Fog Computing Prototype Documentation

1. Used Technologies **2**
- 1.1. Node.js Server with Typescript 2
- 1.2. Container Technologies with Docker 2
- 1.3. Google Cloud Engine (GCE) for deploying the Cloud Server 2
- 1.4. Communication between Local Server and Cloud Server 2
- 1.5. Terraform for setting up the GCE Infrastructure 3

2. Local Server Structure **3**

3. Cloud Server Structure **3**

1. Used Technologies

This section outlines the technologies employed in the development and deployment of the Prototype. The project is hosted at GitHub[1].

1.1. Node.js Server with Typescript

The Prototype utilizes a Node.js[2] server implemented in Typescript. Node.js provides a robust and efficient runtime environment for server-side JavaScript applications. Typescript[3] adds static typing and enhanced developer tooling to ensure code quality and maintainability.

It is recommended to use Node.js version 14.17 or higher for running the Prototype to ensure compatibility. The project is built on the node:14-alpine image, which provides a lightweight and efficient base for the application.

1.2. Container Technologies with Docker

Docker[4] is employed to facilitate containerization for the Prototype. Containerization allows for packaging the application and its dependencies into isolated containers, providing consistency and portability across different environments. Docker simplifies the deployment process and enables efficient resource utilization.

1.3. Google Cloud Engine (GCE) for deploying the Cloud Server

The cloud component of the Prototype is deployed on Google Compute Engine (GCE). GCE provides a reliable infrastructure for hosting cloud-based applications. Leveraging GCE ensures simplicity and efficient resource management for the cloud part of the project.

1.4. Communication between Local Server and Cloud Server

Communication between the local server and the cloud server of the Prototype is established via HTTP requests using a RESTful API. This approach enables seamless data exchange and integration between the local and cloud components of the application. It ensures efficient synchronization and real-time updates between the two environments.

---

[1] https://github.com/Al0ngsy/FC-Project
[2] https://nodejs.org
[3] https://www.typescriptlang.org
[4] https://www.docker.com

1.5. Terraform for setting up the GCE Infrastructure

Terraform[5] is an open-source infrastructure as code (IaC) software tool created by HashiCorp. It allows developers to define and provision data center infrastructure using a declarative configuration language. Terraform is cloud-agnostic, supports a multitude of providers like AWS, Google Cloud, Azure, and more, and handles dependencies between resources, enabling complex changesets to be applied to your infrastructure in a safe and predictable way.

2. Local Server Structure

The local server in the Prototype is designed to mimic an edge server and handle data from multiple sensors. To replicate the behavior of an edge server, the local server spawns two child node environments using the worker_thread module. Each child node environment simulates a sensor.

Communication between the local server and the simulated sensors is based on an event-based messaging pattern. When a simulated sensor has data to send, it emits a 'message' event. The local server listens for these events and executes a provided callback function to handle the received data. The data sent by the sensors includes essential information such as the sensorId and sensorName for identification purposes. Additionally, random data of 1000 bytes is included to simulate the sensor readings.

Upon receiving a message, the local server assigns a uniqueId and timestamp to the data. The timestamp consists of the data creation time (dataTime) and the scheduled time for the next send attempt (nextSendTime). The data is then cached in memory for further processing.

To transmit the sensor data to the cloud server, the local server sends HTTP POST requests at regular intervals. The interval between successive requests is set to 1 second after the previous send requests were completed.

If a data entry is successfully sent to the cloud server for processing, the response will include the processing result and a status of 2xx for the POST request. It will then be removed from the cache. However, if sending fails, the data entry is placed back into the cache with an exponentially increasing nextSendTime value. The exponential increase is calculated as 2^retriesAmount multiplied by the backoffTime. This ensures a progressively longer delay between send attempts, allowing for potential network or cloud server issues to resolve.

3. Cloud Server Structure

The cloud server emulates a deployed microservice that includes a RESTful POST endpoint. This endpoint is designed to receive requests from edge servers. When a request arrives, the data is checked for a unique identifier and then saved in a 'DB'. Regardless of whether

[5] https://www.terraform.io/

the data already exists in the 'DB', it is processed. The processed result is then sent back as a response to the HTTP request.

To demonstrate the reliability of the prototype in handling potential issues during the communication process, we have implemented a "chaos monkey" middleware at both the beginning and end of the data receiving endpoints. This middleware purposely introduces disruptions either in the incoming request or the returned response. This is to highlight the ability of the prototype to cope with unexpected problems that may arise during the communication process.

The server is instantiated within a Docker container, which resides on a Google Compute Engine (GCE) Virtual Machine (VM). This VM is provisioned using Terraform, while the container's launch is orchestrated through the gce-container-declaration, facilitating automatic container startup upon VM initiation.