Coen 424 Assignment 1

Siddhesh Mishra
Computer Engineering

Concordia University

Montreal, Canada

Siddhesh_mishra15@hotmail.com

Introduction

In this assignment, the task was to practice the concepts, and techniques for data models and the communications for resources represented by data models.

I. Data Model

There are two different data models in this assignment as there are two different type of data storage.

The first type of storage is JSON. JSON is a lightweight data interchange format that is used for storing and exchanging data. A great advantage of JSON is that it is language independent, and it is easy to understand. JSON data is in the format of key/value pair, it has a data model of a document database as shown in Figure 1 below.

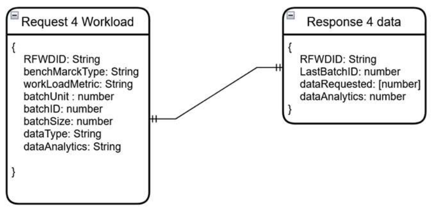
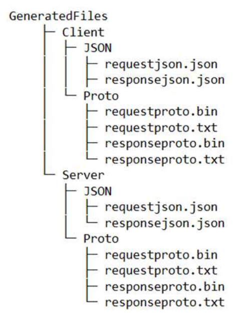


Figure 1: JSON data model

As shown in figure 1, the user will send a request to the server with the different parameters, the request ID, the benchmark type, the workload metric, batch unit, the batch id, batch size, data type and the type of data analytics they wish to get. The server, once have received and decoded the request, will be responding the user with a response for workload which contains the response id, the last batch id being sent, an array of all the data requested, and the data analytic value. The request being sent from the client is saved in a file called requestjson.json. The server, after having processed the request sends back a response which is saved in a JSON file called responsejson.json. The file structure is as shown in figure 2.



```
GeneratedFiles
   ├ Client
   │  ├ JSON
   │  │  ├ requestjson.json
   │  │  ├ responsejson.json
   │  └ Proto
   │     ├ requestproto.bin
   │     ├ requestproto.txt
   │     ├ responseproto.bin
   │     └ responseproto.txt
   └ Server
      ├ JSON
      │  ├ requestjson.json
      │  └ responsejson.json
      └ Proto
         ├ requestproto.bin
         ├ requestproto.txt
         ├ responseproto.bin
         └ responseproto.txt
```

Figure 2: File structure of the request and response files

The figure 2 shows the file structure for both the proto and JSON format files. The reason for both the server and client side generates a request and response files is to allow the user to compare both requests and responses to make sure they are both the same and no data is lost in the transfer.

The second type of storage used is protocol buffers format. As google developers described it, Protocol buffers are language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward-compatible and backward-compatible way. Some advantages of using of using proto buffers are they are a compact data storage, fast parsing, language independent and is optimized functionality through automatically generated classes [1]. The figure 3 shows the data model for the protocol buffer file.

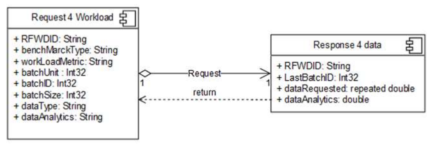


Figure 3: Protocol Buffer Data Model Design

The figure 3 shows how the data interacts with each other. On the left-hand side, we see the request file that is being sent from the client side. The data is then received by the server where it parses the data, de-serializes it, processes the data. The processed data is then sent back to the client side in the format shown on the right-hand side of the figure 3. The file structure is as shown in figure 2, where the client side and the server side both generate the request and response files. Binary files are being generated as they are the format of the protocol buffer. The text files are generated to allow the user to see the content of the proto files request and responses.

II. Data serialization / de-serialization method

Data serialization, according towards data science, is the process of converting structured objects into a sequence of bytes which can be stored in a file system [2]. De-serialization is the reversed process. This is important to know understand this section. In this assignment, there are two different types of data serialization and de-serialization methods used.

The first serialization and de-serialization method used is JSON. JSON can be used for text-based serialization/de-serialization, meaning when it is (de) serialized it becomes a human-readable format from an object. To serialize a JSON object, the json.dumps() method is used. This method takes in a valid data structure, such as a dictionary. To de-serialize a byte string into a JSON object, the method json.loads() is used. This method takes in a byte string.

The second serialization and de-serialization method used is Protocol Buffer. Protocol Buffer can be used for binary serialization/de-serialization. This means it is non-human readable. Protobuf, when serialized, converts structured data into binary string. The method to serialize, in this case a proto-object, is SerializeToString() with a encoding schema of latin-1. To de-serialize a byte-string into a proto-object, the method. FromString(). This method takes in a byte type string.

The serialization methods are called before the request or before the response are being sent over the network. The de-serialization methods are called when the request or response have been received. This way, the de-serialized data received can be processed and operated upon.

III. Technical implementation of response, request and de(serialization)

Firstly, the software packages and libraries used in this assignment are numerous. The library used to deal with data serialization are based on the format of the request/response.

The serialization and de-serialization for JSON objects is done using the json library in Python. This library allows one to convert a JSON object to a byte-string and send this serialized byte-string over a TCP connection. When received, it is de-serialized and operated upon.

The serialization and de-serialization for protocol buffers is done using Google's protobuf library. This library allows one to convert a proto-object to a binary string, which is then sent to the network. The receiver will de-serialize this binary string to a proto-object once again and operate on it.

To send data from a user to a server and receive a response back, a client/sever socket is used. To do so, the socket library is used. This library allows one to create a TCP connection between a sever and a client and send data to each other.

The libraries math, numpy and pandas are used for the processing of the data on the server side. Pandas is used to read the CSV files and get the proper column and data list based on the request. The math and numpy libraries are used to do logical operation for the data analytics part of the response.

On the server side, the libraries, os and pathlib, are used for reading, writing and appending data in files.

The library protoFormat_pb2 is a protocol buffer file generated from the proto compiler. This file is generated based on the protoFormat.proto file. This is the library that is used to create proto-objects on both the server and client side.

On the client side, the library pyinputplus is used on the user inputs. This library allows one to easily do input checks without excessive coding.

The detailed implementation of the steps is as followed:

Firstly, a client, socketCreatingClient(), and server socket, serverSocketCreation(), methods are created. This is done to create a TCP connection between the client and server side to communicate the data.

Secondly, a method, clientInput(), on the client side, is implemented. It takes nine inputs and checks their validity. The inputs are then mapped to their correct values. These inputs consist of the request id, benchmark type, workload metric, batch unit, batch id, batch size, data type, data analytic and the file type (JSON or proto). This method returns all the input values in a variable. This method is called in the socketCreatingClient once the server/client connection has been established.

Now based on the file type input value, the data can take two paths. Let us explore the JSON type format first.

In the socketCreatingClient, the method makeJson() is called which takes the first eight inputs from the clientInput(). This method creates a JSON with these inputs and the serializes it using json.dumps(obj). This request is the written in a request json file on the client side, or if a request file already exists, it appends the request to the file. Finally, the method returns the serialized JSON object.

In the socketCreatingClient, the returned byte-string from makeJson() is encoded in encoding schema latin-1 and sent to the server side.

In the serverSocketCreation, the data is received and sent to the method checkDataType(), where it is decoded and parsed to see what type of data is being received.

Once the server socket knows it's a JSON binary-string, it calls parseJsonDataFromClient() which takes for input the encoded request from the client. This method decodes the request, and de-serializes it using json.loads(request). It then assigns a variable to each of the request parameters which are: request id, benchmark type, workload metric, batch unit, batch id, batch size, data type and data analytic. It then returns them.

The serverSocketCreation calls the processData() which takes as parameter all the returned variables from the parseJsonDataFromClient() and processes them. This method makes sure the request is processed properly and returns the 4 response parameters idRes, lastBatchID, data, analytics.

The serverSocketCreation method calls makeJSON() after having received the outputs from processData(). makeJSON() takes idRes, lastBatchID, data, analytics as parameters. It makes a JSON object using the parameters and then serializes the object using json.dumps(obj). It then writes the serialized byte-string to a response file, if the file path already exists, it then appends the new response in the file. Finally, this method encodes the binary-string to latin-1 and returns the encoded variable to the server socket.

The serverSocketCreation method then sends the encoded response to the client socket.

Once the response received on the client side, the socketCreatingClient method calls jsonResponse() which takes the encoded response. It decodes the response, writes it in a response file and outputs it to the client terminal. The client socket then breaks the connection with the server.

Now going back to the client inputs, the following happens when the client chooses proto as file type.

The socketCreatingClient method calls the makeProto method which takes as parameters the clientInput method's outputs. It then makes a proto object of protoFormat_pb2.requestForWorkload(). It then assigns the parameter values to each attribute in the proto object. It then writes the proto object into a text and binary file or appends it if the file already exists. This method returns a serialized proto object.

In the socketCreatingClient, the returned byte-string from makeProto() sent to the server side.

In the serverSocketCreation, the data is received and sent to the method checkDataType(), where it is decoded and parsed to see what type of data is being received.

Once the server socket knows it's a Proto binary-string, it calls parseProtoDataFromClient() which takes for input the request from the client. This method makes a proto object of protoFormat_pb2.requestForWorkload(). It then de-serializes the request using .FromString(request) method. (). It then assigns the parameter values to each attribute in the proto object, which are: request id, benchmark type, workload metric, batch unit, batch id, batch size, data type and data analytic. It then writes the proto object into a text and binary file or append it if the file already exists. It finally returns new parameters.

The serverSocketCreation calls the processData() which takes as parameter all the returned variables from the parseProtoDataFromClient() and processes them. This method makes sure the request is processed properly and returns the 4 response parameters idRes, lastBatchID, data, analytics.

The serverSocketCreation method calls makeProto() after having received the outputs from processData(). makeProto () takes idRes, lastBatchID, data, analytics as parameters. It makes a proto object of protoFormat_pb2.responseForWorkload(). It then then writes the proto object into a response text and binary files or appends it if the files already exists. Finally, this method serializes the proto object using SerializeToString() with encoding schema latin-1 returns the serialized string to the server socket.

The serverSocketCreation method then sends serialized response to the client socket.

Once the response received on the client side, the socketCreatingClient method calls protoResponse which takes the response. It makes a proto object of protoFormat_pb2.responseForWorkload(), writes the proto object into a response text and binary files or appends it if the files already exists. It also outputs the response on the client's terminal. The client socket then breaks the connection with the server.

IV. Cloud Deployment of your server code

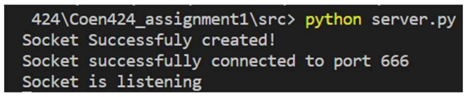
The cloud deployment of the server code is a requirement for students in teams of 2 or more. As this assignment has been completed alone, the cloud deployment part is not done. If time had allowed it, this section would have been completed but midterms prevented this from happening.

V. Instruction on how to run both the client and server applications

To use this application, one must Create 2 terminals. On both terminals, cd into the src directory Use: cd src.

Start by running the server.py first by doing: python server.py

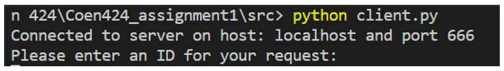
On the server terminal, you should see the successful socket creation messages as figure 4.



```
 424\Coen424_assignment1\src> python server.py
Socket Successfuly created!
Socket successfully connected to port 666
Socket is listening
```

Figure 4: Terminal output when server code has begun

Once the server is running, on the second terminal, start the client by typing: python client.py. You should see the message shown in figure 5:



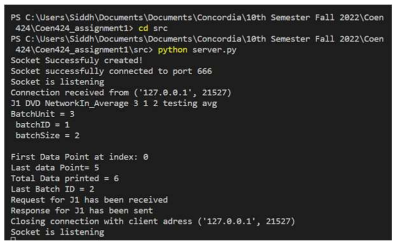
```
n 424\Coen424_assignment1\src> python client.py
Connected to server on host: localhost and port 666
Please enter an ID for your request:
```

Figure 5: Terminal output when client side has begun

On the server-side terminal, you should see the connection received message. Enter the requested inputs and a view the response. Redo the client-side steps to re-run the client code.

Check the GeneratedFiles directory to make sure all files generated are correct.

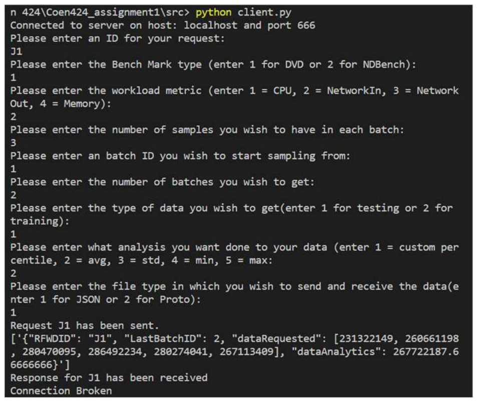
VI. Screenshots of running your application with SUCCESSFUL results.



```
PS C:\Users\Siddh\Documents\Documents\Concordia\10th Semester Fall 2022\Coen
 424\Coen424_assignment1> cd src
PS C:\Users\Siddh\Documents\Documents\Concordia\10th Semester Fall 2022\Coen
 424\Coen424_assignment1\src> python server.py
Socket Successfuly created!
Socket successfully connected to port 666
Socket is listening
Connection received from ('127.0.0.1', 21527)
J1 DVD NetworkIn_Average 3 1 2 testing avg
BatchUnit = 3
 batchID = 1
 batchSize = 2

First Data Point at index: 0
Last data Point= 5
Total Data printed = 6
Last Batch ID = 2
Request for J1 has been received
Response for J1 has been sent
Closing connection with client adress ('127.0.0.1', 21527)
Socket is listening
```

Figure 6: Successful Server-side code run



```
n 424\Coen424_assignment1\src> python client.py
Connected to server on host: localhost and port 666
Please enter an ID for your request:
J1
Please enter the Bench Mark type (enter 1 for DVD or 2 for NDBench):
1
Please enter the workload metric (enter 1 = CPU, 2 = NetworkIn, 3 = Network
Out, 4 = Memory):
2
Please enter the number of samples you wish to have in each batch:
3
Please enter an batch ID you wish to start sampling from:
1
Please enter the number of batches you wish to get:
2
Please enter the type of data you wish to get(enter 1 for testing or 2 for
training):
1
Please enter what analysis you want done to your data (enter 1 = custom per
centile, 2 = avg, 3 = std, 4 = min, 5 = max:
2
Please enter the file type in which you wish to send and receive the data(e
nter 1 for JSON or 2 for Proto):
1
Request J1 has been sent.
['{"RFWDID": "J1", "LastBatchID": 2, "dataRequested": [231322149, 260661198
, 280470095, 286492234, 280274041, 267113409], "dataAnalytics": 267722187.6
6666666}']
Response for J1 has been received
Connection Broken
```

Figure 7: Successful Client-side code run

[1] "Overview | protocol buffers | google developers," *Google*. [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview. [Accessed: 28-Oct-2022].

[2] X. Gao, "What, why and how of (de)serialization in Python," Towards Data Science, 15-Feb-2021. [Online]. Available: https://towardsdatascience.com/what-why-and-how-of-de-serialization-in-python-2d4c3b622f6b. [Accessed: 28-Oct-2022].