Documentation report of the fog prototype

June 27, 2023

This report serves as a comprehensive documentation of the code available in the public GitHub repository located at https://github.com/hassan1595/Fog-Computing-Prototype. The repository contains a collection of code files and resources related to a fog computing prototype. In this report, we provide detailed information about the codebase which consists mainly of two entities client and server. The client's code can be found in `local/client.py`, the server's code can be found in `cloud/server.py` and the dependencies logic can be found in `lib/`

Message pattern

In this prototype, we will employ the Dealer/Router message pattern through the ZeroMQ library, which facilitates asynchronous communication between two entities. This message pattern is straightforward and simple so we will enhance its reliability by introducing robust message delivery methods. These methods will be thoroughly explained in the accompanying documentation to provide a comprehensive understanding.

1 Client

The client periodically produces sensor data and transmits it to the cloud server. The server is responsible for performing a dimensionality reduction technique known as PCA on the received data and returning the results to the client. The client then plots the result and saves the plot locally for future examination. To ensure dependable communication, the client's code incorporates an acknowledgment system for both ends of the communication. The client's code is explained in the following sections.

1.1 Parameters

The client has 5 parameters as shown in Fig. 1 that can be adjusted depending on the use case.

1. **HOST** The IP address of the cloud server that the client sends the data to.

```python
# parameters
HOST = "34.107.5.115"  # The server's IP address
PORT = 4000 # The port used by the server
TIMEOUT = 10 # Time out in seconds before a messeage is retransmitted
buffer_max_size = 100 # maximum buffer size to store messages that not yet received by other components
size = 50 # size of the sent data (sensor data)
sleep_time = 3 # sleep time of the client before generating the next sensor data
```

Figure 1: adjustable parameters of the client

2. **PORT** The port that the cloud servers listens to for incoming data.
3. **TIMEOUT** period of time in seconds that the client has to wait after sending data to the cloud server before retransmitting it.
4. **buffer_max_size** The maximum size of the buffers that the client uses to store unacknowledged data in case of re transmission. If the data is acknowledged the client deletes it from the buffers.
5. **size** The size of the data that the client generates for each sensor.
6. **sleep_time** The time in seconds that the client has to wait before generating the next sensor data.

1.2 Sensor data generator

```python
def simulator_sensors(size, sleep_time = 1):
    """
    generates sensor data indefinatley

    :size: size of the sensor data
    :sleep_time: sleep time before generating the next sensor data

    """
    while(True):
        temperature = np.random.uniform(10, 40, size)  # Simulated temperature in Celsius
        humidity = np.random.uniform(20, 70, size)     # Simulated humidity in relative humidity
        wind = np.random.uniform(10, 150, size)     # Simulated wind speed in km/h
        pressure = np.random.uniform(80, 150, size) # Simulated pressure in Pa
        yield np.array([temperature, humidity, wind, pressure]).T
        print(f'\ngenerated {size} sensor data for temperature, humidity, wind and pressure\n')
        time.sleep(sleep_time)
```

Figure 2: Sensor data generator method

As shown in Fig. 2 the client uses a method that indefinitely generates simulated sensor data. It generates data for 4 simulated environmental sensors which are temperature, humidity, wind, pressure. The method waits sleep_time seconds before generating the next sensor data.

1.3 Buffers

The client creates in Fig. 3 two buffers to store data and the time of sending it temporarily in, in case of retransmission. The two buffers are

- **buffer** stores the data and its id temporarily in case of retransmission.
- **acks** stores the id of the data and the time at which the data was sent temporarily in case of retransmission.

```
18  # buffer to store sent messages in case of retransmission
19  buffer = []
20  # store the awaited id of the acknowledgments
21  acks = []
```

Figure 3: Client creates buffers

1.4 Plots folder

```
43  # recreate folder to store plots
44  shutil.rmtree('plots', ignore_errors= True)
45  os.makedirs("plots")
```

Figure 4: Creating a new Folder to save plots

The client creates a new folder in which the plots can later be saved as shown in Fig. 4.

1.5 Socket

```
49  context = zmq.Context()
50  client_socket = context.socket(zmq.DEALER)
51  client_socket.connect("tcp://{}:{}".format(HOST, PORT))
52  poll = zmq.Poller()
53  poll.register(client_socket, zmq.POLLIN)
54  counter = 0
```

Figure 5: Initialising a socket, poller and a counter

The client creates a dealer socket to implement the dealer/router message pattern, enabling asynchronous message transmission to the server. Additionally, a poller is initialized to monitor incoming messages from the server. Furthermore, the client establishes a counter that serves as a unique identifier for the messages sent from the client.The code for that is shown in Fig. 5

1.6 Client preprocesses and sends data

As shown in Fig. 6 the client continuously generates data, and for each data point, it performs preprocessing by centering and standardizing it. The preprocessed data is then converted into a byte object. To facilitate potential retransmissions, the client saves the preprocessed data along with its corresponding ID and time at which it was sent. The client also manages the buffers to ensure its size does not exceed buffer_max_size. If the buffers reaches its maximum capacity, the client overwrites the earliest added value to accommodate new data. Additionally, the client sends the preprocessed data to the server and increments the counter, taking care not to exceed the buffer_max_size.

```python
for data in simulator_sensors(size, sleep_time):

    #  preprocess data
    data_preprocessed = (data - data.mean(axis = 0))/data.std(axis = 0)
    full_data = {"id": counter , "data" : data_preprocessed}
    full_data_b = pk.dumps(full_data)
    # save in buffer
    buffer.append({"id": counter, "data":full_data_b})
    acks.append({"id" : counter, "time": time.time() })

    # overwrite buffer if reached its maximum size
    if len(buffer) > buffer_max_size:
        buffer.pop(0)
        acks.pop(0)


    # sent data with its id to server
    client_socket.send(full_data_b)
    print(f'\nsent data of id {counter}\n')
    counter = (counter +1) % buffer_max_size
```

Figure 6: Client reprocesses and sends data

1.7 Client polls messages

```python
    # polls to check received messages
    sockets = dict(poll.poll(1000))
    if client_socket in sockets:
        if sockets[client_socket] == zmq.POLLIN:
            obj_b = client_socket.recv()
            obj = pk.loads(obj_b)
```

Figure 7: Client polls messages

As shown in Fig. 14 client polls to check for received messages from server. If so the client then reads the sent byte object and converts it into a readable object. The server can send the client only on the the following messages.

- **ack** The server acknowledges the client that the sent data was received.
- **result** The server sends the client the result of the sent data.

1.8 Client receives ack from server

```python
            # server sent ack to state they received data.
            if "ack" in obj:
                print(f'\ngot ack of id {obj["ack"]}\n')

                # delete data from buffer because server already received it
                acks = [ack for ack in acks if ack["id"] != obj["ack"]]
                buffer = [elem for elem in buffer if elem["id"] != obj["ack"]]
```

Figure 8: Client receives ack from server

Fig. 8 illustrates the process by which the client manages acknowledgments from the server. Upon receiving an acknowledgment from the server, the client removes the corresponding data from its buffers. This action is taken by the client based on the knowledge that the server has successfully received the data.

1.9 Client receives result from server

```
95              # server sent result
96              elif "result" in obj:
97                  print(f'\ngot result of id {obj["id"]}\n')
98                  # sent ack to server
99                  client_socket.send(pk.dumps({"ack" : obj["id"]}))
100
101                 # plots result and save it
102                 fig, ax = plt.subplots()
103                 ax.scatter(*obj["result"].T, c = "red")
104                 ax.set_title("Dimentionality reduction using PCA", fontsize = 17)
105                 ax.set_xlabel(f'{obj["pca"][0][0]:.2f}*temperature + {obj["pca"][0][1]:.2f}*humidity + \
106                     {obj["pca"][0][2]:.2f}*wind + {obj["pca"][0][3]:.2f}*pressure', fontsize = 10)
107                 ax.set_ylabel(f'{obj["pca"][1][0]:.2f}*temperature + {obj["pca"][1][1]:.2f}*humidity + \
108                     {obj["pca"][1][2]:.2f}*wind + {obj["pca"][1][3]:.2f}*pressure', fontsize = 10)
109                 ax.grid()
110                 fig.savefig(f'plots/plot_{obj["id"]}.png')
111                 plt.close()
```

Figure 9: Client receives result from server

As shown in Fig. 9, if the client receives a result from the server, it creates a plot to visualize the results and then saves the plot in the specified folder for later evaluation. The client also sends an acknowledgment to the server, indicating that it has successfully received the data.

1.10 Client retransmitts message

```
113     # check for time out of the latest message and retransmit if necessary
114     if len(acks):
115         if  time.time() - acks[0]["time"]   > TIMEOUT:
116             client_socket.send(buffer[0]["data"])
117             print(f'\nretranssmited result with id  {buffer[0]["id"]}\n')
118             ack = acks.pop(0)
119             ack["time"] = time.time()
120             acks.append(ack)
121             buffer.append(buffer.pop(0))
```

Figure 10: Client retransmitts data if time out happened

The client periodically evaluates the need for data retransmission, as depicted in Figure 10. This evaluation involves checking if the earliest unacknowledged data has exceeded the designated timeout. If the timeout is exceeded, the client infers that the server did not receive the message and initiates a retransmission process using data stored in the buffers. The client then updates the time at which the data was sent and adds it to the end of the queue.

2 Server

The server is designed to efficiently handle client requests while ensuring reliability. It serves as a central hub for receiving data from clients and performing dimensionality reduction using the PCA algorithm imported from the lib/pca.py library. After the reduction process, the server promptly sends the transformed data back to the client. To ensure secure communication, the server implements

an acknowledgment system that confirms the successful delivery of messages between the client and server. The server component is meant to be run in the cloud. The server's code is explained in the following sections.

2.1 Parameters

```
11 # parameters
12 HOST = "0.0.0.0" # accepts connections from all clients
13 PORT = 4000  # The port used by the server
14 TIMEOUT = 10 # Time out in seconds before a messeage is retransmitted
15 buffer_max_size = 100 # maximum buffer size to store messages that not yet received by other components
16 sleep_time = 0.5 # sleep time of the server before the next message poll
```

Figure 11: adjustable parameters of the server

As for the client, the server has parameters that can be adjusted as shown in fig"11

1. **HOST** The IP address that the server listens to typically 0.0.0.0 to listen to all clients.
2. **PORT** The port that the server listens to.
3. **TIMEOUT** period of time in seconds that the server has to wait after sending result to the client before retransmitting it.
4. **buffer_max_size** The maximum size of the buffers that the server uses to store unacknowledged results in case of re transmission. If the result is acknowledged the client deletes it from the buffers.
5. **sleep_time** The time in seconds that the client has to wait before generating the next sensor data.

2.2 Buffers

```
18 # buffer to store sent messages in case of retransmission
19 buffer = []
20 # store the awaited id of the acknowledgments
21 acks = []
```

Figure 12: Server creates buffers

The server creates as in Fig. 12 two buffers to store results temporarily and the time of sending it in, in case of retransmission. The two buffers are

- **buffer** stores the results as a and its id as a queue temporarily in case of retransmission.
- **acks** stores the id of the data and the time at which the results was sent a queue temporarily in case of retransmission.

```
24 context = zmq.Context()
25 server_socket = context.socket(zmq.ROUTER)
26 server_socket.bind("tcp://{}:{}".format(HOST, PORT))
27 poll = zmq.Poller()
28 poll.register(server_socket, zmq.POLLIN)
```

Figure 13: Initialising a socket and poller

2.3 Socket

The server creates a router socket to implement the dealer/router message pattern, so the server can receive asynchronously sent messages from the client . Additionally, a poller is initialized to monitor incoming messages. The code for that is shown in Fig. 5

2.4 Server polls messages

```
30 while True:
31
32     #  polls to check received messages
33     sockets = dict(poll.poll(1000))
34     if server_socket in sockets:
35         if sockets[server_socket] == zmq.POLLIN:
36             _id = server_socket.recv()
37             obj_b = server_socket.recv()
38             obj = pk.loads(obj_b)
```

Figure 14: Server polls messages

As shown in Fig. 14 server enters the cycle where it polls to check for received messages from client. If so the client's id and as well as the sent byte object is read then the byte object is converted into a readable object. The client can send the server only on the the following messages.

- **data** The client sends the sensor data to the server and expects its result.
- **ack** The server sends the client the result of the sent data.

2.5 Server receives data

As shown in Fig. 15, if the server receives data from the client, it saves first the data, its id, the id of the client and the time at which the data was sent. That is essential if later retransmission is required. The server then sends an acknowledgment to the client to notifying that the server has received the data. Last but not least, the server applies the imported dimentiality reduction technique (PCA) on the sent data and sends the result back to the client.

```
40          # client sent data
41          if "data" in obj:
42
43              print(f'\nrecieved data with id {obj["id"]} from client {_id}\n')
44
45              # add result to buffer in case of retransmission
46              buffer.append({"id" : obj["id"], "data": obj["data"], "socket_id": _id})
47              acks.append({"id" : obj["id"], "time": time.time()})
48
49              # sent ack to client
50              ack_obj_b = pk.dumps({"ack" : obj["id"]})
51              server_socket.send(_id, zmq.SNDMORE)
52              server_socket.send(ack_obj_b)
53
54
55              # apply pca to data
56              pca = PCA(obj["data"])
57              result = pca.project(obj["data"],2)
58              result_b = pk.dumps({"id": obj["id"], "result": result, "pca": pca.U[:,:2].T})
59
60              # send result to client
61              server_socket.send(_id, zmq.SNDMORE)
62              server_socket.send(result_b)
63              print(f'\nsent result with id {obj["id"]} to client {_id}\n')
```

Figure 15: Server receives data

```
65              # client sent ack to state they received a result
66              elif "ack" in obj:
67                  print(f'\ngot ack of id {obj["ack"]} from client {_id}\n')
68
69                  # delete result from buffer because client already received it
70                  acks = [ack for ack in acks if ack["id"] != obj["ack"]]
71                  buffer = [elem for elem in buffer if elem["id"] != obj["ack"]]
```

Figure 16: Server receives ack from client

2.6 Server receives ack

Fig. 16 shows how the client manages acknowledgments from the server. Upon receiving an acknowledgment from the client, the server removes the corresponding result from its buffers. This action is taken by the server based on the knowledge that the client has successfully received the result.

2.7 Server retransmitts message

```
74      # check for time out of the latest message and retranssmit if necessary
75      if len(acks):
76          if  time.time() - acks[0]["time"]  > TIMEOUT:
77
78              print("\ntime out happened\n")
79              pca = PCA(buffer[0]["data"])
80              result = pca.project(buffer[0]["data"],2)
81              result_b = pk.dumps({"id": buffer[0]["id"], "result": result, "pca": pca.U[:,:2].T})
82              server_socket.send(buffer[0]["socket_id"], zmq.SNDMORE)
83              server_socket.send(result_b)
84              print(f'\nretranssmited result with id {buffer[0]["id"]} to client {buffer[0]["socket_id"]}')
85              ack = acks.pop(0)
86              ack["time"] = time.time()
87              acks.append(ack)
88              buffer.append(buffer.pop(0))
89
90      time.sleep(sleep_time)
```

Figure 17: Server retransmitts data if time out happened

As shown in Figure 17, the server regularly assesses the requirement for data retransmission. This assessment involves examining whether the earliest unacknowledged result has surpassed the specified timeout duration. If the timeout is exceeded, the server proceeds with a retransmission process utilizing

the data stored in the buffers. The server then updates the time at which the result was sent and adds it to the end of the queue.

3 Reliable message delivery

The communication between the server and the client is implemented using an acknowledgement system to ensure reliable message delivery between the two entities. The communication is therefore robust and fault tolerant in many different scenarios such as the following.

1. **Client did not receive data from server** The client defines a time-out time and checks if each sent data is acknowledged in that duration. So in that scenario the client will retransmit the data after the time out happens and will continue retransmitting it periodically until the server responds.
2. **Server did not receive data from client** The server as well defines a time-out and will retransmit the result to client after the time-out happens and will continue retransmitting it periodically until the client responds.
3. **Client crashed** If the client for any reason disconnected, the server will save the result in buffers and will retransmit it periodically to the client until the client acknowledges it. The server will not stop working and producing result in the mean time.
4. **Server crashed** If the server for any reason disconnected or crashed, the client will save the generated data in buffers and will retransmit it periodically to the server until the server acknowledges it. The client will not stop working and generating data in the mean time.

4 Requirements

The implemented prototype successfully meets the specified requirements. A demonstration video showcasing its functionality can be found in the associated GitHub repository.

1. The application comprises a local component that runs on one's own machine, and a component running in the Cloud, e.g., on GCE.
2. The local component collects and makes use of (simulated) environmental information. For this purpose, four virtual sensors that continuously generate realistic data are used.
3. Data is transmitted regularly (multiple times a minute) between the local component and the Cloud component in both directions.
4. When disconnected and/or crashed, the local and Cloud component keeps working while preserving data for later transmission. Upon reconnection, the queued data needs to be delivered.