Bryn Mathias

37 Followers About Follow

Kafka and Google Protobuf a match made in Python

Bryn Mathias Jul 30, 2018 · 7 min read

A little bit of preface; Over the last few years I've been working in early stage startups, trying as always to do things quickly, correctly and pragmatically. Be this collecting billing information from smart-ish meters from 1000's of homes, reconciling this data, issuing bills and taking payments from cash, card and direct debit or building systems for collecting real time health data from medical trial participants and provide a platform for automatic actions based on these data.

buffers.

All the code in this post is released under an MIT license (Feel free to do what ever you want with it, just don't blame me if it breaks) and can be found at:

Moving away from the monolith

The first part of the majority of start up work is moving from an initial hacky system that one person has built, knows like the back of their hand and can develop really quickly is making it into something more than one person can work on. This does all sorts of wonderful things like improving productivity, making the code testable, letting people go on holiday and all of that stuff.

To this end, docker and a message passing system making coding like playing with Lego, don't like the size or colour of this brick… swap it out seamlessly with this better brick. But there's enough raving about containers and micro services out there that I don't need to do it now!

The key component of breaking down this monolith is inter-service communication, for us a rest based RPC system in something like cherry py, was easy to use, but didn't cut it in terms of scalibility or failure recovery, transactional RPC calls have their place and I'd recommend Google's GRPC framework for this for many reasons. I'll write a post later on joining GRPC with kafka on both the front and back end of your micro service architecture, but for the moment I'll focus on where we started: kafka.

post which I found a while ago on reddit's r/python:
http://blog.adnansiddiqi.me/getting-started-with-apache-kafka-in-python

We started by breaking out a few parts of our core functionality into containers and passing data between them using json blobs the pitfalls of this were:

1. Meet your (very) dyslexic author: Spelling field names wrong is bad enough when you're writing a receiving service, worse when you're writing a sending service. It's really nice to have named fields which your IDE can help you with, rather than spelling a field name differently each time you try to use it.
2. Changing field names because a team member decides name a is more descriptive than name b, or prefers a different style of capitalisation.
3. Adding / removing fields due to requirement change
4. Documenting the JSON field names and the type structure.
5. …. there are more I'm sure but I can't think what they were.

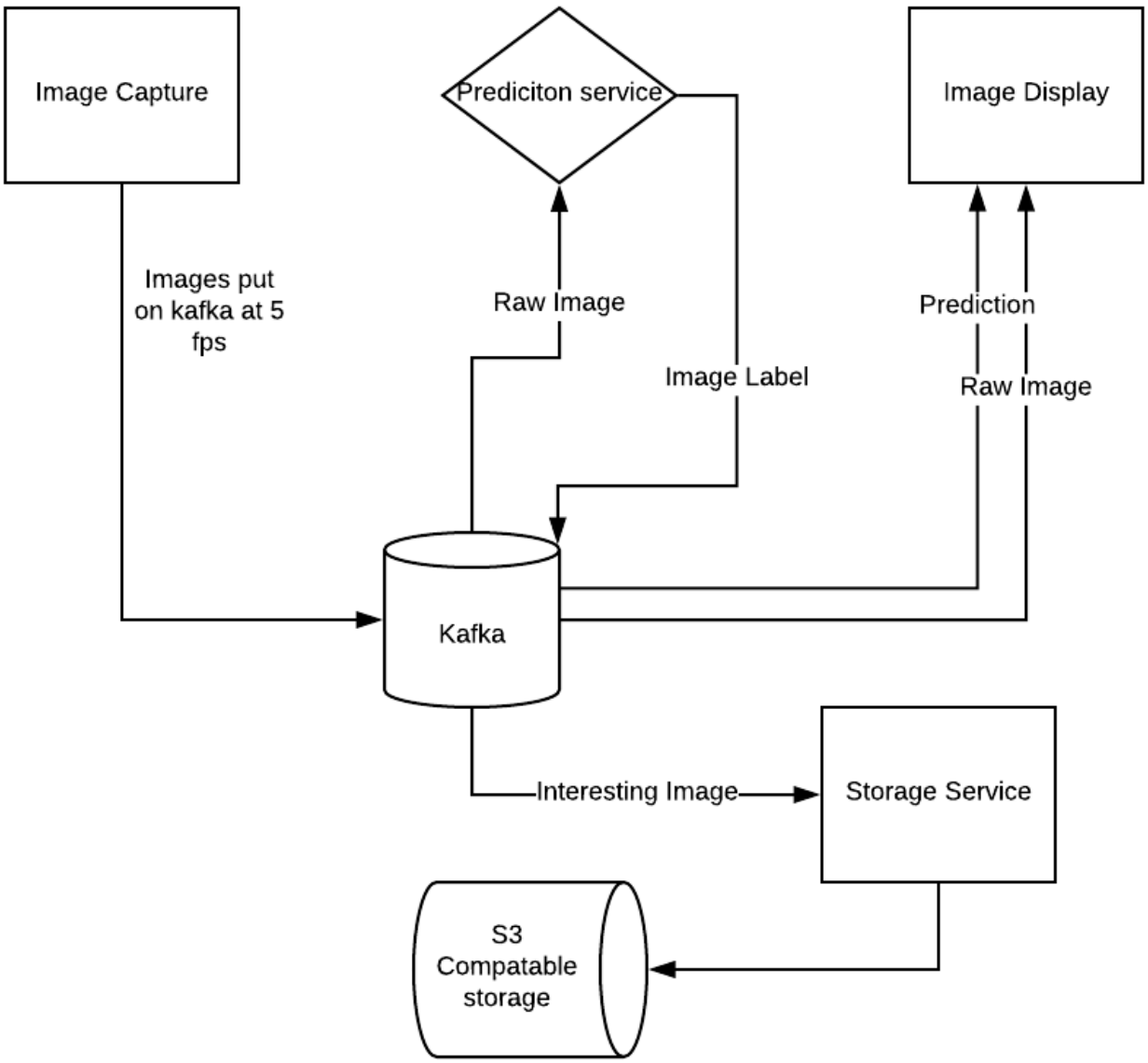
The possible solutions to this were:

1. JSON schema: In theory this works but in practice forcing the schema checking is a pain and it isn't enforceable in the code.
2. Apache Avro: Tried, nested structures are horrible.
3. Google proto buffers (PB): We found lots of benifits; cross language compatibility, easy to write and understand, use with GRPC framework etc etc.

Example System

All the code below is drawn from an example system which I've been writing to go along with my CV.

It's a very basic system involving image capture, image labelling using a pre-trained neural network in Keras, logging using the ELK stack and storage of frames related



Simple Systems Arch for Image capture, prediction, display and storage system

For this example we want to send an image captured from our webcam with pygame to the display service and the prediction service. In JSON this looks like:

```
im = {'image_data': image.tostring(...), 'height': 480, 'width':640,
'channels': 'RGB', 'frame_no': frame_n, 'meta_data': {...} }
```

The PB definition of this looks like:

```
1  syntax = 'proto3';
2
3  package protobuf_data;
```

```
6
7   message Image {
8       bytes image_data = 1;
9       int32 height = 2;
10      int32 width = 3;
11      int64 frame = 4;
12      protobuf.common.messageData _message_data = 5;
13
14  }
```

ImageData.proto hosted with ❤️ by GitHub view raw

Where the `_message_data` field is our meta data from above. I'll go into more depth on this sub message in a later post, but for now we'll just include it, if you want to see the contents check the git repo mentioned at the start of the post.

Here we can see that we have defined names and types, we can import sub message types. Protobufs can be far more complicated, with lots of message definitions per `.proto` file but for now we'll keep it simple.

Playing with the Image object in python gives us something like below

```
puddlejumper ~ » python3
Python 3.6.5 (default, Apr  1 2018, 05:46:30)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> from protobuf.data.ImageData_pb2 import Image
>>> my_image = Image()
>>> print(my_image)

>>> my_image.height=480
>>> my_image.width=640
>>> my_image.frame = 10
>>> my_image.image_data = b"test image data bytes"
>>> print(my_image)
image_data: "test image data bytes"
height: 480
width: 640
frame: 10

>>> my_other_image = Image(image_data=b"second test image data",
width=480, height=480, frame=11)
>>> print(my_other_image)
image_data: "second test image data"
height: 480
```

Here we can see we can set the members either via their member names, or in the constructor. Note any empty fields are not printed.

Sending protos over the wire with Kafka

Now on to the cool bit, we want to put these messages on to the kafka broker for any services want to consume them. We'll use the kafka-python library since it's well maintained. For a more indepth introduction to kafka see the excellent post at:

Getting started with Apache Kafka in Python

In this post, I am going to discuss Apache Kafka and how Python programmers can use it for building distributed...

blog.adnansiddiqi.me

For our purposes we need to wrap the producer and consumer classes, adding some functionality for automatically adding meta data, forcing a channel to PB mapping, compressing / decompressing the messages and converting the PBs to a format that's kafka friendly again for for the code see the repo, but the work that was required is explained below.

1. Message Handler, used by the producer; this adds meta data to the PB object and has functions for converting the object to a string and compressing this string using `zlib`
2. Message Loader, used by the consumer to load the message from kafka based on the channel the consumer is subscribed to, updating any required meta data
3. Channel to Object map, this has to be hand written. In a larger team / solid production env you might want to replace this with a redis look up so you can dynamically add channels to your system, but in this case we'd rather go for having to check by hand before deployment.

Wrapping the producer and consumer classes gives us the following:

1. Auto loading of the correct PB type

receive the messages in round robin mode so you can scale, rather than having each instance receive all the messages, kafka does this automatically based on the group name of the consumers subscribed to a topic.

4. Logging, meaning we can turn off the kafka-python loggers and only see the information we need.

Examples of sending and receiving messages are shown below (also in the examples folder which contains a docker-compose file for bringing up kafka)

To send the messages we create a producer, talking to the kafka broker on `localhost:9092` on channel `"VideoStream"` we then need an instance of the Image PB to send, with some example data. To show updating the message on the fly we increment the frame number member for each message we send.

```
#!/usr/bin/env python3
from MessageCorps.broker import kafka_producer
from protobuf.data.ImageData_pb2 import Image
from time import sleep
import logging

logging.basicConfig(level=logging.DEBUG)

def main():

    producer = kafka_producer(bootstrap_servers=['localhost:9092'], topic="VideoStream"

    test_image = Image(
        image_data = b"this is just a test string",
        height = 10,
        width = 10
    )
    for frame in range(100):
        test_image.frame = frame
        producer.send(test_image)
        sleep(1)




if __name__ == "__main__":
```

Consuming the messages is just as simple, we create a consumer in much the same way as the producer and iterate though the messages, if the are no pending messages the consumer waits for new ones, as expected.

```
#!/usr/bin/env python3

from MessageCorps.broker import kafka_consumer
import logging

logging.basicConfig(level=logging.DEBUG)


def main():
    consumer = kafka_consumer(bootstrap_servers="localhost:9092", topic="VideoStream")

    for message in consumer.consume():
        print(message)


if __name__ == '__main__':
    main()
```

Example_consumer.py hosted with ❤️ by GitHub view raw

A quick note on thread safety, consumers are thread safe and producers are not.

Summary

To sum up, we've taken a quick look at PBs and why they're useful when combined with kafka. We've wrapped Kafka to make it easy to use with PBs and added some meta data for tracing calls though the system (more in depth view on this coming up).

The bigger picture of the Image collection, display, prediction and storage application will be the subject of later posts on systems design as well as extending the MessageCorps library with GRPC support.

For now though feel free to use / make suggestions on the MessageCorps lib and critique of the post is also welcome.