Architecting a distributed crowdsourcing infrastructure based on Mqtt by using Apache Kafka

Santosh Kumar Ghosh *SBUID: 109770622*

Motivation:

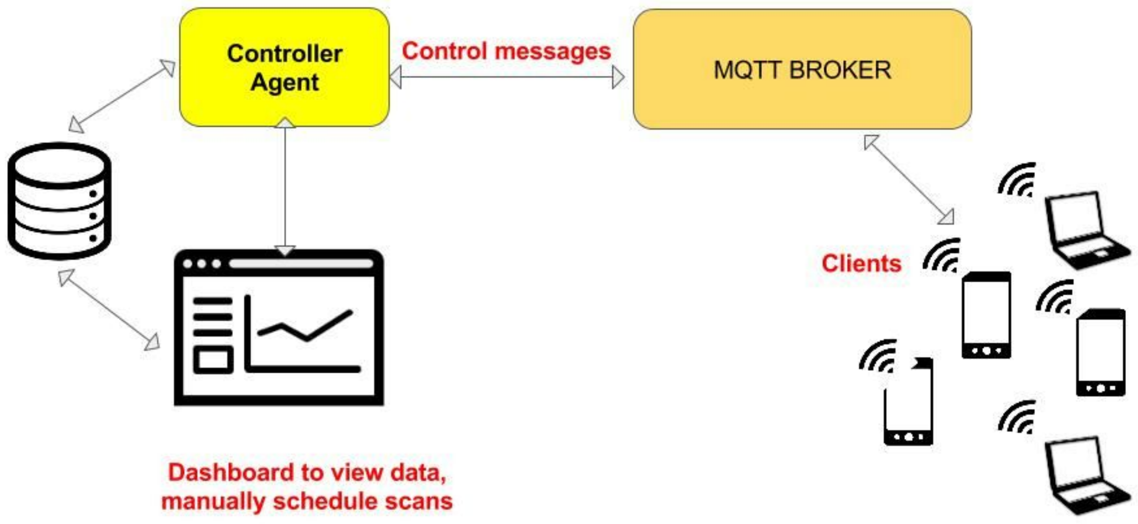
Crowdsourcing is a collective information gathering procedure whereby we have a system of agents who send data to a central server. At the heart of it is a distributed system where multiple asynchronous agents send data to a central coordinating agent in real time. Such systems face the problems such as a is there a bottleneck node which is causing the overall system to perform poorly, how well the system scales as new agents come and go, what happens if some elements of the system fail? In this project I have designed a system that will aims to solve these issues.

Problem statement:

The system I built is motivated from a practical problem I faced while designing a crowdsourcing platform for collecting spectrum availability data from mobile phones. The controller agent here is responsible to gather all the data sent from the mobile phones as well as send control commands to specific phones asking them to do certain tasks and send some specific information. For sending and receiving messages I use a popular pub-sub paradigm called Mqtt [6]In this paradigm the clients send data to a central agent called broker. The broker coordinates the message passing between the clients. Clients are decoupled in space as well as time. [7]

For the purpose of this project let us define the following two terms:

Clients : The agents (mobile/sensor devices etc.) that sends data to the mqtt broker.
Controller Agent: The entity that consumes all messages sent by the clients as well as sends control messages to the clients.
A very simple crowdsourcing architecture looks like this.



I want to address two problems:

a. The brokers must not become a single point of failure.
b. The controller agent must not be overwhelmed by the steady stream of messages sent from the client devices via the mqtt broker.

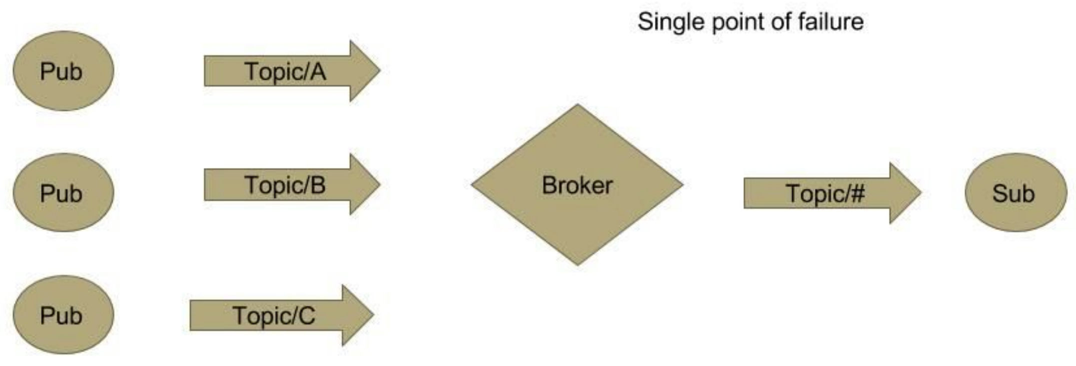
To address the problems by using 3 different architectures. They are:

1) Single Broker Single Controller
2) Multiple Broker Single Controller
3) Multiple Broker with Hybrid Clients and Kafka Cluster

Single Broker Single Controller

This is the simplest form of setup. Here we have a single broker a single controller agent and many client devices. The setup looks like this:

Single Broker Single Subscriber - A1



Setup and Code:

The setup files are present in the Arch1 directory.

1) *arch1.setup* : Instructions for setup of this architecture.
2) *Controller.py* : The implementation for the modified Controller agent.
3) *hive_mq.cfg* : Sample configuration for a single hive_mq broker.

Pros:

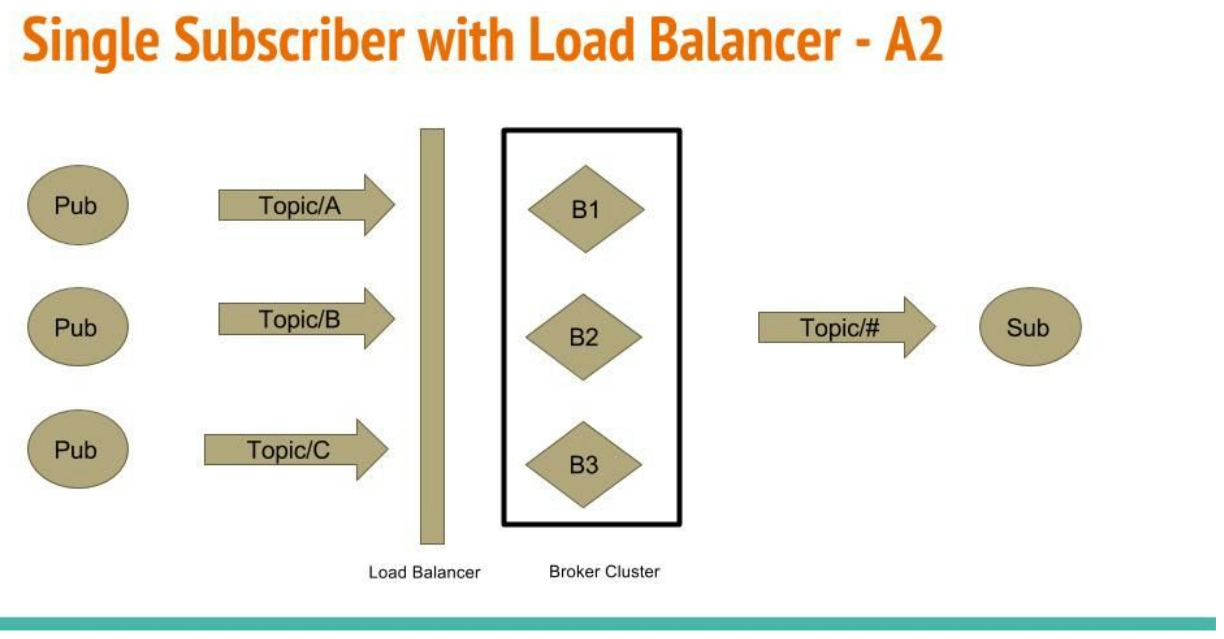
1) Simple to maintain and understand.

Cons:

1) Broker is the single point of failure.
2) Controller easily gets overwhelmed by the number of messages leading to message loss as the number of clients increases.

Multiple Broker Single Consumer Architecture:

In this architecture we solve the problem of the mqtt broker being a single point of failure by having cluster of Mqtt Brokers behind a tcp load balancer [2]. I used HAProxy [5][7] for this purpose. The benefit of this architecture is that the load balancer will distribute the load across the brokers. Even if one of the brokers fail the load balancer will reroute all traffic to it. Also since we have used the clustering functionality of HiveMQ [3] every broker knows about each other's subscriptions. So incase of one broker going down its subscriptions are handled by other broker. So there is no message loss although there is redundant messages sent between the clusters. This needs to be handled by client code. I handle it by keeping a set() of unique clients in the controller.



Setup and Code:

The code, configuration files and the setup instructions are all put in the directory **Arch2**

1) *arch2.setup* : Instructions for setup of this architecture.
2) *Controller.py* : The implementation for the modified Controller agent.
3) *haproxy.cfg* : Sample configuration for the load balancer.
4) *hive_mq.cfg* : Sample configuration for a single hive_mq broker.

Pros:

1) Solves the single point of failure problem as the brokers are clustered.

Cons:

1) All the upstream data passes through a single broker, the one with which the single controller is connected to. So this broker instance becomes a hot node and prone to fail. This solution does not address the fact that that the controller agent may get overwhelmed with data flows.

Multiple Broker with Kafka Cluster:

Architecture 2 solved the problem of single point of failure but failed to address one critical disadvantage of producer consumer based systems - that of the consumer being overwhelmed by the producer. The Controller agent in architecture 2 was handling all the “message_received” events from the broker cluster. With a large number of sensors sending messages to the broker it won't be long till the controller experiences message loss; the single controller simply cannot keep up; - this fact is validated by experiments that I ran with simulated clients. To address this problem I introduced a well known pub-sub tool named Apache Kafka [4] which is a distributed, highly scalable and fault tolerant replicated log system. Kafka sits between the *HybridClients* and the Controller. Kafka will act as a distributed producer - consumer buffer , thus solving the problem of the single *Controller Agent* being overwhelmed.

The glue between the Mqtt broker custer and the Kafka cluster is a thin client layer which I call *HybridClient*. There are multiple instances of the such *HybridClient* s running in the system. The *HybridClient* instances will subscribe to broker cluster for data messages sent by the sensors. So the broker clusters will push the messages to the *HybridClients*. Since there are multiple *HybridClients,* this ensures parallel consumption of the data messages and hence the problem of all messages passing through a single broker (the so called “hot node”) is removed here. The HybridClient s will continuously push messages to the kafka cluster.

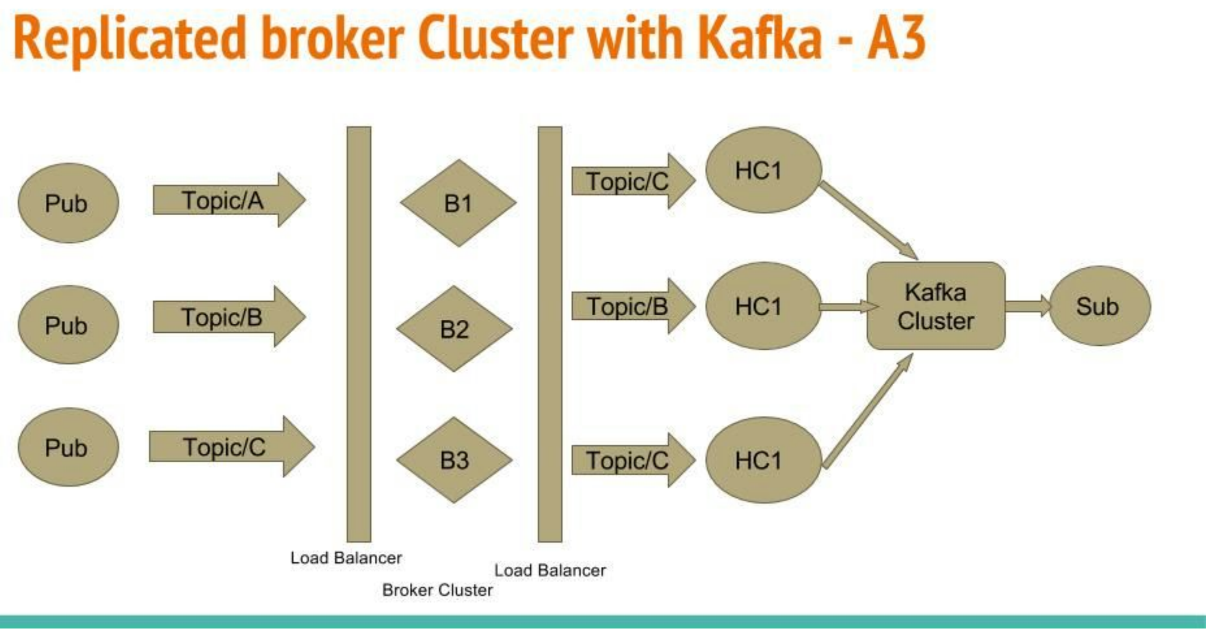
The *Controller agent* is also modified here. The *Controller* will continuously pull data, sent by the *HybridClients*, from the kafka cluster. The Controller agent can thus pull messages in its own pace and don't worry about the message loss as Kafka will persist those messages. This is the most complex of the three architectures.

In summary it is a 3-layered architecture where,

Layer 1 consists of the Mqtt Broker cluster that sits behind the HAProxy load balancer.
Layer 2 consists of the HybridClients that acts a glue between the broker and the Kafka

Layer 3 is the actual Controller which consumes messages from the Kafka cluster.

The design of the system was heavily inspired by Tim kellog's talk at the Apache Con at Denver, Colorado in April 2014. [1]



Setup and Code:

The code, configuration files and the setup instructions are all put in the directory **Arch3**

1) *arch3.setup* : Instructions for setup of this architecture.
2) *HybridClients.py* : The implementation for the hybrid controllers.
3) *Controller.py* : The implementation for the modified Controller agent.
4) *haproxy.cfg* : Sample configuration for the load balancer.
5) *hive_mq.cfg* : Sample configuration for a single hive_mq broker.

Pros:

1) Removes any single point of failure and presence of a over used mqtt broker instance (so called hot node).
2) Ensures Controller is not overwhelmed by the incoming messages.

Cons:

1) Complex architecture. Needs synchronisation among four (HAProxy, Mqtt, Kafka, Python) tool stack to get the setup working.

Testing and results:

For testing the architectures proposed above I wrote two generic scripts. One of which is a threaded python program that simulates a large number of sensor devices sending data to the broker. The second one is a script to measure the latency and the message loss. We measure message latency by lacing each data sent by the sensor node with the current timestamp and measure the difference in the time that this message reaches this message. We measure message loss as such: Each data sent by the client has a unique serial number associated with it. At the controller side I just find out if there are any missing messages.

Setup and Code:

The code for the testing codes are in the directory **PerformanceMeasurements**. It contains the following files:

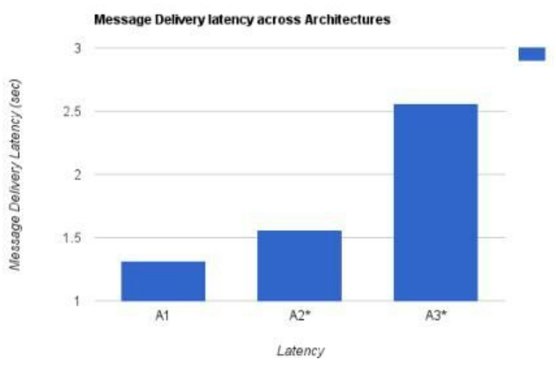
1) *UE_LoadSimulator.py* : This simulates a large number of client devices. The UE means User Equipment which is a jargon used in the cellular world and refers to a mobile phone.
2) *ConsumerStats.py* : Logs the latency and message loss.

I also tested the performance of the brokers using a custom plugin called JMX and JVM plugin that comes with HiveMq. This helps us to monitor the load on each broker in terms of CPU usage, heap space used, mesage dropped at the broker etc. A point to note is that messages can also be lost at the broker because the broker is ultimately a piece of software that handles a large number of incoming messages. But a commercial software like HiveMQ that has a threaded interface is likely to drop very less messages if the load is properly balanced and enough number of broker instances are provisioned. In this project I assumed that there were sufficient number of brokers and each machine running the broker sufficiently provisioned so that there is no message loss at the broker.

Results:

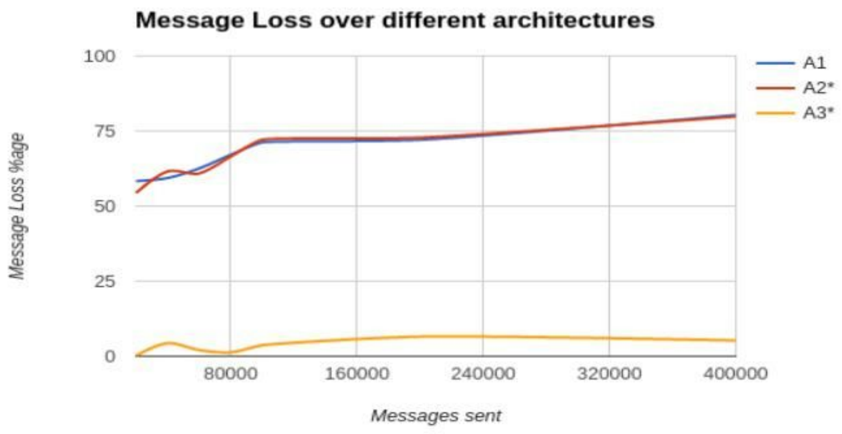
I obtained the following results regarding latency. The latency is minimum when the architecture is “Single Broker Single Controller”. As we increase the layers of software we introduce some latency. The latency is maximized in case of the last architecture. This is because of the multiple sleep statements in the HybridClients and Controller_Arch3 files. The sleep statements are needed to reduce the CPU usage. If we have a simple infinite while loop that does nothing it consumes a lot more CPU (almost 100%). But if we have a infinite loop with a sleep the CPU usage drops dramatically. In both the above mentioned files we have infinite loops pulling messages off the kafka cluster.

Message Latency Across Architectures



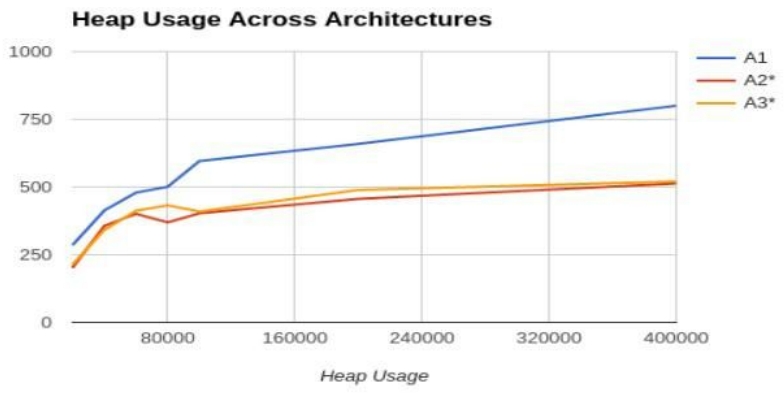
The message loss shows the dramatic improvement that is obtained by the third architecture. As we now have a broker cluster and a persistent replicated log we experience almost zero message loss. The only loss experienced is due to the loss at the broker, not at the controller. This can be alleviated by provisioning more hivemq nodes.

Message Loss Across Architectures



I also measured the heap usage which acts as a proxy for the stress experience by the brokers as we increase the number of clients. This gives us an insight into how we should provision the broker cluster.

Heap Usage in Mqtt Brokers



In summary we should weigh the tradeoffs when deploying such solutions. Trade off means to answer questions like how much do we want to sacrifice message loss for latency or message loss for system complexity? That depends on the type of application we are applying the methods above to.

References:

1) Scaling Mqtt , Tim Kellog , 12/3/2015
2) Medium io , Leylan Blog, 12/3/2015
3) Hivemq clustering , HiveMq Blog , 12/3/2015
4) Apache kafka, Apache Kafka Website, 12/3/2015
5) HAproxy, HaProxy website, 12/3/2015
6) Mqtt , Mqtt.org, 12/4/2015
7) Mqtt Essentials Hive Mqtt , Hive Mqtt Blog, 12/3/2015
8) Digital ocean setting up HAProxy