The Bifrost

Release 0.1

Andrew Garcia, Ph.D.

May 16, 2023

CONTENTS

CHAPTER

ONE

A BRIDGE FOR YOUR DATA STRUCTURES

The Bifrost is currently a platform suite of scripts libraries based on Google Protocol Buffers (protobufs) designed to connect data structures across different programming languages.

Building a system using protobufs can be a game-changer for developers and organizations. It makes building distributed systems much simpler because it provides a common data interchange format that can be used across different parts of the system. This means you won't have to write a lot of code to handle data conversion and serialization/deserialization.

In addition, using more mature versions of our current suite will improve application performance and efficiency. This is because our use of protobufs reduce the size of data transferred over the network, which can significantly improve data transfer speeds. The fact that protobufs have standardized data formats also improve the reliability of data transfers between different systems. It makes it easier to understand and validate data, which can help prevent errors and improve overall system stability.

Building this system, based on Google Protocol Buffers, can be an incredibly useful tool for modern software developers. It has the potential to simplify building distributed systems, improve application performance and efficiency, and provide a reliable and standardized data interchange format that can be used across different systems.

Check out the *Usage* section for further information, including how to *Installation* the project.

Note: This project is under development and currently on its nascent stages. Open-source contributors are more than welcome for collaborations.

CHAPTER

TWO

CONTENTS

2.1 Usage

2.1.1 Installation

Due to the early stage of development of the project, we recommend that you clone the repository [using SSH] and manually review the scripts using a text editor. This will allow you to have greater control over the code and ensure that you understand any changes or modifications that are made to the project.

```
$ # Clone the repository and change dir (cd) to cloned repo
$ git clone git@github.com:username/bifrost.git
$ cd bifrost
$ # Open the repository in your preferred text editor
$ code .   #if using Visual Studio Code
```

If you're unfamiliar with terminal commands, here are more graphical-friendly instructions.

To clone the Bifrost repository, first navigate to the GitHub repository website using your web browser. Once you're in the repository, look for a green button labeled "Clone" on the right-hand side of the page. Click on this button to reveal a dropdown menu.

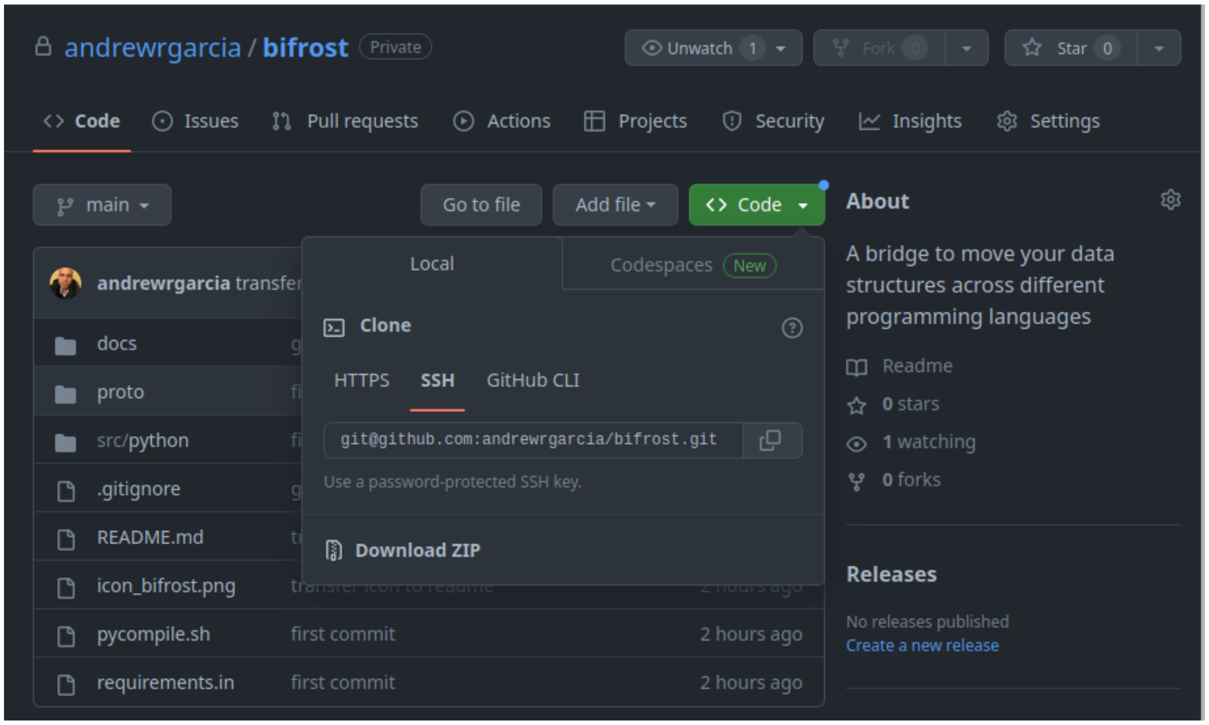
In the dropdown menu, you'll see a few options for cloning the repository. Select the one that says "HTTPS" if you're unfamiliar with the other options. Then, copy the provided code that appears in the box next to the "HTTPS" option.

Next, open up your terminal or Git shell on your computer. In the terminal, navigate to the directory where you'd like to clone the repository (for example, `~/Documents/`).

Once you're in the desired directory, paste the code you copied from the GitHub website into your terminal and press enter. This will initiate the cloning process, and the Bifrost repository will be copied to your local machine for you to access and work with.

After this, proceed by installing libprotobuf-dev, which is a package in Ubuntu that provides the development files for the Google Protocol Buffers library. The package includes header files, static libraries, and other resources necessary for developing software that uses Google Protocol Buffers:

```
sudo apt install libprotobuf-dev
```



```
git@github.com:andrewrgarcia/bifrost.git
```

Use a password-protected SSH key.

A bridge to move your data structures across different programming languages

Python

```
pip install -r requirements.in
```

2.1.2 Functional Organization Structure

The Bifrost project has a well-designed project structure that is both lean and functional. At the highest level, we have high-level scripts such as compile.sh to manage the project. The `proto/` directory contains all the Protobuf schemas used to define data structures, while the `src/` directory houses the source code for running these data structures.

To keep things organized, the `src/` directory is divided into subdirectories for different programming languages. Additionally, the `tests/` directory has the same organizational structure as src, but is dedicated to test scripts for ensuring the system operates as expected. This structure enables us to navigate and organize the project with ease, which is crucial for software development.

```
bifrost/
├── compile.sh
├── proto/
│   ├── schema1.proto
│   ├── schema2.proto
│   ├── ...
│   └── schemaN.proto
├── src/
│   ├── python/
│   │   ├── module1.py
│   │   ├── module2.py
│   │   ├── ...
│   │   └── moduleN.py
│   │
```

(continues on next page)

(continued from previous page)

```
│   ├── javascript/
│   ├── cpp/
│   ├── java/
│   ├── go/
│   └── ...
├── tests/
│   ├── python/
│   ├── javascript/
│   ├── cpp/
│   ├── java/
│   ├── go/
│   └── ...
└── ...
```

2.1.3 Easy Compiling

We have made a simple shell script based on the protobuf tutorials to compile our protobuf files in the `bifrost/proto` directory. To compile the `.proto` files to any language, simply execute the `compile.sh` script followed by the name of the programming language you wish to compile the protobuf file to.

For example, if you would like to compile all these files to cpp, type:

```
>>> bash compile.sh cpp
```

To do so in Python, type:

```
>>> bash compile.sh python
```

After running these 2 commands, cpp- and python-compatible compiled data structure files will be generated in new `proto/` directories to those languages housed within the `src/` folder, as below:

```
bifrost/
│   ...
├── src/
│   ├── python/
│   │   ├── proto/
│   │   ├── ...
│   ├── cpp/
│   │   ├── proto/
│   │   ├── ...
│   └── ...
└── ...
```

2.1.4 The Proto Files

So far, we have created two Protobuf schemas, which are designed to facilitate the transfer of data structures between different programming languages. These schemas specify the structure and data type of the information being transferred, enabling it to be defined in any language that supports Protobuf.

The first schema we designed handles a container called **Object** which consists of a string field and a one-dimensional array. It's worth noting that the different fields in this container must be indexed for easy access and manipulation.

```
syntax = "proto3";

message Object {
string string_field = 1;
repeated int32 array_field = 2;
}
```

The second schema we developed was created to handle a third-order numerical tensor called **NumTensor**. This schema was developed to support the author's interest in representing three-dimensional objects mathematically. The **NumTensor** schema allows for the efficient storage and manipulation of numerical data in a three-dimensional format, making it useful for a wide range of applications.

```
syntax = "proto3";

message NumTensor {
repeated NumMatrix my_arrays = 1;
}

message NumMatrix {
repeated NumRow my_sub_arrays = 1;
}

message NumRow {
repeated int32 my_array = 1 [packed=true];
}
```

By using a programming language that supports Protobuf, we can easily create data structures based on the schemas we have defined. To create an **Object** container and a **NumTensor** tensor with Python, we may call the relevant functions in the `src` directory housing the `Python` scripts, which pass the necessary parameters as specified by our Protobuf schema.

The first function call is to a script which takes the **Object** object from `proto/object.proto`, defining the string field as *hello world* and the array field as a list from 0 to 9:

```
>>> python src/python/message.py
string_field: "hello world"
array_field: 0
array_field: 1
array_field: 2
array_field: 3
array_field: 4
array_field: 5
array_field: 6
array_field: 7
array_field: 8
array_field: 9
```

The second function call applies the **NumTensor** object from the `proto/tensors.proto` file to make a 3x3x3 tensor with random numbers from 0 to 99:

```
>>> python src/python/num_tensors.py
my_arrays {
  my_sub_arrays {
    my_array: 66
    my_array: 43
    my_array: 51
  }
  my_sub_arrays {
    my_array: 1
    my_array: 30
    my_array: 72
  }
  my_sub_arrays {
    my_array: 50
    my_array: 41
    my_array: 20
  }
}
my_arrays {
  my_sub_arrays {
    my_array: 91
    my_array: 88
    my_array: 69
  }
  my_sub_arrays {
    my_array: 6
    my_array: 43
    my_array: 23
  }
  my_sub_arrays {
    my_array: 38
    my_array: 39
    my_array: 85
  }
}
my_arrays {
  my_sub_arrays {
    my_array: 20
    my_array: 69
    my_array: 94
  }
  my_sub_arrays {
    my_array: 35
    my_array: 91
    my_array: 5
  }
  my_sub_arrays {
    my_array: 34
    my_array: 81
    my_array: 15
```

(continues on next page)

(continued from previous page)

```
  }
}
```

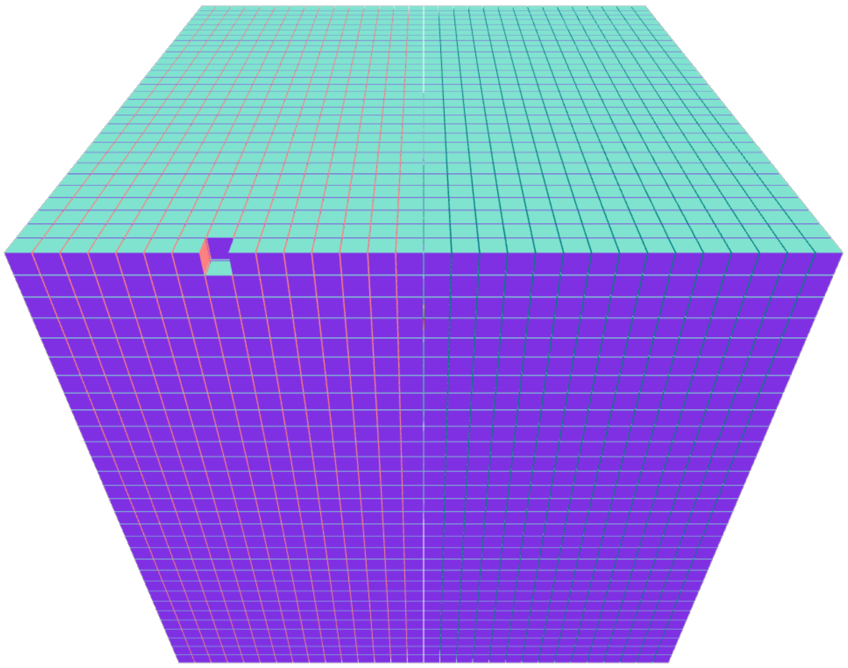
2.2 API Reference

2.3 Python

2.3.1 Writing and Reading 3-D Arrays using Protocol Buffers in Python

Protocol Buffers (protobuf) is a language- and platform-neutral data serialization format developed by Google. In this tutorial, we will show you how to use protobuf to write and read 3-D arrays in Python. We will also demonstrate how to convert the resulting protobuf objects into 3-D numpy arrays for further processing.

Memory Savings



A 30x30x30 dense array stored as a protobuf object occupies only **31 kB** of memory. In contrast, the same array saved as a coordinate matrix in "zyxa" format consumes approximately **320 kB** of memory, which is over 10 times more than the protobuf object. This demonstrates the efficiency of protocol buffers in terms of memory usage.

*z,y,x for coordinates, a for entry value

Requirements

Before we begin, make sure that you have protobuf installed in your Python environment. You can install it via pip by running:

```
pip install protobuf
```

Writing a 3-D Array to a protobuf File

First, we need to define a protobuf message that represents a 3-D numeric tensor. We have drafted a file to do this and named it tensors.proto:

```
syntax = "proto3";

message NumTensor {
  repeated NumMatrix my_arrays = 1;
}

message NumMatrix {
  repeated NumRow my_sub_arrays = 1;
}

message NumRow {
  repeated int32 my_array = 1 [packed=true];
}
```

This definition specifies a NumTensor message that contains a repeated field of NumMatrix messages, each of which contains a repeated field of NumRow messages. The NumRow message simply holds a 1-D array of int32 values.

After this we must compile our *.proto* file to python format. This is done by running the compile.sh file in the main github directory of this platform:

```
bash compile.sh python
```

Next, let's write a Python script that creates a 3-D numpy array and populates the corresponding protobuf message fields. The script then writes the serialized message to a file in binary format. Here's an example implementation taken from our tensors_port.py code:

```
import proto.tensors_pb2 as protoTensors
import numpy as np


'declare a numeric tensor from `NumTensor` protobuf'
tensor_3d = protoTensors.NumTensor()

#This is a 3-D tensor with random numbers between 0 - 99
tensor = np.random.randint(0,100,(3,3,3))
Z,Y,X = tensor.shape

'populate fields of protobuf object'
for i in range(Z):
    my_array = tensor_3d.my_arrays.add()
    for j in range(Y):
```

(continues on next page)

(continued from previous page)

```
        my_sub_array = my_array.my_sub_arrays.add()
        for k in range(X):
            my_sub_array.my_array.append(tensor[i,j,k])

print(tensor_3d)

with open("tensor.bin", "wb") as f:
    f.write(tensor_3d.SerializeToString())
```

In this example, we first create a 3-D numpy array tensor with random integer values between 0 and 99. We then create a NumTensor protobuf message tensor_3d and populate its fields by iterating over the tensor array and appending its values to the corresponding NumRow fields.

Finally, we serialize the tensor_3d message into a binary string using the SerializeToString() method and write it to a file named "tensor.bin" using the open() and write() functions.

Reading a 3-D Array from a protobuf File

Now that we have written a 3-D numpy array to a protobuf file, let's demonstrate how to read it back into a numpy array. We first need to parse the binary string in the file into a NumTensor protobuf message, and then we can extract the values and create a 3-D numpy array.

Here's an example implementation that can be found in our tensors_load.py code:

```
import proto.tensors_pb2 as protoTensors
import numpy as np

# Load the serialized tensor from file
with open("tensor.bin", "rb") as f:
    serialized_tensor = f.read()

# Parse the serialized tensor into a NumTensor protobuf object
tensor_3d = protoTensors.NumTensor()
tensor_3d.ParseFromString(serialized_tensor)

# Unpack the NumTensor protobuf object into a 3-D numpy array
my_array = tensor_3d.my_arrays[0]  # get first my_array
Z, Y, X = len(tensor_3d.my_arrays), len(my_array.my_sub_arrays), len(my_array.my_sub_
↪arrays[0].my_array)
tensor = np.zeros((Z, Y, X), dtype=np.int32)

for i in range(Z):
    my_array = tensor_3d.my_arrays[i]
    for j in range(Y):
        my_sub_array = my_array.my_sub_arrays[j]
        for k in range(X):
            tensor[i,j,k] = my_sub_array.my_array[k]

print(tensor)
```

This approach allows for efficient serialization and deserialization of large multidimensional arrays. Additionally, because protobuf messages are platform-independent, the same file can be read by any other programming language that supports protobuf. In particular, the above code can be easily adapted to work with other numpy arrays of any dimensions.