A Framework for Real-Time Processing of Sensor Data in the Cloud

Supun Kamburugamuve
skamburu@indiana.edu
Leif Christiansen
grindvald@gmail.com
Geoffrey Fox
gcf@indiana.edu
School of Informatics and Computing and Community Grids Laboratory
Indiana University, Bloomington IN 47408 USA

Abstract: In this paper we describe IoTCloud; a platform to connect smart devices to cloud services for real time data processing and control. A connected device to IoTCloud can communicate with real time data analysis frameworks deployed in the cloud via messaging. The platform design is scalable in connecting devices, transferring and processing data. A user develops real time data processing algorithms in an abstract framework without concern for underlying details of how the data is distributed and transferred. For this platform we primarily consider real time robotics applications such as autonomous robot navigation, where there are strict requirements on processing latency and demand for scalable processing. To demonstrate the feasibility of the system, a robotic application is developed on top of the framework. The system and the robotics application characteristics are measured to show that data processing in central servers is feasible for real time sensor applications.

1. Introduction

The availability of internet connections and low manufacturing costs have led to a boom in smart objects, devices with a tripartite construction consisting of a CPU, memory storage, and a wireless connection. These smart objects (or devices) are equipped with sensors that produce data and actuators that are capable of receiving commands. Such devices are widespread in all the fields and usages are expected to grow exponentially in the future. For these devices, central data processing has been shown to be advantageous due to numerous factors including: the ability to easily draw from vast stores of information, efficient allocation of computing resources and a proclivity for parallelization. Because of these factors, many devices may benefit from processing only some data locally and offloading the remaining processing to central servers. Among the aforementioned devices, and increasingly present in modern life, are robots. Robots such as the iRobot Roomba, a robot that can clean the floor, present affordable, automated aids for daily living. Additionally, Amazon and Google are researching and developing platforms for delivering consumer products using drones. Most of these robots have limited onboard processing power but still generate large amounts of data. Cloud based analysis of data coming from such robots creates many challenges due to strict latency requirements and high volumes of data production.

To process data coming from many smart devices, we need scalable data processing platforms. Cloud is an ideal computational platform for hosting data processing applications for smart devices, because of its efficiency and agility. The. Cloud computing[1] refers to both applications delivered as services over the Internet and the hardware and system software in the datacenters that provide those services. Cloud computing enables computing as a utility and is gradually becoming the standard for computation, allowing the systems and users to use Platform as a Service (PaaS), Infrastructure as a Service (IaaS), and Software as a Service (SaaS). The computational nodes are provisioned, configured and reconfigured dynamically in the cloud. These machines can be in the form of virtual machines or physical machines. Furthermore, sensor based applications can benefit from in-house private cloud environments hosted within organizations or from public clouds hosted by large organizations.

In order to process data generated by smart devices in a cloud environment, the data must be transmitted from the devices to the cloud in an efficient and scalable manner. The communication between cloud applications and the devices is essentially based on events, which suggests that the traditional request/response approach is not appropriate. For example, when using requests and responses, a device requiring real time control has to poll the applications continuously. Continuous polling increases the latency and network traffic. Transmission of events is well supported by publish-subscribe messaging[2] where a publisher makes information available to subscribers in an asynchronous fashion. Over time Publish-Subscribe messaging has emerged as a distributed integration paradigm for deployment of scalable and loosely coupled systems. Subscribers have the ability to express their interest in an event, or a pattern of events, and are subsequently notified of any event, generated by a publisher, which matches their registered interest. An event is asynchronously propagated to all subscribers that registered interest in that given event and subscribers. Publish-Subscribe messaging decouples the message producers and consumers in the dimensions of time, space and synchronization. The decoupling favors the scalability of the message producing and consuming systems. Because of these features, publish-subscribe messaging is being proposed as a good fit for connecting smart devices to cloud applications.

Topic based and content based are two different widely used schemes of pub-sub systems. In topic based systems the messages are published to topics which are identified by keywords. The consumers subscribe to topics and receive messages coming to these topics. In content based systems the consumers subscribe to messages based on the properties of the messages. This means the content of each message has to be examined at the middleware to select a consumer among possibly a large set of consumers. Because of the simple design of topic based middleware, they tend to scale well compared to content based brokers and introduces less overhead.

We envision a cloud based data intensive computing architecture where stream based real time analysis and batch analysis are combined together to form a rich infrastructure for sensor applications. We propose Cloud DIKW (Data, Information, Knowledge, Wisdom) based architecture for sensor data analysis in the cloud. The high level DIKW view of the system is shown in Figure 1.

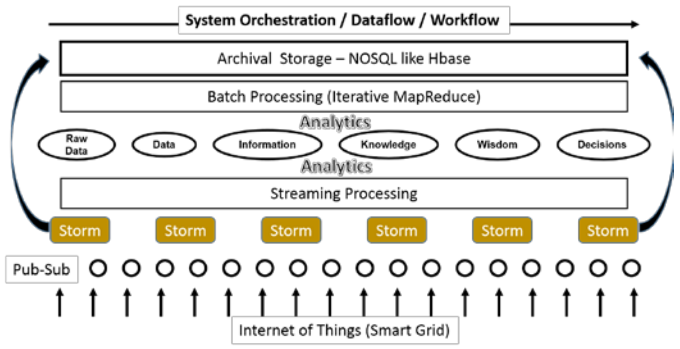

System Orchestration / Dataflow / Workflow
Archival Storage – NOSQL like Hbase
Batch Processing (Iterative MapReduce)
Analytics
Raw Data
Data
Information
Knowledge
Wisdom
Decisions
Analytics
Streaming Processing
Storm
Storm
Storm
Storm
Storm
Storm
Pub-Sub
Internet of Things (Smart Grid)

Figure 1 DIKW View of the System

We can assume that for all our devices, data is sent to cloud as a stream of events. It is important to process the data as a stream (before storing it) to achieve real time processing guarantees. Parallel processing of events coming from a single source can help to reduce the latency in most applications. The ability to connect large number of devices creates a need for a scalable infrastructure to process the data. Distributed event processing engines (DSPEs)[3-6] are a good fit for such requirements. A DSPE abstracts out the event delivery, propagation and processing semantics and greatly simplifies the real time algorithm development. Also a DSPE can act as a messaging fabric that distributes data to other data sinks like databases, file systems etc.; for batch processing and archival purposes, after some pre-processing of the data.

By combining the above requirements, we have developed our IoTCloud platform, which is a distributed software platform capable of connecting devices to the cloud services. IoTCloud uses topics based publish-subscribe messaging to transfer data between the devices and the cloud services and uses a DSPE to process the data in the cloud. The platform supports two publish-subscribe brokers with different semantics that are suitable for different applications. We have developed a robotic application that runs through a private in house cloud to demonstrate how to use the system and measured the characteristics of the system in order to show that we can do real time processing of sensor data in a cloud environment in a scalable manner. The main contribution of our work is to explore scalable cloud based real time data processing for sensor applications.

Section 2 of the paper describes the related work in this area. Section 3 explains the architecture of the framework and section 4 the robotics application we have developed. Next in section 5, we present a series of tests we have done to evaluate the system and discuss the observations. Finally, in section 6 and 7 we conclude the paper with conclusions and future work.

2. Related Work

To best of our knowledge, connecting devices to cloud services for real time processing in a scalable manner is not addressed in the literature. Also, work related to cloud based control of robots is largely lacking. Hassan[7] is a content based publish/subscribe framework for connecting sensor data to cloud services. Content based pub-sub allows greater flexibility for the application designers than topic based pub-sub systems. But content based pub-sub systems usually involves higher overhead than topic based pub-sub systems because the brokers has to inspect message content. Furthermore, content based pub-sub brokers are not popular and are not widely available as products.

Mires[8], TinySIP[9], DV/DRP[10] are all publish/subscribe messaging middleware for WSNs. They address the different issues in connecting WSNs and communicating with sensors. MQTT-S[11] is an open topic-based pub-sub protocol defined for transferring data from sensors. The protocol enables data transfer between sensors and traditional networks. In our work we assume that sensor data is available to be transported to cloud services and we handle the transferring of gathered data from devices to cloud services. For example, a device connected to our system can send data via a dedicated communication channel, public Internet etc. Also many devices can be connected in WSNs using above mentioned protocols or messaging systems and our platform can transfer this data to cloud services for processing.

Reference architectures for integrating sensors and cloud services have being discussed in the literature[12, 13]. Both works explore the general architecture that can be used to connect the sensors to cloud services and the potential issues. In our work we provide a framework that can be used to send sensor data from devices to the cloud as well as show how to process the data within a generic framework. We also discuss how to transfer data from devices to cloud services and process it in a scalable way, topics that are not fully addressed in above papers.

3. IoTCloud Architecture

A system view of the architecture is shown in Figure 2. Our architecture consists of three main layers.

1. Gateway Layer
2. Publish-Subscribe messaging layer
3. Cloud based big data processing layer

We consider a device as a set of sensors and actuators. Users develop a driver that can communicate with the device and deploys it in a gateway. This driver doesn’t always have to directly connect to the device. For example, it can connect to the device via a TCP connection or through a message broker. The data generated by the driver application is sent to the cloud-processing layer using publish-subscribe messaging brokers. The cloud processing layer processes the data and sends control messages back to the driver using the message brokers. The driver converts the information to a format that suites the device and communicates this to the device. The platform is implemented in the Java programming language.

3.1 Gateway: Drivers are deployed in gateways responsible for managing drivers. There can be multiple Gateways in the system and each gateway has a unique id. A Gateway master controls the gateways by issuing commands to deploy/un-deploy, start/stop drivers etc. A Gateway is connected to multiple message brokers and these brokers can be in a cluster configuration. By default the platform supports RabbitMQ[14], ActiveMQ and Kafka[15] message brokers. Gateways manage the connections to the brokers and handle the load balancing of the device data to the brokers. Gateways update the master about the drivers deployed in it and status of the gateways. Master stores the state information in a ZooKeeper[16] cluster.

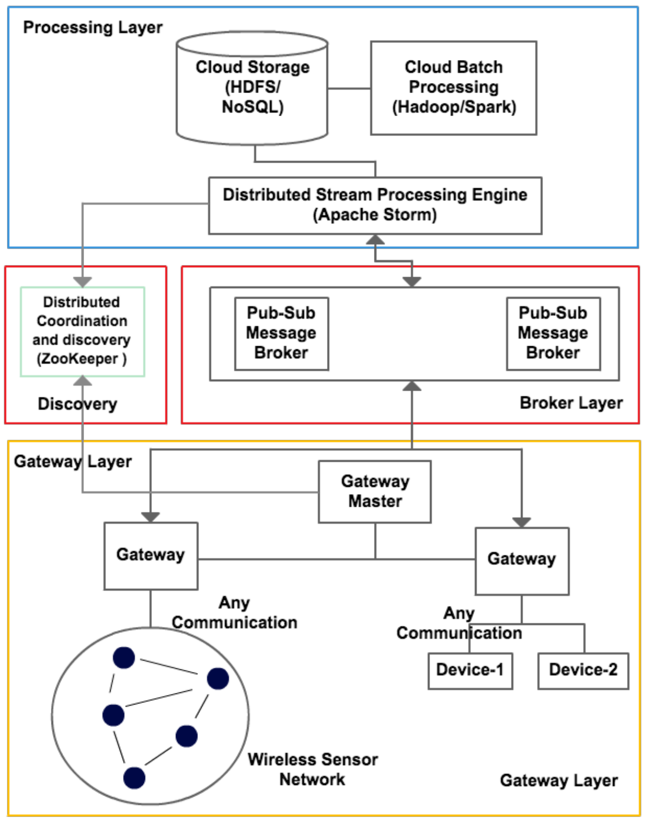


Figure 2 IOTCloud Architecture

3.2 Driver: The driver is the data bridge between a device and the cloud applications. The driver converts data coming from the device to a format that the cloud applications expect and vice versa. A driver has a name and a set of communication channels. When a driver is deployed, the running instance gets an instance id. This instance id is used for controlling the driver after the deployment. The same driver can be deployed multiple times and each of the instances receive a unique id. A driver can have multiple communication channels and each channel within a driver has a unique name. A communication channel connects the driver to publish-subscribe messaging brokers. When a driver is deployed, its information is saved in ZooKeeper. The default structure of driver information in ZooKeeper is:

```
/iot/sensors/[driver_name]/[driver_instance_id]/[cha
nnel_name]
```

A zookeeper node (ZNode) with the driver instance id contains information about the driver like its status, metadata etc. ZNodes with channel name contains the information about the channels. The framework allows shared and exclusive channels to be created. An exclusive channel can give faster communication between the drivers and the cloud processing. But in large-scale deployment of drivers, an exclusive channel can result in large number resources in the brokers. Some applications don’t have strict latency requirements and can use shared channels consuming less system resources.

3.3 Brokers: The platform specifically focuses on Topic-Based publish-subscribe brokers rather than content-based publish-subscribe brokers. We chose Topic based brokers due to several reasons. 1. Stable, open source topic based brokers are available 2. Topic based brokers are simple to use and configure 3. The overhead introduces by the broker is minimal compared to content based brokers. For this project the most important factors are 1 and 3, because our applications require low latency and topics based brokers are the ones readily available for use. The messaging layer needs to preserve the message ordering preventing multiple consumers consuming messages from the same driver.

There are many open source brokers available that full fill our needs for the messaging infrastructure. Such brokers includes ActiveMQ[17], RabbitMQ[14], Kafka[15, 18] Kestrel, HonertMQ etc. From these brokers ActiveMQ, RabbitMQ and Kafka are widely used topic based publish subscribe brokers. The preliminary studies showed that ActiveMQ and RabbitMQ have identical functionalities for our purposes and latter is capable of handling more load with less overhead. So we decided to use RabbitMQ. The Kafka broker has very good clustering capabilities and can handle parallel consumer reads for the same Topic. So we decided to support both these brokers in our platform.

Each communication channel created in a driver is connected with a topic created in the message broker. The framework supports two mappings of channels to topics hence creating two types of channels. In the first type, each channel is mapped to a unique queue in the broker. We call this type of channels exclusive channels. In the other type of channel, a set of channels share the same topic in the broker. This type of channel is called a shared channel. At the moment we use a very simple rule to map the channels to a shared queue. We map the same channel from multiple instances of a driver deployed in one gateway to a single topic.

For shared channels: $No\ of\ topics = No\ of\ Gateways$
Exclusive channels: $No\ of\ topics = No\ of\ Driver\ instances$

For a shared channel, corresponding topic name is of the format of "gateway_id.driver_name.queue_name". For an exclusive channel, topic name is of the format of "gateway_id.driver_name.driver_id.queue_name".

RabbitMQ: RabbitMQ is a message broker primarily supporting Advanced Message Queuing Protocol (AMQP)[19]. Even though the core of RabbitMQ is designed to support AMQP protocol, the broker has been extended to support other message protocols like STOMP, MQTT etc. RabbitMQ is written in the Erlang programing language and supports low latency high throughput messaging. RabbitMQ has a rich API and architecture for developing consumers and publishers. RabbitMQ topics are easy to create and manage using its APIs. These topics are light weight and can be created without much burden to the broker. We allow both shared channels and exclusive channels to be created for RabbitMQ. The metadata about the messages are sent using RabbitMQ message headers. The metadata includes sensor id, gateway id and custom properties.

Kafka: Kafka is publish-subscribe message broker backed by a commit log. The messages sent by the producers are appended to a commit log and the consumers read the messages from this commit log. Kafka implements its own message protocol and does not support standard protocols like AMQP or MQTT. At the core of Kafka messaging is the concept of a Topic. A topic is divided into multiple partitions and a message is sent to a single partition. In our platform, partition for a message is chosen using a key accompanying a message. So messages with the same key go to the same partition. Consumers consume messages from partitions. Partitions of a single topic can spread across a cluster of Kafka servers. Furthermore, a single partition is replicated in a Kafka cluster for reliability. Kafka guarantees ordering of messages in a partition and doesn't guarantee ordering across partitions. Because a topic consists of multiple partitions, consumers can read from the same topic in parallel without affecting the message ordering for a single message key. In IoTCloud platform we use the driver id as the key for a message.

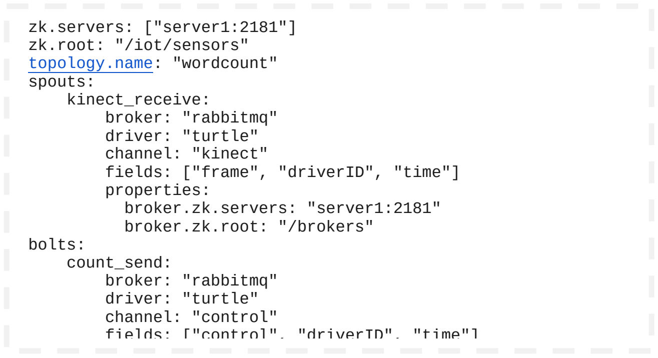
In IoTCloud we need to send metadata with a message such as the driver id, site id and some properties. Because Kafka only supports byte messages without any headers, we use a Thrift[20] based message format to send metadata about the message. Use of driver id as the key, makes sure that the messages belonging to a single driver instance will always be in one partition. We use at most one consumer per partition to ensure the message ordering for a driver. Because Kafka topics can be partitioned we will have parallel read capability and write capabilities for shared channels. Because of this, the platform only support shared channels for Kafka.

3.4 Cloud Processing: As the primary cloud-processing framework we are using Apache Storm[6], which is an open source DSPE. There are many DSPEs available but we chose Storm because of its scalability, performance, excellent development community support and the ability to use scripting languages to write its applications. Storm can be used to process the data and send responses back immediately or it can be used to do some pre-processing of the data and store them for later processing by batch engines such as Apache Hadoop. The applications we have developed doesn't uses batch processing at the moment, so we haven't incorporated such engines to the platform yet but our architecture permits integration of engines like Hadoop. We use FutureGrid[21] as our cloud platform for deploying the Storm Cluster. Futuregrid has an OpenStack based could implementation and we provision VM images using the OpenStack tools.

Apache Storm: Storm is a distributed stream processing engine designed to process large amounts of streaming data in a scalable and efficient way. Data processing applications are written as Storm topologies. A topology defines a DAG structure for processing the streaming data coming from the devices as an unbounded stream of tuples. The DAG consists of a set of Spouts and Bolts written to process the data. The tuples of the stream flow through the nodes (Spouts and Bolts) of the DAG. Spouts and Bolts are primarily written in Java but other programming languages like Python, Ruby is permitted. Data enters a topology through Spouts and the processing happens in bolts. The components in the DAG are connected to each other using stream (tuple) groupings. Pub-sub is a common pattern for ingesting data in to a Storm topology. A bolt can consume the connected input streams, do some processing on the tuples and generate and emit new tuples to the output streams. Usually the last bolts in the topology DAG write the results to a DB or send the results to remote nodes using pub-sub messaging. The spouts and bolts of a topology can be run in parallel in different computation nodes.

To ease the development of Storm topologies in our platform we allow the external communication points of a Storm Topology to be defined in a configuration file. Figure 3 is an example of such configuration file. The topology has two external communication channels. A "kinect_receive" spout is where we get the input data from devices and a "count_send" bolt is where we send output information back to the devices. We can use the above configuration to build the outer layer of a topology automatically. The algorithm has to be written by the application developer.

We can run many instances of any of the components in a Storm Topology in parallel. For example to read data parallelly from many devices, we can spawn several instances of the kinect_receive spout in different nodes. This can be done for any bolt in the topology as well. The parallelism can be changed at runtime as well. This allows the system to scale with the addition of drivers.



```
zk.servers: ["server1:2181"]
zk.root: "/iot/sensors"
topology.name: "wordcount"
spouts:
    kinect_receive:
        broker: "rabbitmq"
        driver: "turtle"
        channel: "kinect"
        fields: ["frame", "driverID", "time"]
        properties:
          broker.zk.servers: "server1:2181"
          broker.zk.root: "/brokers"
bolts:
    count_send:
        broker: "rabbitmq"
        driver: "turtle"
        channel: "control"
        fields: ["control", "driverID", "time"]
```

Figure 3 Topology Endpoint Configuration

3.5 Discovery: Because Storm is a parallel processing framework, it requires coordination among the processing units. For example when a communication channel is created in the broker for a device, the parallel units responsible for communicating with that channel should pick a leader because multiple units reading from the same channel can lead to data duplication and out of order processing, which is not desirable for most applications. Also the distributed processing units should be able to detect when the drivers come online and go offline. To adapt to such a distributed dynamic processing environment we need discovery and coordination. We use Apache ZooKeeper[16] for achieving both. When drivers come online the information about the drivers is saved in the ZooKeeper. The discovery component discovers and connects this information to the cloud processors dynamically at runtime. This allows the processing layers to automatically distribute the load and adjust accordingly to the changes in the data producer side.

A storm Topology is deployed with a number of parallel Spouts and Bolts that send and receive data from the pub-sub brokers. We can change the parallelism of a Spout or a Bolt at the runtime as well. When a topology deploys its external communication components (Spout and Bolts) does not know about the physical addresses of the topics or how many topics they have to listen to. So at the very beginning the topology does not have any active message listeners or senders. The topology knows that it has to exchange messages with a set of drivers deployed in the gateways. The topology has information about the ZooKeeper and the drivers that it is interested in. It uses this information to dynamically discover the topics that it has to listen and add those consumers and producers to the topology at runtime

3.6 Processing Parallelism: The processing parallelism at the endpoints of the topology is bound to the message brokers and how we can distribute the topics across the brokers. For middle processing bolts, maximum parallelism is not bounded and depends on the application. A Storm topology gets its messages through the spouts. Same spout can run multiple instances in parallel to read the messages coming from multiple devices connected to the system. A spout always reads the messages from a single channel of a device. If a processing algorithm requires input from multiple channels, the topology must have multiple spouts. A running instance of a Spout can connect to multiple topics to read the messages, but all these topics must be connected to a channel with the same name and driver. When a spout needs to read from multiple topics, the topology distributes the topics equally among the running instances of the spout dynamically at the runtime. The message flow through the Storm topology happens primarily using the driver ids. The bolts that are communicating with the brokers know about all the topics in the system and they can send a message to an appropriate topic by selecting the correct topic using the driver id.

RabbitMQ: There is a limit to the number of parallel spouts that we can run due to the number of topics created per channel. Following gives an upper bound on number of spouts we can run when RabbitMQ brokers are used.

Shared Channels: $No\ of$ parallel $spouts\ \leq No\ of\ gateways$
Exclusive Channels: $No\ of$ parallel $spouts\ \leq No\ of\ channels$

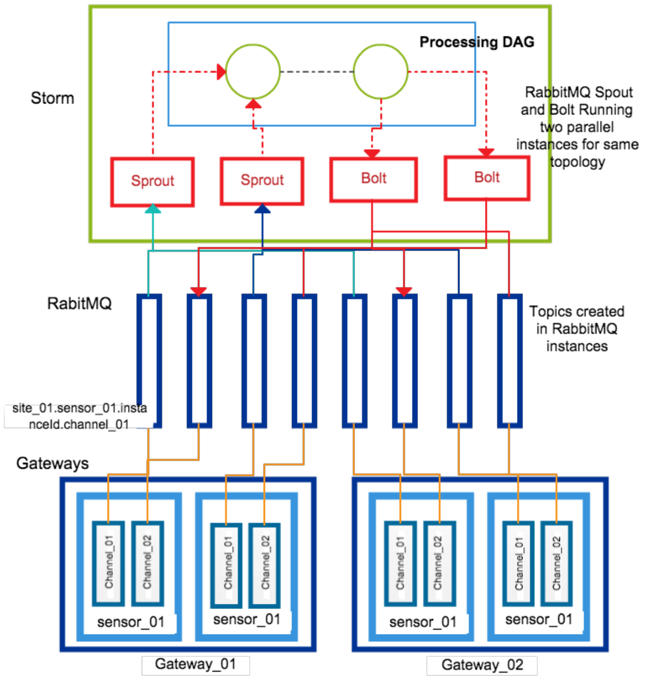

Storm
Processing DAG
RabbitMQ Spout and Bolt Running two parallel instances for same topology
Sprout
Sprout
Bolt
Bolt
RabitMQ
Topics created in RabbitMQ instances
site_01.sensor_01.instanceId.channel_01
Gateways
Channel_01
Channel_02
sensor_01
sensor_01
sensor_01
sensor_01
Gateway_01
Gateway_02

Figure 4 RabbitMQ Exclusive Channels & Storm

In general we cannot do parallel reads from a topic due to the ordering constrains. Figure 4 shows how exclusive channels created by a driver named sensor_01 is connected to the storm topology. Here, the storm topology runs only one instance for each spout reading from channel_01 and channel_02. Because we have 8 channels in 4 instances of the drivers, we need 8 topics in the broker. Because we only have 2 spouts and 2 bolts in the topology, each spout is connected to 2 topics and each bolt is communicating with 2 topics. Figure 5 shows the same scenario with shared channels. In this case we only have 4 topics because the two drivers deployed in the same gateway are using the same topics.

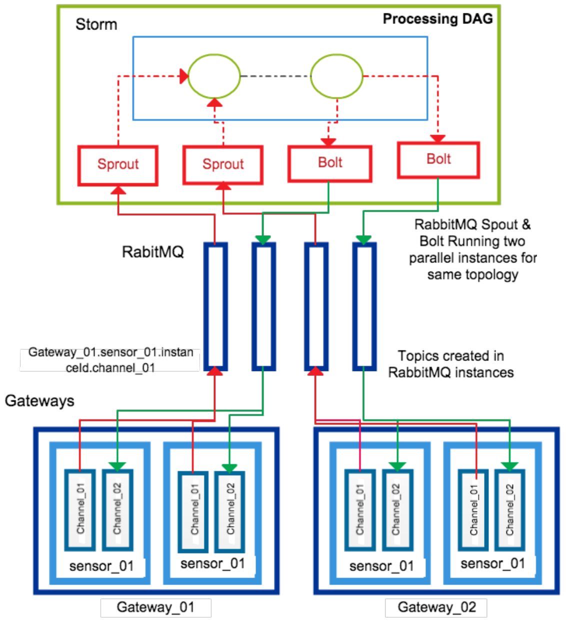

Storm
Processing DAG
Sprout
Sprout
Bolt
Bolt
RabitMQ
RabbitMQ Spout & Bolt Running two parallel instances for same topology
Gateway_01.sensor_01.instanceld.channel_01
Topics created in RabbitMQ instances
Gateways
Channel_01
Channel_02
sensor_01
Gateway_01
Gateway_02

Figure 5 RabbitMQ Shared Channels & Storm

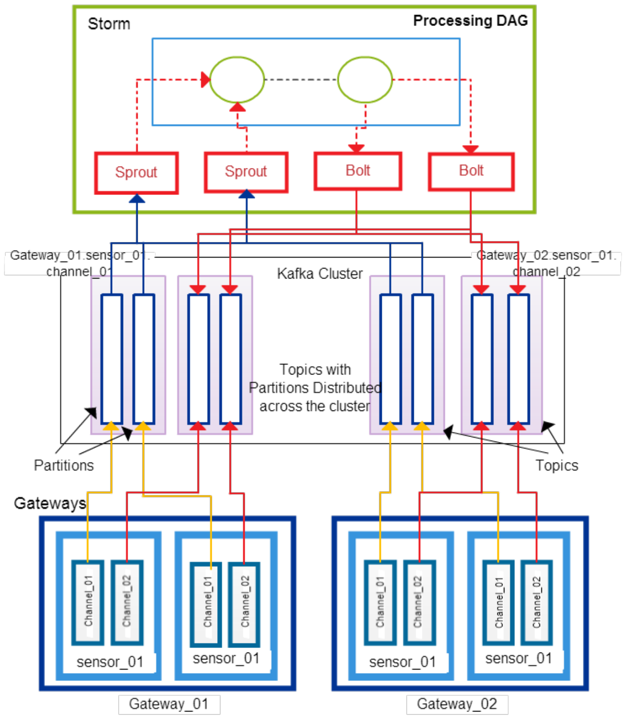


Figure 6 Kafka Shared Channels & Storm

Kafka: Kafka topics are more heavy weight than RabbitMQ. For every topic in the system, Kafka has to create a log files and index files in the file system for its partitions. If the replication is enabled for fault tolerance, these files have to be replicated in the Kafka cluster. Kafka also supports parallel reads for a single topic. Because of these reasons we only support shared channels for Kafka. In Kafka the number of spouts possible depends on the number of partitions for a topic.

$$No\ of\ parellel\ spouts \leq No\ of\ gateways \times Partitions\ per\ Topic$$

Figure 6 shows topics distribution with Kafka for the same scenario as in Figure 4. In the Figure 6 each Kafka topic has 2 partitions and we have 4 topics because the channels are shared. Because each topic has two partitions, read and write parallelism in this case is equal to the exclusive channel scenario with RabbitMQ (Figure 5). But in practical scenarios we will have less number of partitions than devices connected per gateway. This will make the parallelism greater than the shared channels with RabbitMQ but less than the exclusive channels.

4. TurtleBot Follower Application

In order to explore possible configurations for the IoTCloud framework, we have used the Microsoft Kinect[22] and TurtleBot[23]. The Microsoft Kinect consists of an IR camera, an RGB camera, an IR emitter, and several auxiliary features. Our project was not concerned with the details of the hardware but complete discussions of the Kinects specifications and

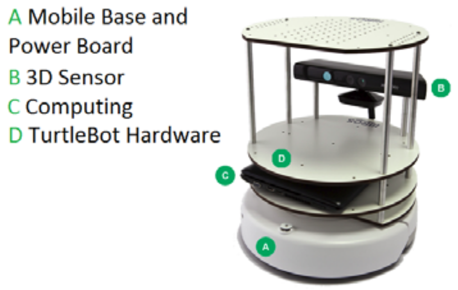


Figure 7 TurtleBot

method of depth calculation are available. Currently, there are numerous open-source projects and academic studies utilizing the Kinect, due to the sensors affordability and host of applications. In addition, a well-documented robot incorporating the Kinect is already available, the TurtleBot by Willow Garage. It is because of these many resources that the Kinect and TurtleBot were chosen as a subject for the development of a sensor to cloud processing framework.
In our application the TurtleBot follows a large target in front of it by trying to maintain a constant distance to the target. Compressed depth images of the Kinect camera are sent to the cloud and the processing topology calculates a point cloud of the TurtleBot's field of view. The algorithm uses the point cloud to calculate an average point, the centroid, of a hypothetical box in front of the TurtleBot. Shifts in the centroid

are calculated and command messages, in the form of vectors, are sent back to the Turtlebot using its ROS[24] API. The Turtlebot then actuates these vectors in order to maintain a set distance from the centroid.

4.1 Reading Depth Frames from the Kinect: The initial step in developing our application utilizing the Kinect depth camera was finding a driver to read in the Kinect data stream. The TurtleBot is operated with ROS, the open-source robotics operating system, which has an available Kinect driver. The ROS Kinect driver is built on OpenKinect's libfreenect[25] driver so in order to avoid any unnecessary overhead, libfreenect was used pure. Libfreenect is an open-source Kinect driver that provides a Java interface to both the IR and RGB cameras. Methods are provided to start a depth stream and handle frames. libfreenect was originally implemented in C++, although a Java JNA wrapper is now available.

4.2 Compression: In the course of the project several compression schemes were tested. In the early stages the LZ4, Snappy[26] and JZlib Java compression libraries were tested. Snappy achieved less compression but was faster than the other two. Ultimately, we chose a two-stage compression process using Mehrotra et al's [27] inversion technique as the first stage and Snappy as the second. Mehrotra et al's[27] inversion technique takes advantage of the error endemic to the depth camera. The depth camera's accuracy decreases proportional to the inverse of the squared depth. Hence, multiple values may be encoded to the same number without any loss in fidelity[27]. From using this inversion technique every two-byte disparity can be compressed to one byte. It is worth noting however that the inversion algorithm takes distance as an input, not disparity. Mehrotra et al. achieve a startling 5ms compression time for their whole 3-step process with little optimization. For the sake of expediency, our project used an existing java compression library (Snappy) rather than Mehrotra et al's RLE/Golomb-Rice compression.

The last decision left was whether to implement the prediction strategy mentioned in Mehrotra et al. The prediction strategy takes advantage of the heterogeneous nature of the depths of objects. This translates into long runs of values in the depth data. The prediction strategy is simple and converts any run into a run of 0's. For an RLE this will have a clear advantage but when tested with Snappy the gain was negligible and thus not worth the added computation. Ultimately, we were able to achieve a compression ratio of 10:1 in a time of 10ms. This compares favorably to Mehrotra et al's 7:1 ratio in 5ms. The data compression happens in the Laptop computer inside the Turtlebot. After the compression the data is sent to a driver application that runs in an IoTCloud gateway. This Gateway relays the information to the cloud.

4.3 Calculation of Velocity: The Storm topology for our scenario consists of 3 processing units. One spout receives the data, a bolt un-compresses this data and calculate the velocity vector required by the TurtleBot to move and last bolt send these vectors to the TurtleBot.

All the literature indicates that the Kinect should stream each frame as 307,200 11-bit disparity values, 2047 being sent to indicate an unreadable point. But upon inspection of received disparity values, the highest value observed was 1024. When this value was treated as the unreadable flag, the depth map displayed appeared normal. Depth shadows were rendered correctly along with the minimum and maximum readable distances. The code was then adjusted to expect only 10-bit disparity values and everything functions normally. The full range of the Kinect, 80 cm – 400 cm can be encoded with only 10-bit values. It is unclear whether the 10-bit values are a result of the Java libfreenect wrapper or faulty code, but our programs are fully functional and the issue was left unresolved. An explanation of this phenomenon would no doubt prove beneficial and may be a point of latter investigation.

The processing bolt creates a point cloud using the depth frames it receives using an approximation technique mentioned in [28]. The algorithm defines a hypothetical box in the TurtleBot field of view. The average point of this box is calculated and a velocity vector is generated for TurtleBot to move towards or away from this average point. This way TurtleBot always tries to keep a fixed distance to an object in front of it.

4.4 Controlling the TurtleBot: The driver running in the Gateway receives the velocity vectors from the processing application. It then converts this vectors to a format that the ROS API of the TurtleBot accepts. Ultimately the ROS API is used by the driver to control the TurtleBot. We use a Java version of ROS available for interfacing with ROS, which is primarily written in Python.

5. Results & Discussion

We primarily focused on the latency of the system and the scalability of the system. A series of experiments were conducted to measure the latency and how well the system performs under deployment of multiple of sensors. We used FutureGrid as our cloud platform and deployed the setup on FutureGrid OpenStack medium flavors. An instance of medium flavor has 2 VCPUs, 4GB of memory and 40 GB of hard disk. We run Storm Nimbus & ZooKeeper on 1 node, Gateways Servers on 2 nodes, Storm Supervisors on 3 nodes and Brokers on 2 nodes. Altogether our setup contained 8 Virtual Machines with moderate configurations.

In order to test the latency of the system we deployed 4 driver applications on the two Gateways that produce data at a constant rate. This data were relayed through the two brokers and injected to a Storm topology. Storm topology passes the data back to the Gateways and it was running 4 spout instances in parallel to get the data and 4 bolts in parallel to send the data out. The round-trip latency was measured at the gateways for each message. This setup was repeated for different message sizes and message rates. We went up to 100 message per second and increased the messages size up to 1MB. Each driver sent 200 messages and we got the average across all the drivers. We tested the system with RabbitMQ and Kafka brokers. For measuring the scalability we progressively increased the number of drivers deployed in the gateways and observed how many devices can be handled by the system.

The TurtleBot application is an application deployed on the FutureGrid. We observe TurtleBot following a human in front of it when this application was deployed. We tested the TurtleBot application through the Indiana University computer network and measured the latency observed.

5.1 Latency: Figure 6 shows the latency observed when running the tests through a RabbitMQ server. Up to 200KB messages, the latency was at a considerably lower value for all the message rates we tested. At 300KB messages the latency started to grow rapidly after message rate of 50.

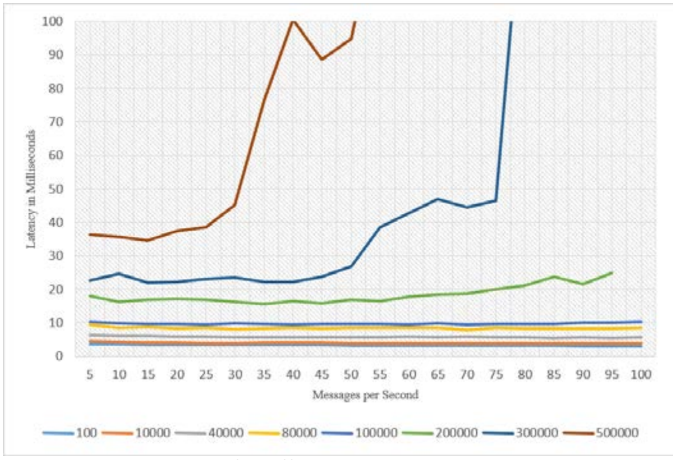


Figure 6 Average Latency for different message sizes with RabbitMQ. The different lines are for different message sizes in bytes.

Figure 7 shows the average latency observed with the Kafka broker. We observed some drastically high latency values frequently. The frequency of these values increased the average latency considerably. Despite variations in latency, in average the system was running with a considerably low latency with Kafka. The Kafka broker is better suited to be run in machines with high disk IO rates. We ran our tests on computation nodes that doesn't have very good IO performance. But there are other performance results of Kafka that were done on high disk IO nodes that shows some large variations in latency as well. In our setup Kafka broker latency was started to increase much more quickly than the RabbitMQ brokers.

5.2 Jitter: For most real time applications uniformity of the latency over time is very important. Figure 8 shows the latency variation in observed latencies for a particular message size and rate with RabbitMQ broker. The variation in latency was also minimal for message sizes up to 200KB. After that there was a large variation in the latency. The Kafka latency variation is very high compared to the RabbitMQ broker and we are not including those results here.

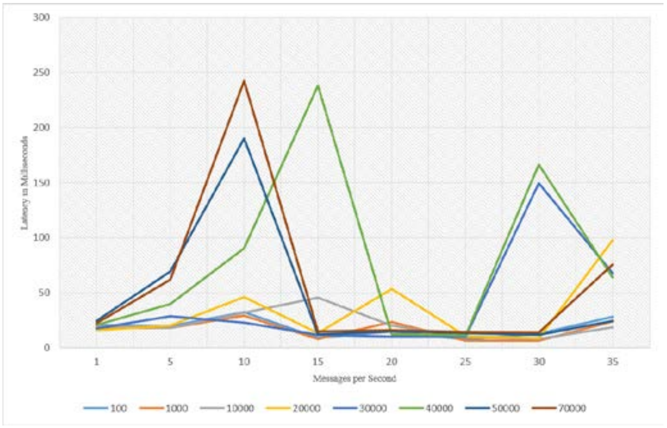


Figure 7 Average Latency for different message sizes with Kafka. The different lines are for different message sizes in bytes.

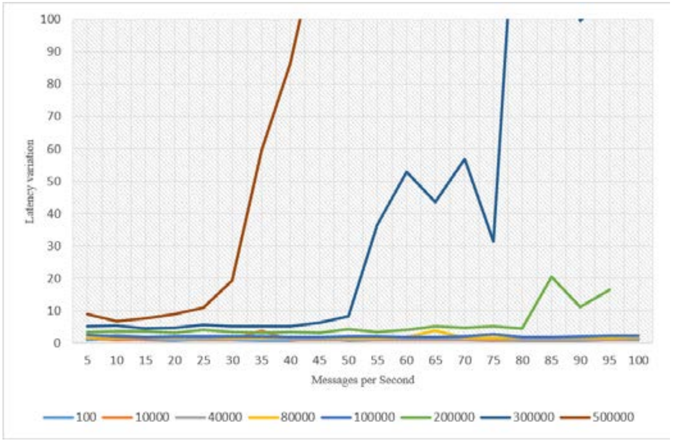


Figure 8 Latency standard deviation with different message sizes and message rates for RabbitMQ. The different lines are for different message sizes in bytes.

5.3 Scalability: In the test we did for observing the scalability of the system we deployed 1000 mock drivers in two gateways and measured the latency. These drivers can generate 100byte messages at a rate of 5 message per second. We use low values for both message rate and size so that we can make sure the system doesn't slow down due to large amount of data produced. Figure 9 shows the latency with RabbitMQ. Latency observed was little higher than the previous test we did with 4 drivers but it was consistent up to 1000 drivers and stayed within reasonable range. The increase in latency can be attributed to increased use of resources. At 1000 sensors the latency started to increase. Because this test was done in shared channel mode, only 2 spouts were actively reading from the 2 queues created.

We did the same test with the Kafka broker. Because we partitioned each topic in to 4, all 4 spouts were actively reading from the Topics. This is the advantage of having a Kafka like distributed broker. The latency observed is shown in Figure 10. As expected, there were big variations in the latencies observed. We tried to remove these big numbers and draw the graph to see how they affect the average latency. Figure 10 shows graphs with values > 200 removed. We can observe that the average latency is at a considerable low range after these very high values are removed. Kafka is a relatively new broker under development and we believe its development community is working on fixing these issues with the broker and expect these variations to reduce in future versions.

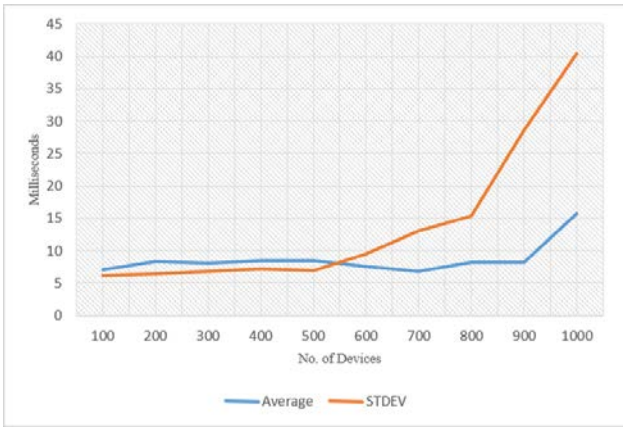


Figure 9 Latency with varying number of devices - RabbitMQ

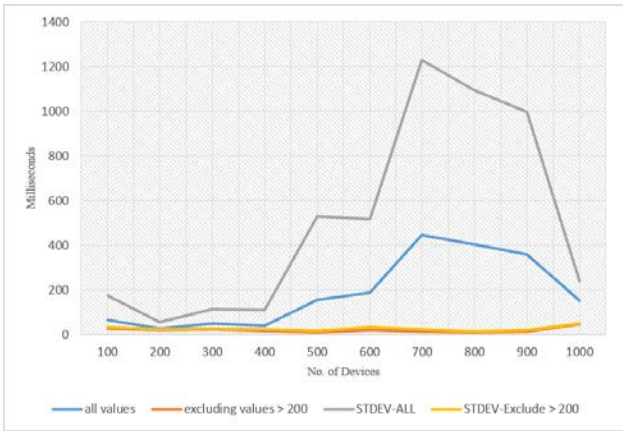


Figure 10 Latency with varying number of devices – Kafka

All the tests were done for the best case scenario in terms of latency of Storm based analysis. A real application would involve much more complex processing and a complex DAG structure for data processing. Those processing latencies will add to the overall latency in real applications. Also in our tests we sent and received the same message through the cloud. In real applications messages generated after the processing is usually minimal compared to the data messages. So we expect a reduction in latency as well.

5.3 TurtleBot: Because of the latency requirements, we used the RabbitMQ broker for the TurtleBot application. The TurleBot was functioning properly under the latencies we have observed. Figure 11 shows the latency values we observed for 1500 Kinect frames. The average latency fluctuated between 35ms and 25ms. The TurtleBot was sending messages of size 60KB in a 20 message/sec rate. The best case latency without any processing for such messages is around 10ms. The network latency and the processing is adding around another 25ms to the latency. The processing includes both compression and decompression time of Kinect frames. There were some outliers that went to values such as 50ms. These were not frequent but can be seen occurring with some high probability. We could not recognize any patterns in such high latency observations and some of the reasons for these increases can be network congestions, Java garbage collections and other users using the same network and resources in FutureGrid. We observed, average latency of 33.26 milliseconds and standard deviation of 2.91.

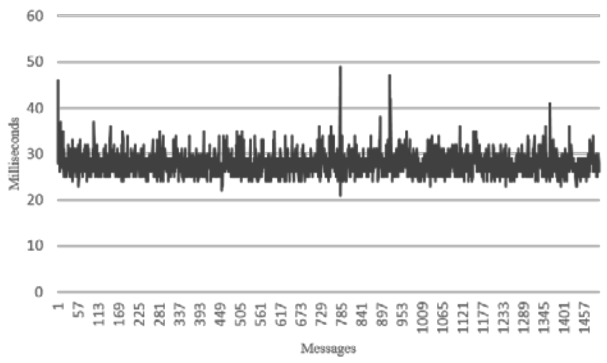


Figure 11 Latency observed in Turtlebot application

6. Conclusions

In this paper we introduced a scalable, distributed architecture for connecting devices to cloud services and processing data in real time. Further we discussed about a robotics application built on top of this framework. We investigated how to scale the system with topic based publish-subscribe messaging brokers and a distributed stream processing engine in the cloud. We measured the performance characteristics of the system and showed that we can achieve low latencies with moderate hardware in the cloud. Also the results indicate we can scale the architecture to hundreds of connected devices. Because of the low latencies, framework with the RabbitMQ broker is suitable for applications with real time requirements. Applications involving massive amount of devices without strict latency requirements can benefit from the scalability of Kafka brokers. The results also indicate that reasonably uniform behavior in message processing latencies can be maintained which is important factor for modeling most problems.

7. Future Work

As our platform evolves, we would like to extend our system to Cloud DIKW applications which involve both real time analysis and batch analysis. A primary concern for real time applications is the recovery from faults. A robot guided by a cloud application should work amidst the application level failures and middleware level failures. We would like to explore different fault tolerant techniques for making our platform more robust. The discovery of devices is coarse grained at the moment and we would like to enable finer grained discovery of devices at the cloud processing layer. For example selecting devices that meet specific criteria like geographical locations for processing is important for some applications. We observed that there are variations in the latency observed in our applications. In some applications it is required to contain the processing latency with hard limits. It will be interesting to look at methods for enabling such guarantees for our applications. Simultaneously we are working to build new robotics applications based on our platform.

8. Acknowledgement

The authors would like to thank the Indiana University FutureGrid team for their support in setting up the system in FutureGrid NSF award OCI-0910812. This work was partially supported by AFOSR award FA9550-13-1-0225 “Cloud-Based Perception and Control of Sensor Nets and Robot Swarms”.

References

1. Armbrust, M., et al., *A view of cloud computing.* Communications of the ACM, 2010. **53**(4): p. 50-58.
2. Eugster, P.T., et al., *The many faces of publish/subscribe.* ACM Computing Surveys (CSUR), 2003. **35**(2): p. 114-131.
3. Abadi, D.J., et al. *The Design of the Borealis Stream Processing Engine*. in *CIDR*. 2005.
4. Gedik, B., et al. *SPADE: the system s declarative stream processing engine*. in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008. ACM.
5. Neumeyer, L., et al. *S4: Distributed stream computing platform*. in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. 2010. IEEE.
6. Anderson, Q., *Storm Real-time Processing Cookbook.* 2013: Packt Publishing Ltd.
7. Hassan, M.M., B. Song, and E.-N. Huh. *A framework of sensor-cloud integration opportunities and challenges*. in *Proceedings of the 3rd international conference on Ubiquitous information management and communication*. 2009. ACM.
8. Souto, E., et al., *Mires: a publish/subscribe middleware for sensor networks.* Personal and Ubiquitous Computing, 2006. **10**(1): p. 37-44.
9. Krishnamurthy, S. *TinySIP: Providing seamless access to sensor-based services*. in *Mobile and Ubiquitous Systems-Workshops, 2006. 3rd Annual International Conference on*. 2006. IEEE.
10. Hall, C.P., A. Carzaniga, and A.L. Wolf, *DV/DRP: A content-based networking protocol for sensor networks.* 2006, Technical Report 2006/04, Faculty of Informatics, University of Lugano.
11. Hunkeler, U., H.L. Truong, and A. Stanford-Clark. *MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks*. in *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*. 2008. IEEE.
12. Dash, S.K., et al., *Sensor-cloud: assimilation of wireless sensor network and the cloud*, in *Advances in Computer Science and Information Technology. Networks and Communications*. 2012, Springer. p. 455-464.
13. Alamri, A., et al., *A survey on sensor-cloud: architecture, applications, and approaches.* International Journal of Distributed Sensor Networks, 2013. **2013**.
14. Videla, A. and J.J. Williams, *RabbitMQ in action*. 2012: Manning.
15. Kreps, J., N. Narkhede, and J. Rao. *Kafka: A distributed messaging system for log processing*. in *Proceedings of the NetDB*. 2011.
16. Hunt, P., et al. *ZooKeeper: Wait-free Coordination for Internet-scale Systems*. in *USENIX Annual Technical Conference*. 2010.
17. Snyder, B., D. Bosnanac, and R. Davies, *ActiveMQ in action*. 2011: Manning.
18. Goodhope, K., et al., *Building LinkedIn's Real-time Activity Data Pipeline.* IEEE Data Eng. Bull., 2012. **35**(2): p. 33-45.
19. Vinoski, S., *Advanced message queuing protocol.* IEEE Internet Computing, 2006. **10**(6): p. 87-89.
20. Agarwal, A., M. Slee, and M. Kwiatkowski, *Thrift: Scalable cross-language services implementation*. 2007, Tech. rep., Facebook (4 2007), http://thrift. apache. org/static/files/thrift-20070401. pdf.
21. Fox, G., et al., *FutureGrid—A reconfigurable testbed for Cloud, HPC and Grid Computing.* Contemporary High Performance Computing: From Petascale toward Exascale, Computational Science. Chapman and Hall/CRC, 2013.
22. Zhang, Z., *Microsoft kinect sensor and its effect.* MultiMedia, IEEE, 2012. **19**(2): p. 4-10.
23. Garage, W., *TurtleBot.* Website: http://turtlebot. com/last visited, 2011: p. 11-25.
24. Quigley, M., et al. *ROS: an open-source Robot Operating System*. in *ICRA workshop on open source software*. 2009.
25. openkinect. *Open Kinect*. 2014 [cited 2014; Available from: http://openkinect.org/.
26. Google. *snappy*. 2014 [cited 2014; Available from: https://code.google.com/p/snappy/.
27. Mehrotra, S., et al. *Low-complexity, near-lossless coding of depth maps from kinect-like depth cameras*. in *Multimedia Signal Processing (MMSP), 2011 IEEE 13th International Workshop on*. 2011. IEEE.
28. openkinect. *Imaging Information*. 2014; Available from: http://openkinect.org/wiki/Imaging_Information.