# Building a real time NYC subway tracker with Apache Kafka

## A practical use case of Apache Kafka

Lei He Follow

Jul 21, 2018 · 4 min read

Data come in different shapes and forms. In the real world, a lot of data are generated in streams. Streaming data is a continuous flow of events

generated by various sources. With the ubiquitousness of mobile devices, social media, IoT and big data platforms, more and more data sources offer streaming data for analytics consumption. Real time analytics of streaming data differs from batch data in that analytics are updated based on a few new data events, instead of the entire dataset.

Some examples of streaming data are

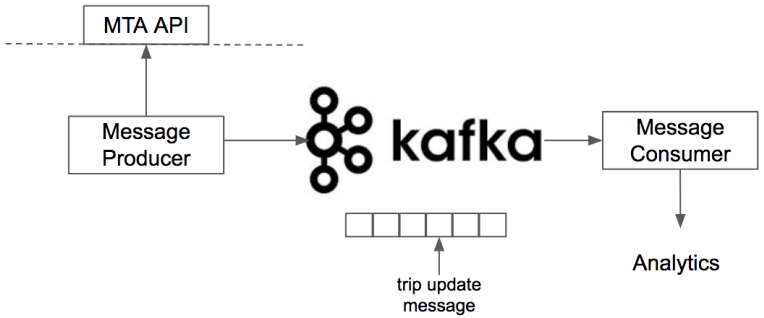
1. Facebook/Instagram/Twitter posts
2. Sensor data sent from airplane engines
3. Time series data from financial stocks trading on exchanges

In recent years, Apache Kafka has become the technology of choice when it comes to working with streaming data. It offers persistent storage that guarantees message ordering, and a pub-sub semantics that are easy to reason with. It can handle terabyte throughput easily and is a popular choice for any company that works with massive amount of event-driven data.

MTA, the agency operating NYC subways, publishes real time subway trip updates via a set of RESTful APIs that are available to the public.

In this post we will use Apache Kafka to build a real time NYC subway tracker that shows you when the next train will arrive in the station. Here at Cloudbox Labs, building cloud native applications is our business. In that spirit, we will host our Kafka cluster on Amazon Web Services (AWS) and our application on EC2.

The overall architecture of our system looks like this




In order to interact with Kafka pub-sub model, we will write a message producer that generates message streams and publish them onto Kafka. MTA's data stream refreshes every 30 seconds so our producer will query the API for the latest subway trip updates and publish these updates to a Kafka topic. On the other end we will spin up a message consumer that listens on a particular topic and process the update messages as they come through. The consumer keeps an in-memory data structure that stores timing of the next arrival train for each station by subway line.

For demo purpose our Kafka cluster setup is fairly simple. We set up a two node Kafka cluster so we can take advantage of its distributed and replicated storage capability. We created one topic that will store our message queue. With that we are ready to interact with the Kafka cluster.

MTA's trip update API uses protobuf in gtfs-realtime feed specification so the data has a standard format. In order to parse the response from the API we need to first auto-generate the python protobuf classes based on gtfs-realtime.proto

```
python -m grpc_tools.protoc -Iprotos/ --python_out=. --
grpc_python_out=. protos/gtfs-realtime.proto
```

Using the Confluent Kafka python client, writing Kafka producer and consumer are fairly easy. Here is the code snippet of our trip update producer

```
import time

import requests
from google.protobuf.json_format import MessageToJson
from confluent_kafka import Producer

import gtfs_realtime_pb2


class MTARealTime(object):

    def __init__(self):
        with open('.mta_api_key', 'r') as key_in:
            self.api_key = key_in.read().strip()

        self.mta_api_url =
'http://datamine.mta.info/mta_esi.php?key=
{}&feed_id=1'.format(
```

```
            self.api_key)
        self.kafka_topic = 'test'
        self.kafka_producer = Producer({'bootstrap.servers':
'localhost:9092'})

    def produce_trip_updates(self):
        feed = gtfs_realtime_pb2.FeedMessage()
        response = requests.get(self.mta_api_url)
        feed.ParseFromString(response.content)

        for entity in feed.entity:
            if entity.HasField('trip_update'):
                update_json =
MessageToJson(entity.trip_update)
                self.kafka_producer.produce(
                    self.kafka_topic,
update_json.encode('utf-8'))

        self.kafka_producer.flush()

    def run(self):
        while True:
            self.produce_trip_updates()
            time.sleep(30)
```

And here is our trip update consumer

```
import csv
from collections import defaultdict
import json

import arrow
from confluent_kafka import Consumer, KafkaError


class MTATrainTracker(object):
```

```
    def __init__(self):
        self.kafka_consumer = Consumer({
            'bootstrap.servers': 'localhost:9092',
            'group.id': 'test_consumer_group',
            'default.topic.config': {
                'auto.offset.reset': 'smallest'
            }
        })
        self.kafka_topic = 'test'

        # subway line number -> (stop_id, direction) -> next
arrival time
        self.arrival_times = defaultdict(lambda:
defaultdict(lambda: -1))

        self.stations = {}
        with open('static/mta_stations.csv') as csvf:
            reader = csv.DictReader(csvf)
            for row in reader:
                self.stations[row['GTFS Stop ID']] =
row['Stop Name']

    def process_message(self, message):
        trip_update = json.loads(message)

        trip_header = trip_update.get('trip')
        if not trip_header:
            return

        route_id = trip_header['routeId']
        stop_time_updates =
trip_update.get('stopTimeUpdate')
        if not stop_time_updates:
            return

        for update in stop_time_updates:
            if 'arrival' not in update or 'stopId' not in
update:
                continue

            stop_id, direction = update['stopId'][0:3],
update['stopId'][3:]
```

```
            new_arrival_ts = int(update['arrival']['time'])

            next_arrival_ts = self.arrival_times[route_id]
[(stop_id, direction)]
            now = arrow.now(tz='US/Eastern')

            if new_arrival_ts >= now.timestamp and \
                    (next_arrival_ts == -1 or new_arrival_ts
< next_arrival_ts):
                self.arrival_times[route_id][(stop_id,
direction)] = new_arrival_ts

                # convert time delta to minutes
                time_delta = arrow.get(new_arrival_ts) - now
                minutes = divmod(divmod(time_delta.seconds,
3600)[1], 60)[0]
                print('Next {} bound {} train will arrive at
station {} in {} minutes'.format(
                    direction, route_id,
self.stations[stop_id], minutes))

    def run(self):
        self.kafka_consumer.subscribe([self.kafka_topic])

        while True:
            msg = self.kafka_consumer.poll(1.0)

            if msg is None or not msg.value():
                continue
            if msg.error() and msg.error().code() !=
KafkaError._PARTITION_EOF:
                raise ValueError('Kafka consumer exception:
{}'.format(msg.error()))

            msg = msg.value()
            self.process_message(msg.decode('utf-8'))
```

Both the Kafka cluster and the application components run on separate EC2 on AWS. Once we spin up all the pieces, we can see that the

consumer is listing the next train arrival time for each station

```
Next S bound 1 train will arrive at station Rector St in 0
minutes

Next S bound 1 train will arrive at station South Ferry in 2
minutes

Next S bound 1 train will arrive at station Cortlandt St in
0 minutes

Next N bound 6 train will arrive at station Spring St in 4
minutes

Next N bound 6 train will arrive at station Bleecker St in 5
minutes

Next N bound 6 train will arrive at station Astor Pl in 6
minutes
```

Streaming data has major applications in modern data analytics. At Cloudbox Labs, we love building stream processing platforms so our clients can analyze data faster and at scale.

. . .

As always you can find the full code discussed in this post on Cloudbox Labs github.

Tags: Amazon Web Services, Apache Kafka, gtfs-realtime, NYC subway tracker, protocol buffer, rest API, streaming data