Prototyping Assignment

Fog-like application for plant care

Repository link: https://github.com/1ucket/fog21

For our prototyping assignment, we decided on a plant care application that measures temperature, brightness, humidity, and moisture level via sensors. The edge nodes measure and pre-process the data and send it to the cloud component, which stores it and sends instructions back to the clients.

Client

Every client is an instance of a class which contains sensors, an ID, a name and a location (e.g. a compartment in the greenhouse). The sensors have methods to return the current data. This is currently implemented as a mock using random values with a gaussian distribution around a reference value with respect to the previous values.

Two functions are called periodically by the main loop with a default interval of 1000ms:

collectData()

This function reads the sensor values and stores them in a list. It includes the sensor, the value and a time stamp.

sendData()

This function will send the data from the list to the cloud component via a TCP connection. If no connection is currently open it tries to establish a new connection. On a successful send the data is removed from the list, otherwise it will be kept and stored for the next iteration until transmitted successfully. The function also handles incoming data from the cloud component if available.

Both functions above are independent from each other and don't need to be called in the same interval. On a connection loss the data collection will continue undisturbed and the data will be transmitted as soon as a new connection was established.

Server

For the server side, we created a virtual machine instance on Google Compute Engine to simulate our cloud component (Fig. 1). TCP is used for transmission (Fig. 2).

INSTANZEN INSTANZZEITPLAN

Filter Name oder Wert des Attributs eingeben

☐	Status	Name ↑	Zone	Empfehlungen	Verwendet von	Interne IP	Externe IP-Adresse	Verbinden
☐	✔	fc-1	europe-west3-c			10.156.0.2 (nic0)	35.198.79.71	RDP

Figure 1: VM-Instance in GCE

☐	Name	Typ	Ziele	Filter	Protokolle/Ports	Aktion	Priorität
☐	default-allow-http	Eingehend	http-server	IP-Bereiche: 0.0.0.0/0	tcp:80	Zulassen	800

Figure 2: Firewall rule for instance to allow incoming TCP traffic on port 80

The Server.java class will establish the connection, obtain the streams of an incoming client, and create a client handler object. The while loop assures that incoming connections are continuously accepted (Fig. 3).

```java
// server is listening on port 80
InetAddress ip = InetAddress.getByName("10.156.0.2");
ServerSocket ss = new ServerSocket(80, 50, ip);
Storage storage = new Storage();

// running infinite loop for getting
// client request
while (true)
{
    Socket s = null;

    try
    {
        // socket object to receive incoming client requests
        System.out.println(ss.getInetAddress());
        System.out.println(ss.getLocalPort());
        s = ss.accept();

        System.out.println("A new client is connected : " + s);

        // obtaining input and out streams
        DataInputStream dis = new DataInputStream(s.getInputStream());
        DataOutputStream dos = new DataOutputStream(s.getOutputStream());

        // create a new thread object
        Thread t = new ClientHandler(s, dis, dos, storage);

        // Invoking the start() method
        t.start();
```

Figure 3: Establishing connections and creating handler objects for incoming clients

Receiving and processing new data

Upon connection, the client handler checks if there is any data from previous sessions of this client that needs to be sent first. If not, the client handler reads the information from the input stream. This data is then parsed into a map with the help of a custom container. Afterwards, it gets passed to a new data handling object (Fig. 4).

```java
// start processing data
JSONObject answerJSON = new JSONObject();

// from: https://www.tutorialspoint.com/json_simple/json_simple_container_factory.htm
JSONParser parser = new JSONParser();
ContainerFactory containerFactory = new ContainerFactory() {
    @Override
    public Map createObjectContainer() {
        return new LinkedHashMap<>();
    }

    @Override
    public List creatArrayContainer() {
        return new LinkedList<>();
    }
};
try {
    Map map = (Map) parser.parse(received, containerFactory);

    // compare received data and check if edge has to take action
    DataHandling dh = new DataHandling(map);
    Map answer = dh.handleData();
    JSONObject answ = createJSON(map, answer, answerJSON);
```

Figure 4: Creating a map for JSON object

To proceed with the data processing, it is first determined which type of data was sent (temperature, brightness, humidity or moisture level). Depending on that, the compareData()-method then compares the passed value from the sensor with the optimum value (Fig. 5).

```java
    // get the type of sensor data
    public Map handleData() {
        String type = map.get("type").toString();
        switch (type) {
            case "TEMPERATURE":
                answer = compareData(OPT_TEMP);
                break;
            case "MOISTURE":
                answer = compareData(OPT_MOIST);
                break;
            case "HUMIDITY":
                answer = compareData(OPT_HUMID);
                break;
            case "BRIGHTNESS":
                answer = compareData(OPT_BRGT);
                break;
            default:
                break;
        }
        return answer;
    }
```

```java
    // compare sensor value with optimum value and decide what action needs to be taken
    private Map compareData(double opt) {
        String measuredString = map.get("value").toString();
        double measured = Double.parseDouble(measuredString);
        double gap = opt - measured;
        if (gap > 0) {
            command = "raise";
        } else if (gap < 0) {
            command = "lower";
        } else {
            command = "ok";
        }
        answer.put("command", command); // add action to answer object
        answer.put("gap", gap); // add value that shows how far off the sent value is from the optimum
        return answer;
    }

}
```

Figure 5: Processing data from client

At last, two further key/value pairs are added to the new object: One with a command ("rise", if measured value is too low; "lower", if measured value is too high; or "ok", if measured value corresponds to optimum value), and one with the numerical difference between the measured and the optimum value.

Sending data back to the client

The new extended JSON object can now be sent back to client. Before it gets written into the output stream, it is put in a storage, in case the connection will break off before the data can reach the client. All objects also get a timestamp and the information if the object is from a previous session or not (Fig. 6). Upon successful arrival, the client will send back an ok-message.

```java
// send data back to edge as String
answ.put("prev", false);
String time = Long.toString(System.currentTimeMillis());
answ.put("timestamp", time);
System.out.println(answ);

storage.putSendOff(answ); // save new JSON in storage

dos.writeUTF(answ.toString());
```

Figure 6: Storing all objects and sending data back to the client

Sending data from previous sessions after connection loss

In case the client crashes, the client handler will do three things: (1) Store the IP address of the client, (2) store the number of successfully transmitted objects (if the client sends an ok-message, a counter will go up), and (3) close the connection. If the client reconnects now, the client handler will know that there is still data from a previous session by checking if the IP address is already in the storage.

```java
            if (firstCheck) { // only needs to be checked on first loop
                sendPreviousData();
            }
            firstCheck = false;
```

```java
    private void sendPreviousData() {
        // make sure there is no data left that needs to be sent out first before receiving new one
        try {
            unavailable = storage.getUnavailable();
            // check if edge was previously connected but lost connection
            if (unavailable.contains(ip)) {
                sendOff = storage.getSendOff();
                // check that there are JSON objects in the storage from previous session
                if (!(sendOff.isEmpty())) {
                    sent = storage.getSentVariable();
                    System.out.println("SENT momentan: " + sent);
                    // all objects created minus the objects that were successfully sent off = objects that previously couldn't be sent to edge
                    for (int i = 0; i < sent; i++) {
                        storage.removeFirst(); // the last objects created are the ones that weren't sent, thus remove all the ones from storage that were successfully sent
                        System.out.println("Groesse:" + storage.getSendOff().size());
                    }
                    // send all previous objects that weren't sent to edge
                    for (JSONObject j : storage.getSendOff()) {
                        System.out.println("Alte Objekte:" + storage.getSendOff().size());
                        j.put("prev", true); // tell edge that this is data from a previous session
                        String time = Long.toString(System.currentTimeMillis());
                        j.put("timestamp", time);
                        dos.writeUTF(j.toString());
                    }
                    storage.clearSendOff(); // empty storage so objects from current session can be stored
                    sent = 0;
                }
            }
            storage.clearUnavailable(); // remove ip address from storage, since all data has been transmitted now
        } catch (IOException e) {
            handleException();
        }
    }
```

Figure 7: Sending data from previous sessions that did not arrive at the client

Therefore, it will call the sendPreviousData()-method (Fig. 7). Since the storage has all objects created in the previous session, and the number of objects that actually arrived, we can subtract them from each other and get the number of objects, that did not arrive (If that number is e.g., 4, we need the last four objects from the total objects list). These objects are now sent off to the client (the info that this is a message from a previous session is added as well). Afterwards, the list of all objects and the counter for successfully sent objects are cleared/set back to 0 so that they are reset for the new session. The IP address is removed from the list of crashed clients as well. Now, the client handler starts reading from the input stream.