Streaming Applications for IOT with Real Time QoS

Supun Kamburugamuve, Geoffrey C. Fox
School of Informatics & Computing
Indiana University, Bloomington, IN USA

1. Introduction

Cloud Computing has long being identified as a key enabling technology for Internet of Things applications. In consideration of this we have developed an open source framework called IoTCloud[1] to connect IoT devices to cloud services. IoTCloud was funded as a part of research for AFOSR in Cloud-Based Perception and Control of Sensor Nets and Robot Swarms. It consists of: a set of distributed nodes running close to the devices to gather data; a set of publish-subscribe brokers to relay the information to the cloud services; and a distributed stream processing framework (DSPF) coupled with batch processing engines in the cloud to process the data and return (control) information to the IoT devices. Real-time applications execute data analytics at the DSPF layer to achieve streaming real-time processing. Our open-source IoTCloud platform uses Apache Storm[1] as the DSPF, RabbitMQ or Kafka as the message broker and an OpenStack academic cloud (or bare-metal cluster) FutureSystems as the platform. To scale the applications with the number of devices, we need distributed coordination among parallel tasks and discovery of devices, both achieved with a ZooKeeper-based coordination and discovery service. In this white paper we mainly focus on quality of service (QoS) for real-time applications in IoTCloud.

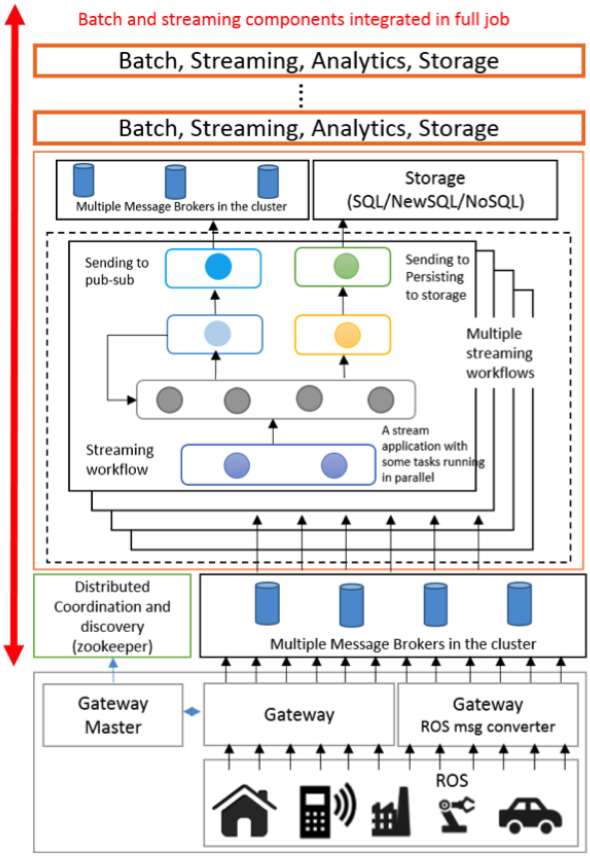

Figure 1 IoTCloud Architecture

In general a real-time application running in a DSPF can be modeled as a directed graph consisting of streams and stream processing tasks. Stream tasks are the nodes of the graph and streams are the edges connecting these nodes. A stream is an unbounded sequence of events flowing through the edges of the graph. The processing tasks at the nodes consume input streams and produce output streams. In our work, if a real-time application produces correct answers but violates timing requirements, we classify this as having performance faults. For most streaming applications, latency is of utmost importance and the system should be able to recover fast enough from faults for the normal processing to continue with minimal effect on the applications. We have developed several cloud-based robotics applications, including parallel simultaneous localization and mapping (SLAM)[2], collision avoidance[3] and image processing applications for robots such as TurtleBot and drones, all requiring strict message processing guarantees.

2. IoT Challenges for Streaming Applications

We present five categories of streaming applications based on challenges they pose to the backend Cloud control system.

1. Set of independent events where precise time sequencing is unimportant. **Example:** independent measurements from large number of temperature sensors.
2. Time series of connected small events where time ordering is important. **Example:** streaming audio or video; robot monitoring.
3. Set of independent large events where each event needs parallel processing with time sequencing not being critical. **Example:** processing images from telescopes or light sources with material or biological sciences.
4. Set of connected large events where each event needs parallel processing with time sequencing being critical. **Example:** processing high resolution monitoring (including video) information from robots (self-driving cars) with real-time response needed.
5. Stream of connected small or large events that need to be integrated in a complex way. **Example:** tweets or other

[1] https://storm.apache.org/

online data which we are using to update old clusters and find new ones rather than just classifying tweets based on previous clusters as in category 1), i.e. where we update models as well as using them to classify events.

These 5 categories can be considered in terms of single or multiple heterogeneous streams. Our initial work has identified difficulties in meeting real-time constraints in cloud-controlled IoT due to either the intrinsic time needed to process events or fluctuations in processing time caused by virtualization, multi-stream interference and messaging fluctuations. Figure 2 shows the fluctuations we observed with RabbitMQ and Kafka with minimal processing in Apache Storm. It also exhibits fluctuations in processing Kinect data in Storm from a Turtlebot with RabbitMQ. Large computational complexity in event processing is naturally addressed by using parallelism in the Storm bolts, although that can also lead to further sensitivity to fluctuations. With current technologies we can handle category 1) automatically and 3) with user-designed parallelism. The other cases require careful tuning on a case-by-case basis and we still can see unexpected large fluctuations in processing time that currently are not addressed except by over-provisioning.

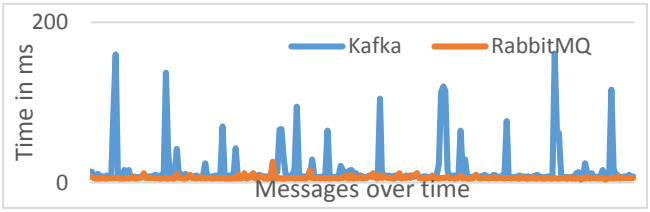

Figure 2 Fluctuations in Time of IoTCloud using RabbitMQ and Kafka with Minimal Processing in Storm

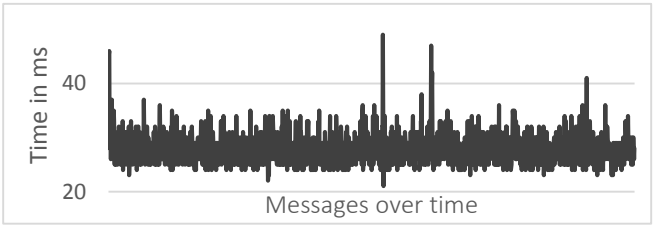

Figure 4 Fluctuation in Time of IoTCloud with processing Kinect data from TurtleBot with RabbitMQ

3. Next Steps

To reduce the fluctuations, we propose to duplicate some of the expensive computations as shown in Figure 3. In this architecture we dynamically replicate the streaming computation tasks within cloud clusters to achieve good performance in at least one replica. This replication will not be universal but rather only when achieving QoS demands that we do so, like when monitoring shows that the initial task is delayed. This will drastically reduce overhead from replication in many cases. We will dynamically identify the streaming tasks that require replication and apply it at the task level rather than the streaming application level. This dynamic replication of streaming tasks will be implemented for Apache Storm as described above. To dynamically increase the Storm servers, we will use a resource manager such as Apache Yarn coupled with the IaaS layer.

For the processing stage in Fig. 2, the fluctuations in time at the broker are much less pronounced in RabbitMQ than in Kafka. We will scale the brokers at runtime to minimize such effects to the system by monitoring performance of brokers. Then a controller will directly use the IaaS infrastructure to scale the brokers as needed by increasing the number of assigned VMs.

To scale an application that receives input from multiple sources as a single stream and needs to differentiate each source, the larger stream must be partitioned into sub-streams according to the source. This can be achieved with current frameworks, but when parallel processing and state tracking are

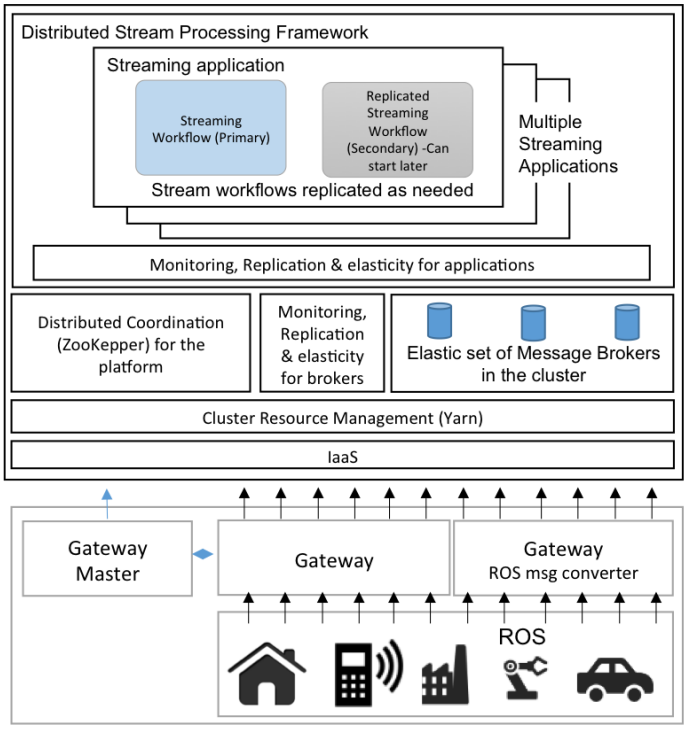

Figure 3 Extended Architecture of IoTCloud

needed, the application code becomes complex. Having tasks running in the same process with threads can make the performance of individual tasks unpredictable. Thread-based parallelism and process-based parallelism have to be examined carefully to choose the best suitable strategy for obtaining adequate QoS guarantees. Scheduling must take subtask parallelism, thread and process-based parallelism into account.

Having efficient communication among the streaming tasks can reduce the network congestion, thus leading to better QoS in the overall system. Also having such efficient communication reduces the latency of the application, which is a quality of service itself. We have found that the current DSPFs available lack efficient communication, which is perfected by the HPC community for operations such as broadcasting. We have adapted these algorithms for Storm, taking into consideration

both cloud runtime environments and the nature of streaming applications. The graph in Fig. 5 shows the latency when using a tree-based broadcasting algorithm and the naïve implementation. With the naïve implementation the broadcasting node send messages to each worker separately.

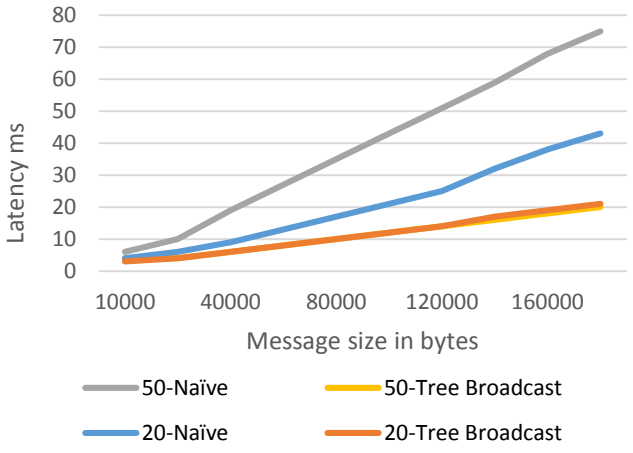

Figure 5. The overall latency of the application when using Naive broadcast implementation and Tree-based broadcast. Different lines correspond to different parallel tasks with the two implementations. 50 and 20 Tree broadcast lines are on top of each other.

The application used in this experiment has one node broadcasting to parallel tasks (20 and 50), which does synchronous processing. Its data flow is *RabbitMQ* → *Broadcast* → *Parallel Workers* → *Gather* → *RabbitMQ*. We are running Storm on an 8-Node cluster, with each node having 4 worker processes. The broadcasting algorithm uses a tree with branching factor equal to the number of nodes for the first level and a branching factor of 2 afterwards. Figure 5 clearly shows that a significant gain can be achieved with the new broadcasting algorithm and latency doesn't increase when increasing the parallel workers. We are continuing to investigate the behavior of different algorithms and settings for synchronous and asynchronous parallel processing streaming applications.

References

1. Kamburugamuve, Supun, Leif Christiansen, and Geoffrey Fox. "A Framework for Real Time Processing of Sensor Data in the Cloud." *Journal of Sensors*2015 (2015).
2. Kamburugamuve, Supun, Hengjing He, Geoffrey Fox, and David Crandall. "Cloud-based Parallel Implementation of SLAM for Mobile Robots".
3. He, Hengjing, Supun Kamburugamuve, and Geoffrey C. Fox. "Cloud based real-time multi-robot collision avoidance for swarm robotics."