Prototyping assignment documentation

Prototyping 12

Our Fog application [1] is utilized to control the intensity of a motor, based on the motor's current temperature. A request-reply pattern is being used with some modifications. Typically, in a request-reply pattern, a client (request socket) sends a request message to a server (reply socket) and waits for a response. However, in our case, if the reply socket does not send a response, the request socket checks for a reply multiple times instead of blocking and waiting indefinitely[2].

1 Local component

To develop our application, we have a local component, that runs on our own machine, consisting of two Python files that act as virtual sensors. We designed each virtual sensor to be specifically responsible for monitoring the temperature of one motor. These files generate random temperature values within the range of 50 to 300, representing the motor's temperature. Every 10 seconds, a new value is generated and saved to a CSV file. For each virtual sensor, there is a corresponding client that transmits the collected data to our cloud component 2 at regular 15-second intervals. This transmission process occurs separately and independently from the other clients, ensuring each virtual sensor operates autonomously. The client sends a data tuple to the server in the following format: *(pid, current time, sensor ID, motor temperature)*. The tuple consists of a process identifier; a timestamp of when the data is sent from the client; a sensor ID that identifies which sensor the data came from; and the engine temperature.

2 Cloud component

Our cloud component runs on a virtual machine on Google Compute Engine. The VM is an e2-medium instance with an Ubuntu 20.04.6 LTS operating system and an external IP address 34.141.3.45. This component communicates with the local component through the TCP protocol. Therefore, a firewall rule that allows the port number 5554 is necessary in the network configuration of the above-mentioned VM.
This component functions as a server, receiving information from the clients and sending separate responses to each client. The server generates a random number to simulate the weather temperature at the local component's location. Based on the temperature, the cloud component determines a maximum threshold value. If the current temperature of a motor exceeds this threshold, the motor intensity is decreased. Conversely, if the motor's temperature is less than the maximum threshold, the motor intensity is slightly increased. The server sends a data tuple to the client in the following format: *(maximum threshold, intensity change, current time)*. The tuple includes the maximum threshold value, which defines the temperature limit for the motor; the intensity change value, indicating the adjustment required for the motor's intensity; and the current time to provide timestamp information.

3 Requirements

The interconnection between the components is designed to handle server crashes effectively. In the event of a server crash, the client implementation incorporates a retry mechanism, attempting to reconnect three times. If the server remains unresponsive after the third attempt, the client introduces a 60-second pause in the reconnection process. It is important to note that during this reconnection pause, the virtual sensor (Python file) responsible for generating temperature values continues its operation independently. The virtual sensor is not affected by the client-server reconnection process and continues generating temperature values as expected. When the server becomes available again, the client follows a specific protocol. Firstly, it sends the last data tuple it intended to transmit before the connection was lost. Afterward, the client proceeds to send all the remaining data tuples from the CSV file in the order they were generated in the queue. The client receives respective replies from the server for each data tuple sent. In the case of a client crash, it is important to note that other clients remain unaffected as they operate autonomously. Similarly, the virtual sensor, responsible for generating temperature data, continues its operation uninterrupted. When the client is up and running again, it resumes its functionality by starting with the first element in the queue.

4 Packages and Usage

In our project, we used the Python programming language along with the ZeroMQ [3] messaging library for the implementation. To successfully run the application you can refer to the `README.md` file[4] in our public GitHub repository for the necessary requirements and instructions.

[1] `https://github.com/Ouassim31/fc-assignment-ss23`
[2] `https://zguide.zeromq.org/docs/chapter4/#Client-Side-Reliability-Lazy-Pirate-Pattern`
[3] `https://zeromq.org/`
[4] `https://github.com/Ouassim31/fc-assignment-ss23/blob/main/README.md`