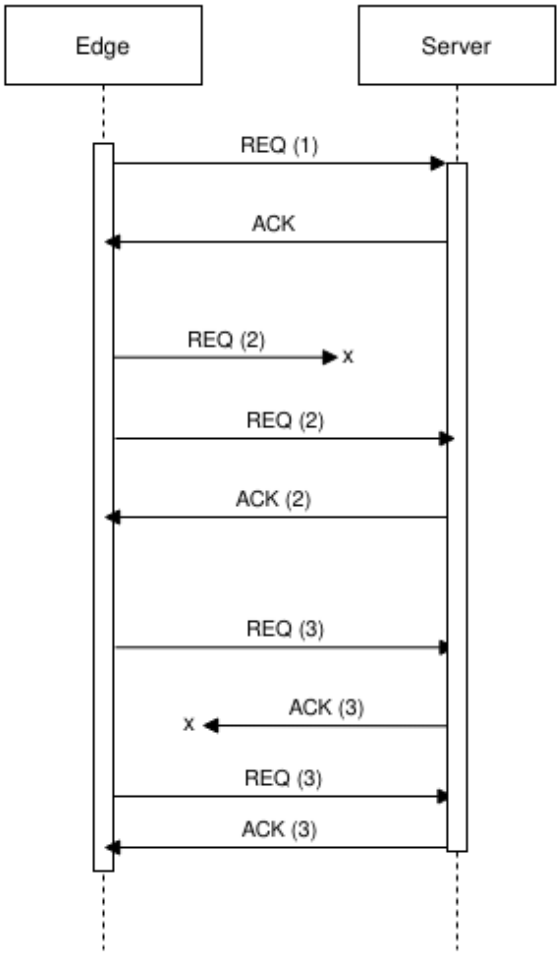
Reliable Messaging

We operate on a client-server model. There can be multiple clients (each as a parking garage) that we call edge nodes. They send data to a server (in the cloud). Data is processed and aggregated. At regular intervals clients request updates on the aggregated data.

Clients can fail or become unresponsive. The network does not guarantee order, timely delivery, or delivery at all. We implement a reliable delivery mechanism based on the client-side Lazy Pirate Pattern[1].

[1] https://zguide.zeromq.org/docs/chapter4/#Client-Side-Reliability-Lazy-Pirate-Pattern

This means the client is responsible for delivery. It awaits an acknowledgement for every message sent. If no ack is received the client may try to resend the message. Only an acknowledgement removes the message from the message queue. This way, different types of messages can be sent via the same mechanism. Generally, it is based on ZMQ's REQUEST-REPLY pattern, meaning every message exchange is to be initiated by the client. Clients can request data, the payload can be piggybacked on the acknowledgement.
In our prototype we simulate messages being dropped (see case 2 and 3 below). We make sure to only handle messages once (idempotency).



Algorithm
Client (Edge Node)

1. enqueue message (in background threads)
2. retrieve next message from queue (oldest eligible first, “FIFO”)
    a. however, wait minimum 2 seconds to resend a message
3. await acknowledgement
4. handle acknowledgement
    a. dequeue message
    b. process data (if applicable)

Server (Cloud)

1. receive message
2. handle message
3. send acknowledgment
    a. include requested data (if applicable)

Requirements check summary

1. 2 edge nodes are started from the local terminal, while the cloud server is deployed in GCE VM.
2. Sensors IN and OUT monitor the car entering and leaving for each edge node continuously as explained above, printed from the edge terminals.
3. Self occupancy, request for other edge occupancies, and car-leaving event messages are sent from edge to cloud; respective acknowledgment and occupancy data is sent back (upon request) from the cloud regularly in random time intervals.
4. While the server crashes, the clients keep their own message queue to be sent once the cloud comes back to reconnect; when one edge crashes, the server will keep listening while the sensors keep monitoring until it recovers to handle the message queue/resend the ones which haven't been acknowledged.