Serialization for Object Data Transportation in Cloud Computing

COEN498 PP Assignment 1 Report

Javier E. Fajardo
Department of Electrical and Computer Engineering
Concordia University
Montréal, Québec, Canada
j_fajard@encs.concordia.ca

***Abstract*—The use of serialization methods for effective data transport is an important concept in networking which remains relevant in the context of cloud computing. This document presents some of the relevant technical details in this regard through the "guessing animal game" software developed in Python. Serialization of objects was achieved using JSON and Google's Protocol Buffer, for text and binary methods respectively. The serialization mechanism was also abstracted to provide the server and client with a simple interface while allowing extensibility of objects.**

I. Running the Application

The "Guessing Animal" game was implemented using Python 3.5 and protocol-buffer version 3.2, therefore, running it from source require Python 3 and the python protobuf bindings to be present. To run the game server from source, a user can navigate through command line to the source directory and type "python server.py". Running the server may require administrator privileges to bind to the known network interfaces. The game client can be executed in a similar fashion, but will require two command line arguments to be provided: the hostname and the serialization method (e.g. "python client.py fajardo.io protobuf").

For the convenience of users that might not have a python installation, precompiled binaries have been provided for Windows and Ubuntu. These compiled binaries must also be run from command line. The compiled client requires the same arguments to be provided.

Starting on Wednesday, March 1st, there will be two cloud computing instances running the application. One instance will be located at "fajardo.io", which is hosted in New York, US by Digital Ocean. Another instance will be located at "timmy.noip.me", a self-hosted Linux machine located in Montréal, CA. Both instances will be running the game server until Friday, March 31st, 2017.

II. Application Data Model

The data model of the application is simple and functional. Three main classes which require serialization were implemented: Animal, Question and Answer. The overall implementation of these classes were made to resemble key-value stores as much as possible to simplify both serialization and programmer use. The primary exchange of data is done using the Question and Answer class in order for the client to guess the correct Animal. The Question class contains and "inquiry" string, which is the name of a member variable in the Animal class, and a "guess" of its value. The Answer class contains the original question being answered along with two Booleans: "response" to indicate if the question is true or false, and "game_over" to indicate that was the last question/answer cycle in the game. The Animal class contains a name and a list of qualities, features, abilities and colors that describe a given instance. Therefore, acceptable values of an "inquiry" in a question can be "name", "qualities", "features", "abilities" and "colors"

The client maintains a QuestionTree, used to determine the animal that must be guessed. The QuestionTree is composed of a simple node data structure, whose value is a Question that must be asked and which has a left and right child which should also be nodes. The client will traverse the tree from the root to one of the leaves by asking the question in each value. If a guess is correct, the client moves to the right child. Conversely, it will move to a left child. The tree was crafted such that:

a. The tree can be traversed easily from an Answer object
b. Most questions inquire about the qualities of an animal

c. All leaf nodes attempt to guess the name of the animal

The server and client have a common list of animals that are known to both. The server picks an Animal object at random from said list and then responds to Questions based of the member variables in the selected Animal. The server will send the full Animal object to the client once a correct, final guess is made.

III. Serialization Methods

The serialization mechanism in the application is very straightforward: classes that can be serialized correctly inherit from the Serializable class and must implement the associated "serializeJSON", "deserializeJSON", "serializeProtobuf", "serializeProtobuf" methods. The class name is inspired by an analogous class found in Java for the same purpose. However, the implemented class allows for greater programmer flexibility in use by also exposing a single "serialize" and "deserialize" method that can be used to specify between JSON and protobuf serial data. This enables some abstraction in the logic of the server and client code.

The underlying JSON and protobuf objects supporting serialization were made to take advantage of the Python's built in dictionary types that is associated to its object-orientation model. They are described in more detail ahead.

A. JSON Serialization Format

In Python, JSON serialization is readily possible in a manner similar to the format's original language, Javascript. Essentially, Python enables object-orientation through its dictionary constructs. Provided that the user-defined class consists of built-in Python types, the "__dict__" property of a class instance provides the information to create a correct JSON string. A less straightforward serialization case occurred with the Answer class of this application, which required the contained Question class instance to be explicitly transformed into or recovered from a dictionary before the Answer class instance could undergo serialization/deserialization.

B. Protobuf Serialization Format

Google's Protocol Buffer (protobuf) serialization required drafting a specific, language agnostic interface of the classes as "Messages" with typed fields. Compared to JSON, the message layout is clearly more succinct and additionally having to specify types allows protobuf to enforce for these in any language.

The protobuf message interface was written for the classes in the application and then compiled into a python usable format. The process of integrating the compiled files was straightforward and without issue.

IV. Application use of serial communication

The design of the data model and serialization methods in the application allows for high level use, since the serialization implementation is completely abstracted from the logic of the client and the server, with only the mode of serialization (JSON or protobuf) being shared between the two before playing the game.

A running game server instance listens for a compatible client on TCP port 4981. When the server receives a connection, a handshake between the server and client takes place to ensure compatibility and the last message of the handshake is the serialization mode sent by the client. Following the handshake, the client sends a Question in the serialization mode previously specified. The server deserializes the data to recover the question and crafts an appropriate Answer that the client must deserialize and interpret through its QuestionTree class instance. The serialization mode is maintained throughout the game and the server will close the connection once the client correctly guesses the secret animal. Therefore, the logic of both the server and the client is centered around completing the game, rather than the implementation of serialization or deserialization. The particular implementation of the serialization mode is encapsulated in the Serializable class. This avoids confusing conditional logic in both the server and the client and provides ease of extensibility.

V. Software Libaries

Python is very complete in terms of its built-in libraries. The language features an internally maintained and efficient JSON parser library along with an impressive collection of networking, operating system and time tools. The only required external library for this application was "protobuf". However, this library was readily obtained using Python's built-in package manage: PIP.

The application was developed in a relatively short time period. For the server, Python's "socketserver" library provided a readily available solution for running a robust TCP server in sixty lines of code. The client is, overall, a lengthier file since is uses the low-level "socket" library, which wraps the BSD socket interface. Nevertheless, Python's use of dictionary types for

enabling its object orientation is perhaps what most influenced the speed of development. For most classes that derive from Serializable, JSON serialization is enabled by default in the "Serializable.serializeJSON" method. Only the Answer class required overriding of said method. Similarly, use of protobuf was greatly enhanced by Python's straightforward syntax.

One issue that did occur was an unintended consequence in the use of Python's ".strip()" method with raw socket data. Because the ".strip()" is intended to remove unwanted newline characters from a stream, when operating on raw protobuf data, it would tend to remove important bytes data. However, this issue was quickly detected and corrected.

VI. Application Images

The following section presents some images of the running application.

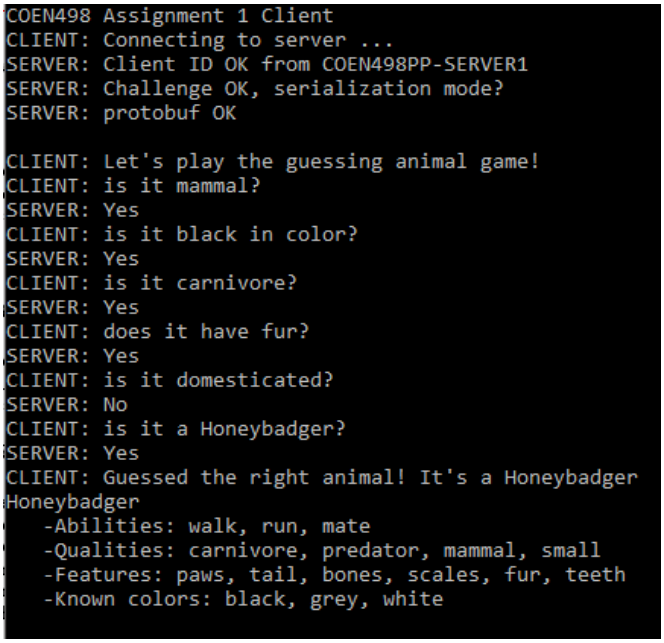

Figure 1: Game Client (Windows) guesses Honeybadger

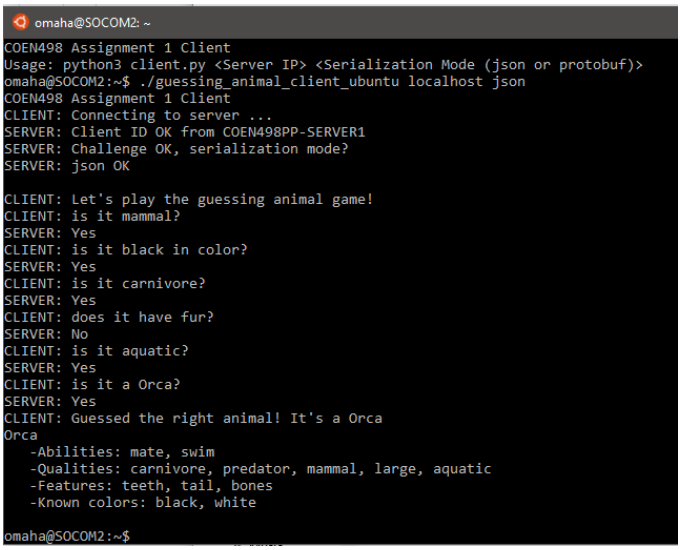

Figure 2: Native Game Client (Ubuntu) guesses Orca

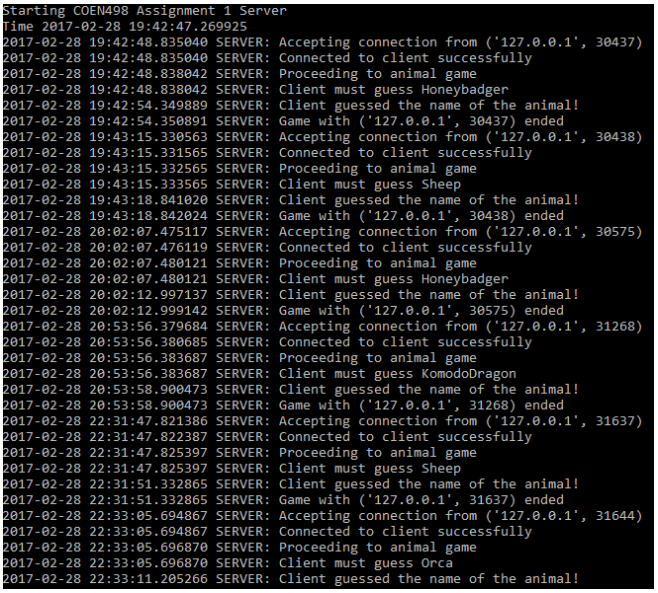

Figure 3: Game Server showing logs