**CS4283/5283: Computer Networks**

Vanderbilt University, Fall 2022; Instructor: Aniruddha Gokhale

**Programming Assignment #1**

Handed out 9/8/2022; Due 9/23/2022 (11:59 pm Central in Brightspace)

# **Assignment Theme:** Custom Application Protocol with Serialization

**Overview**

The goal of this assignment is to develop client-server networking code that uses a custom Application Protocol that works on top of ZeroMQ sockets. This writeup defines the message formats and rules of engagement.

In our case, the client is an Internet of Things (IoT) artifact in the form of an edge-based intelligent/smart refrigerator. It is edge-based because the refrigerator is installed in homes/RVs, which sit at the edge of the network. It is IoT-enabled because it is assumed to include a variety of sensors and networking support.

Our smart refrigerator periodically sends two different types of messages to two different servers. The first message type that it can send out is a *Grocery Order* to a Grocery Server. The second message type that it can send is a *Health Status* to a Health Status Server. The servers in our assignment are very simple and respond with minimal information. The Grocery Server simply acknowledges with an “Order Placed” reply while the Health Status Server always responds with a “You Are Healthy” 😊 reply.

For this assignment, we will mimic one or more such smart refrigerators, one Grocery Server and one Health Status server. We will evaluate the setup locally as well as in Mininet-emulated topologies on our VM. In future assignments we will also deploy our code in Docker Swarm cluster on our VM as well as in Kubernetes cluster on Chameleon Cloud.

**Message Types and Reply Codes**

Our application will support two types of message types: ORDER and HEALTH (just like GET, PUT etc in http). In Flatbuffers, these two can be defined as Enumerated types. Reply codes consist of OK (indicating that the request is valid and that a valid response is attached to the message) or BAD\_REQUEST (indicating that the server received a message that it cannot understand or handle). Even these will be enumerated types in Flatbuffers.

The **ORDER** message has the following nested structure. **Hint:** think of this message structure as being similar to the GET message in HTTP where we had individual fields like User-Agent:, Host:, etc. In JSON, all of this will be a Python dictionary. For Flatbuffers, you will need to define an equivalent schema. See the scaffolding code from instructor’s github sample code examples.

type: ORDER

contents:

veggies:

tomato: <floating point value in pounds>

cucumber: <floating point value in pounds>

[Extend this with another 3-4 of your favorite vegetables using similar format for the value]

drinks:

cans: /\* assume pack of 6 \*/

coke: <integer quantity representing number of packs>

[Extend this with anything else like beer etc; Add another 2 entries]

bottles:

sprite: <integer representing number of bottles>

[Extend this with something more e.g. Gingerale or anything you want to add]

milk: /\* will be an array/list of 2 tuples as shown \*/

[{type: <an enumerated type>, quantity: <floating point in gallons>}+]

/\* the + above is regular expression operator meaning 1 or more \*/

/\* Create an enumeration for type, which can be 1%, 2%, fat free, whole, almond, cashew, oat \*/

bread:

[I urge you to do something similar like we do for milk with bread types, e.g. whole wheat, pumpernickel, rye, etc]

meat:

[I urge you to do something similar like we do for milk but here for meat types]

The **HEALTH** message will be structured as follows (but is simpler than our grocery order message):

type: HEALTH

contents:

dispenser: /\* decide an enumeration like OPTIMAL, PARTIAL, BLOCKAGE \*/

icemaker: <integer reflecting the percentage efficiency it is working at>

lightbulb: <GOOD or BAD (i.e., not working)>

fridge\_temp: <integer representing internal temp>

freezer\_temp: <integer representing internal temp, could be negative>

sensor\_status: /\* some enumeration like GOOD or BAD \*/

/\* use your imagination and add maybe another field or two \*/

The **response** message will be structured as follows:

code: /\* enumeration indicating OK or BAD\_REQUEST \*/

contents: /\* “Order Placed”, or “You are Healthy” or “Bad Request” \*/

**Expected Operation**

The smart refrigerator periodically (say after a random sleep interval), sends either a grocery order or a health status message. You might decide to send one grocery order message for every 5 health messages or just some random decision as to what to send. The contents of the grocery order or health status are irrelevant. In other words, even if the fridge ends up ordering 9,000 lbs of tomatoes, we do not care. What we care about is that the data is properly serialized and sent to the other side and response received from the corresponding server.

We will send such requests for multiple iterations. For each type of message sent, we will also keep track of round-trip time, i.e., time from which the message is sent to after the reply is received. We can then plot the average time and std deviation of end to end latencies per message type and when using Flatbuffers vs JSON.

**Technologies used:**

* Virtual Machine in VirtualBox (or directly on your Mac with M1 chip)
* Ubuntu Linux 20.04 LTS
* Python3
* ZeroMQ
* Mininet

**Where/How to start working on the assignment?**

First and foremost, please look at ZeroMQ tcp\_client/server examples in the github scaffolding code. Then, also look at the Serialization scaffolding code for both Flatbuffers and JSON, and how this can be combined with ZMQ send/recv calls. Finally, a skeleton code is being provided in this directory that you can use to get started and build upon.

**Milestones (and hence the deliverables and expectations):**

So that students do not start the work at the last minute, each assignment will have two milestones of which the first milestone is to ensure progress is being made while the second one is the final, graded one. The following provides a rough idea of the minimum that must be achieved by the specified milestone.

***Milestone 1:***

* Extend the message formats as required by the writeup
* Define Flatbuffer schema and generated serialization code for the schema
* JSON dictionary for the same message format
* Test one refrigerator client, one grocery server and one health status server on a single VM in three different bash shells
* Submit to Brightspace what all have you accomplished up to this point

***Milestone 2 (final):***

* Run the code for multiple Mininet topologies as specified below and for multiple clients but single instance of grocery and health server
* Collect end-to-end latency data
* Plot graphs
* Create video for the grader
* Submit the code, plots etc to Brightspace

**Mininet topologies to be used**

1. Three hosts and one switch: *sudo mn --topo=single,3 --link=tc,delay=10ms*

Will create 3 hosts: h1, h2 and h3 all connected to a single switch. Each link will have a base delay of 10 msec. Deploy your client on h1, grocery server on h2, and health server on h3

1. Three hosts and three switches: *sudo mn --topo=linear,3 --link=tc,delay=10ms*

Will create 3 hosts: h1, h2 and h3 each connected to its switch and the switches connected to each other. Each link will have a base delay of 10 msec. Deploy your client on h1, grocery server on h2, and health server on h3

1. 27 hosts and 13 switches: *sudo mn --topo=tree,depth=3,fanout=3 --link=tc,delay=10ms*

Creates a Tree of depth 3 and each intermediate node (i.e., switch) having 3 children each. Deploy grocery server on h1 and health server on h27. Write a script that then automates deployment of many clients, varying from 1 to 12 clients on hosts h10 thru h21 repeating the experiment by increasing the number of clients by one in each experiment, and collecting the data. The goal here is to see how the crowding/congestion of the network starts affecting our end-to-end latency numbers even without any errors.

**Rubrics (for grading after the final milestone)**

* Correctness (program works): 40%
* Experiment runs in all scenarios 20%
* Plots of results 20%
* README file explaining how to run the code: 10%
* Zoom-demo or self-explanatory video to grading peer: 10%