# Discovery and Scalable Processing of Robotics in the Cloud

Supun Kamburugamuve

Indiana University

Table of Contents

Discovery and Scalable Processing of Robotics in the Cloud 1

Introduction 3

Sensor Layer 4

Gateway 4

Sensor 4

Broker Layer 4

The Storm Layer 5

Sensor Discovery by Topologies 6

Parallelism 6

RabbitMQ & Kafka 6

RabbitMQ 6

Kafka 8

Turtlebot & Drone 9

Turtlebot application 9

Drone application 10

Future Work 11

## Introduction

The architecture of IoTCloud allows sensor data to be brought in to cloud services for real time processing. The architecture consists of three layers and they are

1. Gateway Layer
2. Publish-Subscribe messaging layer
3. Distributed stream processing layer

The data from the sensors are relayed through the gateways to the cloud-processing layer using Pub-Sub messaging middleware. We use topic based publish-subscribe brokers in the brokering layer.

To scale the system to thousands of sensors, we need a mechanism to discover the senors, scale the processing and distribute the load evenly across the resources. We use Apache Storm as our distributed stream-processing engine in the cloud for processing the sensor data in real time. The pub-sub brokering layer delivers the messages to Storm topologies.

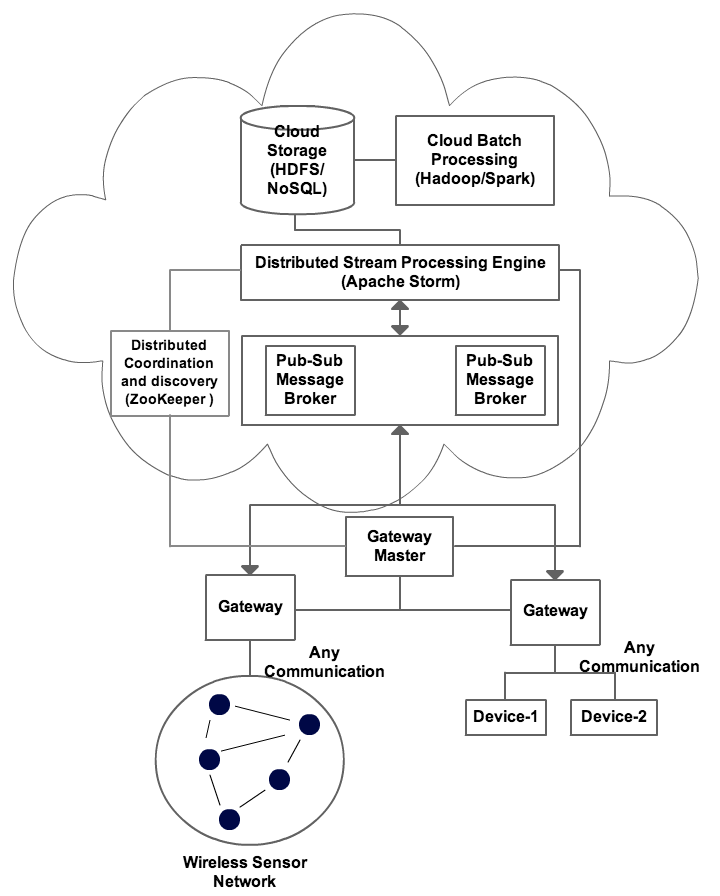


Figure 1 Architecture

## Sensor Layer

### Gateway

A Gateway host sensors and there can be many such gateways. Each gateway has a unique id to identify itself across the cluster. The gateways send the data from the sensor to the pub-sub message brokers. The gateways can use different message brokers to send the data. By default we support RabbitMQ, ActiveMQ and Kafka brokers. All the gateways are connected to a master gateway. The sensor information sent from the gateways are stored in the ZooKeeper by the master.

### Sensor

Sensor has a name and set of communication channels. When a sensor is deployed, the running instance gets an instance id. This instance id is used for controlling the sensor after the deployment. Same sensor can be deployed multiple times and each of the instances will get a unique id. A communication channel connects the data to the publish-subscribe messaging brokers. A sensor can have multiple such channels and each of these channels within a sensor has a unique name.

When a sensor is deployed, its information is saved in ZooKeeper. The default structure of sensor information in ZooKeeper is

/iot/sensors/[sensor\_name]/[sensor\_instance\_id]/[channel\_name]

The ZNode with the sensor instance id contains the information about the sensor like its status, metadata etc. The ZNodes with channel names contains the information about the channels.

The sensors can have two types of channels.

* Grouped channels: where same channel of same sensor in multiple instances grouped together
* Unique channels: Each instance of the sensor channel is separate

The unique channels are exclusive channels and provide direct communication between sensor and processing. The grouped channels are shared among multiple sensors.

## Broker Layer

The brokering layer consists of multiple message brokers. Each gateway can connect to cluster of brokers. In these brokers, topics are created for the channels defined in the sensors. Depending on the type of channel, the format of the name of the topic changes. The following gives the format of the sensors for the two cases

1. Unique Channel: site\_id.sensor\_name.channel\_name
2. Grouped Channel: site\_id.sensor\_name.sensor\_id.channel\_name

The pub-sub brokering layer is pluggable in our system and can use different brokers. We have done tests on pub-sub message brokers; ActiveMQ, RabbitMQ and Kafka. ActiveMQ and RabbitMQ are similar in design and functionality. RabbitMQ has a lower latency in transferring messages. So we chose RabbitMQ as our broker. Kafka has better parallelism support and clustering than RabbitMQ and ActiveMQ. But Kafka latency is much higher than RabbitMQ. The latency of Kafka can be improved by using better hardware so we choose to support Kafka as well because of its scalability.

## The Storm Layer

The data processing algorithms are written as Storm topologies. We use FutureGrid as our cloud platform for deploying the Storm Cluster. A Storm topology is a set of Spouts and Bolts written with the process algorithms and connected in a dag like structure. The components of this dag can be run in parallel in different computation nodes. The data enters a topology through Spouts and the processing happens in bolts. Pub-sub is a common pattern for ingesting data in to a Storm topology. Usually the last bolts in a topology write the results to a DB or send them to remote nodes using pub-sub messaging.

To ease the development of such topologies we allow the external communication points of a Storm Topology to be defined in a configuration file. In our case the external communications happen through pub-sub messaging. Here is an example of such configuration file for Kafka based topology. The topology has two external communication channels. One is the sentence receive spout where we get the input data from sensors and other is the count send where we send output information back to the sensors. We can use the above configuration to build the outer layer of a topology automatically. Still we need to connect the middle bolts using Java programming.

zk.servers: ["localhost:2181"]

zk.root: "/iot/sensors"

[topology.name](http://topology.name/" \t "_blank): "wordcount"

spouts:

    sentence\_receive:

        broker: "kafka"

        sensor: "wordcount"

        channel: "sentence"

        fields: ["sentence", "sensorID", "time"]

        properties:

          ackMode: "auto"

          broker.zk.servers: ""

          broker.zk.root: ""

bolts:

    count\_send:

        broker: "kafka"

        sensor: "wordcount"

        channel: "count"

        fields: ["count", "sensorID", "time"]

        properties:

          request.required.acks: "0"

          metadata.broker.list: "localhost:9090"

## Sensor Discovery by Topologies

A Topology is deployed with a number of parallel Spouts and Bolts that send and receive data from the pub sub brokers. We can configure the parallelism of a Spout or Bolt at the runtime as well. When a topology is deployed its external communication components (Spout and Bolts) doesn’t know about the physical addresses of the topics or how many topics it has to listen to. So at the very beginning the topology doesn’t have any listeners or message senders active. The topology knows that it has to exchange messages with a set of sensors.

The sensor information is saved in ZooKeeper by the gateways. The topology has information about the ZooKeeper and the sensors that it is interested in. The topology uses this information to dynamically discover the topics that it has to listen and add those listeners to the deployed components. The same process happens to bolts that send messages from the topology to brokers.

## Parallelism

The processing parallelism is achieved by running multiple instances of the same bolt or spout of the storm topology. When these components happen to be communicating with the message brokers the parallelism is tightly coupled with the message brokers.

Multiple instances of a Spout is created and run in parallel to read from a single sensor channel distributed across a cluster of brokers. This is same for bolts that send data out of topologies. But usually after the data is processed the output size becomes much smaller than the input size and parallelism is not required. But the system support full parallelism as in Spouts for bolts that send out pub-sub messages.

## RabbitMQ & Kafka

Now lets explore how the topic distribution happens in our two main message brokers. They are similar but there are fundamental differences in the approach.

### RabbitMQ

RabbitMQ has a rich API and architecture for connecting consumers and publishers. The RabbitMQ topics are easy to create and manage using its APIs. Those topics are light weigh and can be created without much burden to the broker. We allow both group channels and unique channels to be created for RabitMQ.

When we group the channels we group the channels from the same gateway in to one topic and doesn’t group channels across different gateways.

The following equations hold for group channels and unique channels deployed in RabbitMQ.

**Group Channels:**

**Unique Channels:**

The above equations are same for bolts.

The metadata about the messages are sent using RabbitMQ message headers. The metadata includes SensorID, siteId and custom properties. Figure 2 shows the distribution of topics and spouts for in a unique channel scenario. Here the channels are not grouped at the broker layer topics and . All these topics are evenly distributed among the spouts. The same holds for bolts as well.

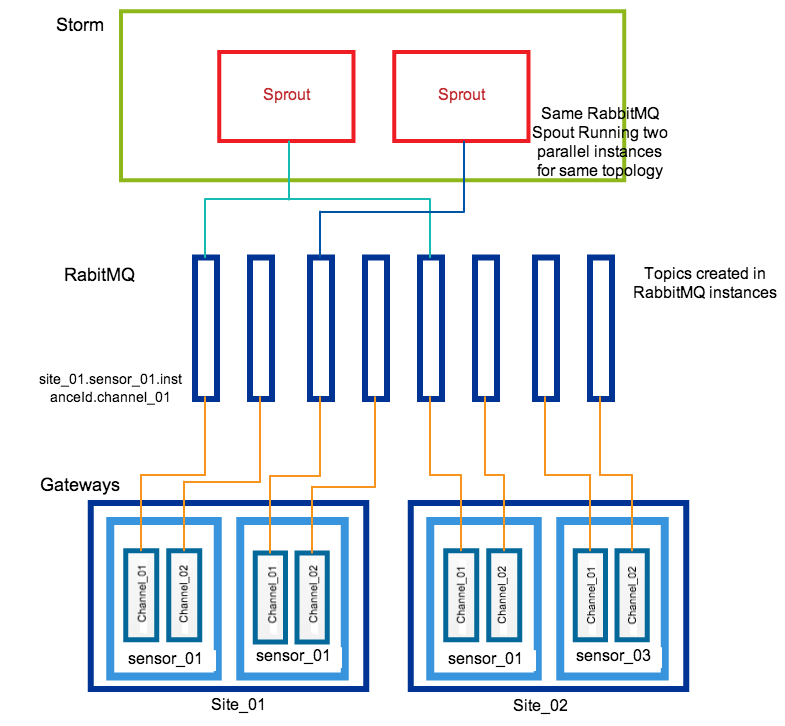


Figure Unique Channels with RabbitMQ

Figure 3 shows the distribution of topics for a grouped channel. In this case the . In this case also the topic listeners are distributed equally among the spouts.

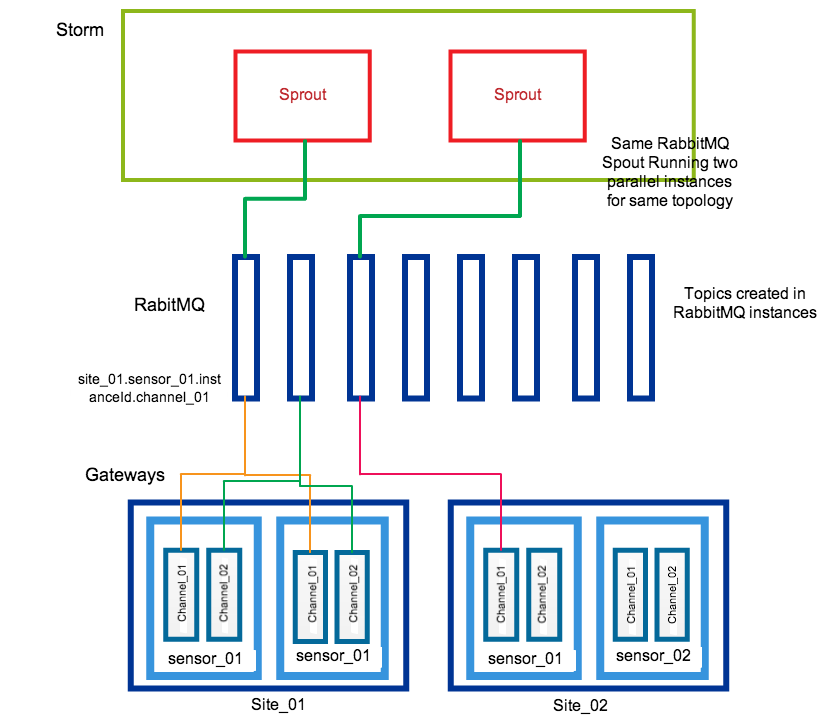


Figure Grouped Channels with RabbitMQ

### Kafka

The Kafka topics can be partitioned to support concurrent parallel reads and writes to the same topic. These partitions are stored across a Kafka cluster and can be replicated to have fault tolerance. A partition for a message is chosen using a message key. Because of this, messages with the same key goes to the same partition.

We need to send additional information about a message like the Sensor ID, site id and some properties. Because Kafka only supports byte messages without any headers we use a Thrift based message format to send the metadata about the message. We use the SensorID as the key of the message we send so that messages belonging to a single sensor instance will always be in one partition. This is important for our use cases because we need in-order delivery of the messages.

For Kafka we only support groped channels. This means that same channel belonging to the sensor instances deployed in one Gateway will get only one broker topic. Because Kafka topics can be partitioned we will have the parallel read capability and write capabilities for multiple sensors. For Kafka based topology the following holds true.

Figure 4 shows the distribution of topics for a Kafka based system. Here the . Because topics are partitioned we can have many spouts to read from the data and the possible read parallelism is much higher than that of RabbitMQ.

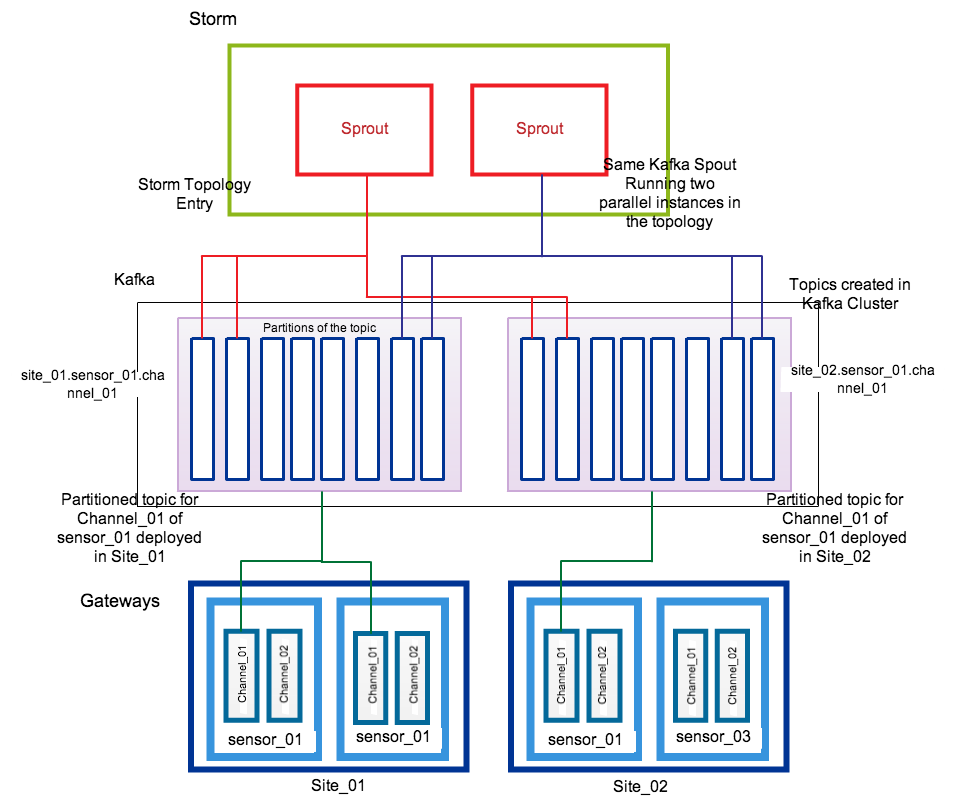


Figure Grouped Channels with Kafka

## Turtlebot & Drone

We are developing application for controlling a Turtlebot and Drone through our framework. In both applications the robots send data to the algorithms running in Storm. The algorithms running in cloud calculates the control commands and sends them back to the robots. The preliminary versions of sequential algorithms are written for the two robots.

### Turtlebot application

The turtlebot is a ground robot with a kinect cemera. The robot can move around and it can see both depth images and color images through the Kinect camera. In this application the turtlebot tries to follow a target in front of it by keeping a constant distance to it. We send Kinect depth images to the cloud and process this data in storm. The processing calculates new movement commands and sends them back to the robot. Figure 5 Show the latency observed for messages sent while running the Turtlebot application.

Figure Turtlebot Application Latency

Average Latency: 34.7

Standard deviation: 10.4

### Drone application

We are using Parrot Drone 2.0 for our application. The drone has a Camera facing downwards. In this application the drone to tracks a moving target on the ground and follow that target. The information captured by the drone is sent to a Cloud environment and processed there. The commands to control the drone is calculated in the Cloud and will be sent to the drone. The latency of the application over set of video frames is show in Figure 5.

Average Latency: 27.3

Standard deviation: 6.2

Figure Drone application latency

By increasing the memory and switching to concurrent mark sweep garbage collector the outliers reduced and the standard deviation also reduced. Figure 6 show the results.

Average Latency: 27.3

Standard deviation: 5.0

Added the following memory settings and the latency dropped significantly.

-Xms768m -Xmx768m -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled -XX:NewSize=300m -XX:MaxNewSize=300m -XX:MaxTenuringThreshold=1 -XX:SurvivorRatio=6

Average Latency: 14.7

Standard deviation: 4.7

## Future Work

We would like to do a performance evaluation of the current architecture and evaluate the different brokers and their scalability. Also we would like to do an evaluation of the Storm Topologies to handle large volumes of data with a upper bound in latency.