Documentation

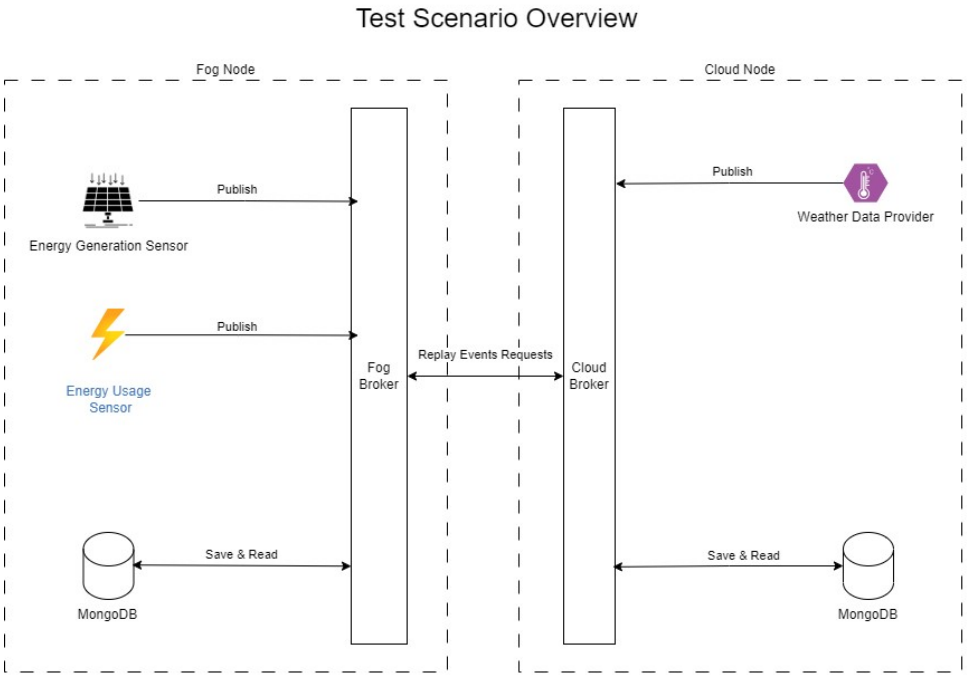
Reliable message delivery between fog and cloud

System Architecture Overview

The architecture of our system incorporates symmetric components deployed both in the cloud and on the local machine. Our aim is to ensure reliable message delivery by utilizing the ZeroMQ pub-sub pattern for internal node communication and the ZeroMQ request-response pattern for between-broker communication. The key components of our system include:

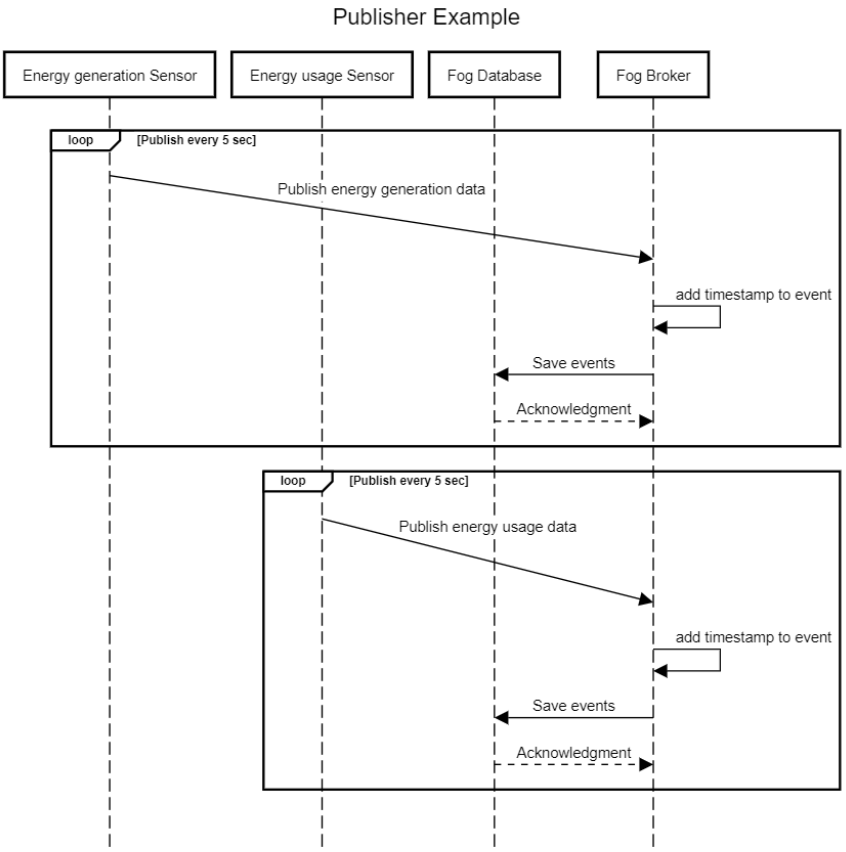
1. **Own Build Message-Broker**: Zeromq does not come with a broker, therefore we had to build our own one. It acts as a central hub, facilitating data exchange between local and cloud components. It efficiently distributes data from publishers to relevant subscribers within the LAN, ensuring reliable and asynchronous communication. Its source code can be found in src/replay_broker/replaybroker.py.
2. **Publisher Component**: The publisher exists in both the local and cloud components, continuously generating realistic data and publishing it to the broker for further processing and distribution. The code for the publishers can be found in src/sensor folder.
3. **Persistence Component**: Ensures data durability and availability by storing received data in a persistent storage mechanism, we use MongoDB. We introduced it since data preservation during system crashes is required. Its source code can be found in src/replay_broker/persistence.py.
4. **Subscriber Component**: Subscribes to specific topics or data streams of interest in both local and cloud components. It receives data from the broker, enabling various actions like analysis, storage, or visualization. Subscribers logic can be found in `src/subscriber` folder. However, in our test scenario, we do not incorporate any subscribers.

Test Scenario



In our example scenario, we have a fog node equipped with two sensors. One sensor is responsible for publishing energy usage data, while the other sensor publishes generated solar energy data. These sensor data are sent to the fog broker, which in turn saves them into a MongoDB database. Similarly, in the cloud node, we have a weather data provider that sends events to the cloud broker, which will save it into MongoDB. Both brokers can send regularly replay requests to each other to exchange newly aquired events. In the test scenario, cloud broker requests energy usage data, while fog requests the weather data.

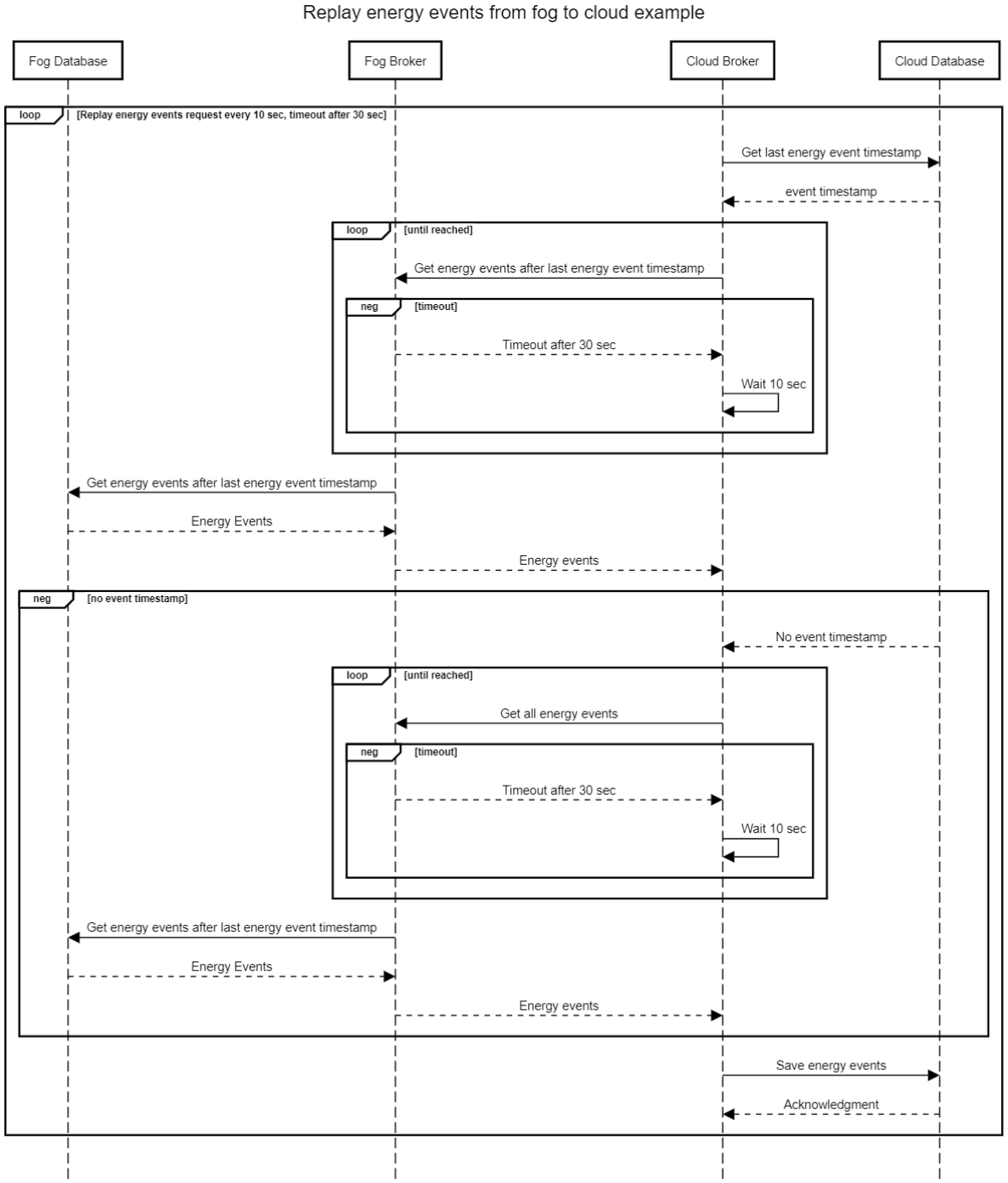
Internal node communication



Within a node, we use the pub-sub pattern for transmitting events from the sensor to the broker. An event refers to a piece of data or a message that is published by a sender and received by one or more subscribers. It represents a specific occurrence or update of information within a system. The broker adds a timestamp to the event and stores it in the MongoDB database. For example, an energy usage event could be:

```
{
  "timestamp":"2023-07-11 11:55:21.305232", // <- exact date and time when the event occurred
  "uuid":"0c7aa3f4-1fd1-11ee-b66f-00155da39be2", // <- unique identifier
  "name":"energy_usage", // <- event type
  "value":179.7 // <- value associated with energy usage
}
```

Between Broker Transmission and Reliable Message Delivery



To ensure synchronization between our local and cloud brokers, they regularly exchange replay requests. The replay request mechanism allows events to be replayed from one broker to another. When our broker initiates, it retrieves the most recent event received from the other broker by querying the local MongoDB database. Based on this information, the broker requests all subsequent events that occurred after the retrieved event. If there are no locally stored events, it requests the replay of all events from the other broker. In cases, where the other broker is not reachable, the request will timeout after 30 sec and retry the replay request until a connection is established. After a successful replay request, we store the new events in a MongoDB database. Together, these features empower our system to achieve reliable message delivery, minimizing the chances of data loss, and ensuring data integrity between the local and cloud broker.
The Reliable Message Delivery can be tested by simulating the shutdown of the nodes or by disconnection.

Deployment

In our deployment strategy, we utilize Terraform, an infrastructure as a code tool, to create and manage our cloud virtual machine on the google cloud platform and Docker to deploy the MongoDB database. In order to overcome the challenges of NAT traversal, we incorporate Tailscale into our system architecture to connect our brokers. The concrete instruction for deployment can be found in the readme.md

Appendix

Full Test Scenario Sequence Diagram