Cloud based real-time multi-robot collision avoidance for swarm robotics

Hengjing He
State Key Lab. of Power System, Dept. of Electrical Engineering, Tsinghua University
Beijing, China
hehj11@mails.tsinghua.edu.cn

Supun Kamburugamuve, Geoffrey C. Fox
School of Informatics and Computing and CGL
Indiana University
Bloomington, USA
{skamburu, gcf}@indiana.edu

Abstract: The IoT Cloud is a cloud platform for implementing applications that remotely control smart devices or process a huge amount of real time stream data from a massive number of physical devices. Such platforms shift computation load from the device to the cloud and provide powerful processing capabilities to a simple device. In Swarm robotics, robots are supposed to be small, energy efficient and low-cost, but still smart enough to carry out individual and swarm intelligence. These two goals are normally contradictory to each other. Besides, in real world robot control, real time on-line data processing is required, but most of the current Cloud Robotic Systems are focusing on off-line batch processing. However, the IoT Cloud may provide a way that leads this research area out of its dilemma. This paper explores the availability of IoT Cloud for real time control of massive complex robots by implementing a relatively complicated but better performed local collision avoidance algorithm on the platform. The IoT Cloud application and the IoT Cloud Driver, which connects the robot and the Cloud, are developed and deployed in the IoT Cloud. Simulation tests are carried out and the results show that, when the number of robots increases, by simply scaling the computation resources for the application, the algorithm can still maintain the preset control frequency. Such characteristics verify that the IoT Cloud is a new platform for studying massive complex robots in swarm robotics.

Key word: Internet of things, cloud computing, swarm robotics, swarm intelligence, collision avoidance, real time stream processing

I. INTRODUCTION

Since Cloud computing can provide elastic, on demand, ubiquitous worldwide accessible computing and storage resources, it has been introduced into various areas from big data analysis to real time robot control. One very promising area is developing a universal platform for Internet of Things(IoT)[1] applications using cloud computing technology. The IoT Cloud system is normally featured as both real time responding and big data processing. As a large number of smart devices are connected to the cloud, massive real time stream data from these devices needs to be analyzed and processed before it can be recorded in the database and processed offline by the cloud. In some scenarios, such as robot control, the stream data from devices has to be processed and fed back in real time. These time-critical tasks require the system to respond fast enough, thus a batch analytics technique such as MapReduce is not viable for this kind of application.

However parallel processing ability and cluster computing framework techniques like MapReduce are very appealing, especially to systems that need to deal with a large number of computation intensive entities. Swarm robotics[2] is a typical research area that commonly deals with such systems. In swarm robotics, normally, the robot should be as small and energy efficient as possible[3, 4], but still it needs to be able to perform the basic behaviors of an intelligent entity under research and can also carry out high level swarm intelligence[5]. For a traditional robotic system, these two aspects are mutually exclusive. But with the help of cloud computing, most of the computation can be offloaded to the cloud and, by utilizing elastic cloud computing, the number of robots in a swarm can scale flexibly. As most of the computation is transferred into the cloud, the onboard system of a robot can be greatly reduced, keeping only sensors, communication and actuation modules and leaving all high level algorithms to the cloud. Motivated by such demand, several cloud platforms[6-10] dedicated to robotic control have been designed. Nevertheless, most of these systems mainly focus on static data processing, such as object recognition, path planning and so on. These tasks are not strictly time-critical as such dynamic tasks as local collision avoidance[11].

The IoT Cloud platform, developed on a real time distributed processing framework, is a scalable real time stream data processing system[1]. This platform is much more suitable for time-critical applications as it processes stream data for real time response. The core of the IoT cloud is a distributed real-time stream computing engine. Data from devices or databases can be injected into the engine as streams and the computing logic running in the engine will continuously process the data and then emit results out. The computing engine utilizes cluster computing paradigm, which makes it easy to scale and also fault-tolerant.

This paper explores the parallelism and scalability of the IoT Cloud platform in real time data processing by implementing multi-robot collision avoidance. Unlike other parallel algorithm research, this paper focuses on entity or agent level parallelization and studies mainly computation resource scaling according to the computation load. And unlike normal swarm robotic researches that seek for simplified models to reduce computation, this paper implements a complicated algorithm that reflects in-depth details about the physical system and can be used in a real world scenario. The results of the experiment demonstrate that the IoT Cloud introduced in this paper is an effective,

scalable platform for swarm robotics. The main contribution of our research is exploring novel cloud frameworks for implementation of computation intensive algorithms in swarm robotics.

The remainder of this paper is organized as follows: Section II briefly introduces collision avoidance theory and related algorithms. Section III describes the architecture of the IoT Cloud Platform. Section IV explains the design of the cloud application in detail. Section V presents our experiment over the whole system and the application and analyzes the results. In the end, Section VI summarizes and concludes the whole paper.

II. LOCAL COLLISION AVOIDANCE FOR NON-HOLONOMIC ROBOTS

Local collision avoidance is one of the most important aspects in robot navigation. The task of local collision avoidance is to dynamically compute the optimal collision free velocity for a robot, which is based on the observation of the environment. Unlike motion and path planning that have static knowledge of the global environment and make one-time decisions, local collision avoidance needs to respond to the dynamics of the environment[11] such as other active entities or obstacles that are not presented in the static map.

Current local collision avoidance methods are mainly based on the Velocity Obstacle (VO) theory[12]. VOs are areas in velocity space where if the velocity of a robot points into one of the areas it will collide with another robot after some time. A diagram of VO is shown in Fig. 1. Several types of VO are defined according to the different VO calculation methods. Reciprocal Velocity Obstacle (RVO)[13] splits the collision avoidance responsibility equally between the two robots that may collide with each other, while the Hybrid RVO(HRVO)[14] translates apex of RVO to the intersection of the RVO leg closest to its own velocity and the leg of VO furthest from its own velocity, which encourages choosing preferred sides and reduces the chance of a reciprocal dance.

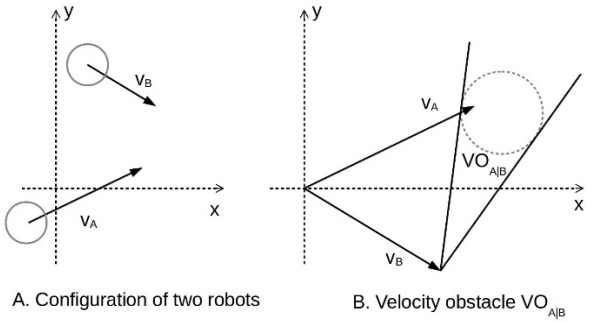


Fig. 1 Velocity obstacle introduced by robot B

All these methods assume that the robot can reach any velocity in the velocity space, one hundred percent accurate localization information and circular robot footprint. However, the real robot cannot satisfy such prerequisites. Therefore other constraints need to be attached to those VOs. These include kinematic constraints such as acceleration and max velocity limits, Non-Holonomic constraints[15] for differential robots, and localization uncertainty[16]. When considering localization uncertainty, a robot footprint needs to be expanded so that it can cover the uncertainty from localization and make sure that the calculated velocity is valid even if localization is not accurate. Simply using a circular footprint with an extended radius may exclude possible valid velocities. So a convex hull footprint[17] calculated from the Minkowski Sum of robots and obstacles is introduced in calculating VOs. The calculation of convex hull footprint for a robot is highly computation intensive and may take around 50 percent of the total computation time.

Once all VOs are obtained from velocity space, an optimization algorithm needs to be designed to select optimal velocity from areas outside all VOs. There are three key methods for collision free velocity selection. They are Optimal Reciprocal Collision Avoidance (ORCA) method, Clear Path method[18] and Sampling based method. According to [17], Clear Path method has relatively better overall performance in real world experiments.

Taking into account all the detailed considerations above, an algorithm is developed[17] that can control robots in the real world to avoid each other in a more effective way. But such an algorithm requires at least a laptop to run. In swarm robotics the number of robots can reach about one hundred or more, and, obviously, equipping a laptop for each robot can greatly increase investment and also the size of the robot, besides which, power consumption of a laptop will lead to less robot running time.

To utilize the algorithm but at the same time, reduce the "side effects", one effective approach is offloading algorithm computation into a cloud environment and connecting the robot through a wireless network[19]. In this paper, an algorithm, is implemented which uses a convex hull footprint for VO calculating, considers all aforementioned constraints, and utilizes Clear Path method for optimal velocity computation. The following sections offer details about the cloud platform and illustrate an implementation of the algorithm.

III. IOT CLOUD ARCHITECTURE

The IoT Cloud[1] is a platform that provides cloud services for a large number of Internet-accessible devices. The IoT Cloud mainly consists of three layers: Front-end Gateway Layer, Stream Processing Middle Layer and Batch/Storage Back-end Layer. The three layers are connected via message broker and coordinated by Zookeeper. Fig. 2 depicts all major components of the system.

The Front-end Gateway Layer is responsible for connecting devices with the Message broker. As IoT Cloud is designed to serve heterogeneous devices, it needs a component to record specific information about the devices and map between message broker channels and native device data channels. Such a component is the Gateway. All devices are connected through the Gateways, and below the

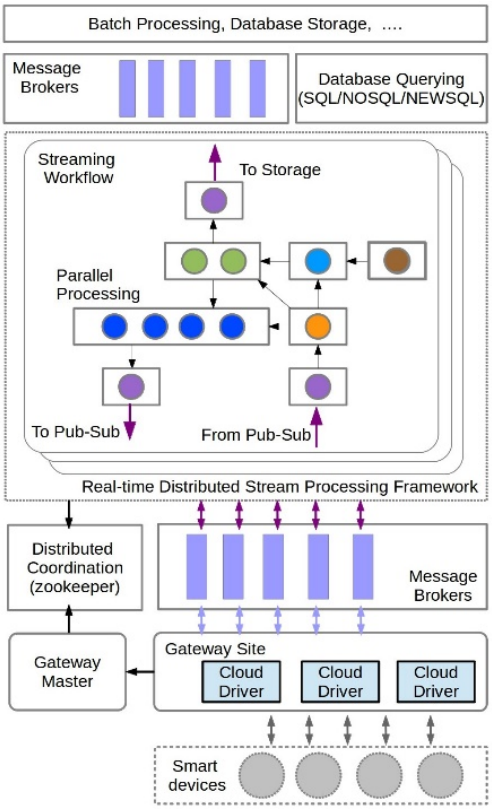


Fig. 2 The architecture of the IoT Cloud

Gateways are IoT Cloud device drivers which convert device data into messages that cloud services can process. The cloud device drivers first get data from devices and then call IoT Cloud APIs to send converted data into the cloud. The Gateway maintains connections between devices and the cloud. At the same time there is a Gateway master that coordinates multiple Gateways and registers connection information, such as channel mapping between message broker and devices, so that the Middle layer can discover devices and provide service entries.

The Stream Processing layer handles real-time data processing. This layer uses Apache Storm[20] as the computation engine. Storm is a distributed real-time streaming processing system that can process at most one million Tuples (data type processed in Storm) per second. It is very suitable for processing stream data from numerous smart devices. Storm gets source data from one of its components called Spout and then sends data to a process component called a Bolt. Spouts and Bolts are arranged as nodes in a graph that are connected by streams which resemble the edges of a graph. Such a stream processing workflow is called Topology. To use the IoT Cloud service, application Topology should be developed first. Since data input and output of the application Topology are closely related to devices, the IoT Cloud platform provides APIs to build custom input Spouts and output Bolts. As mentioned before, the Gateway layer is responsible for maintaining the connections of Spouts and Bolts to the message broker by writing connection information to Zookeeper[21]. To use the real time stream processing service in this layer, data from devices should be sent to the correct message broker channels, which are connected to the input Spouts of an application via IoT Cloud device drivers. Then by subscribing the channels that connect to the output Bolts of the application Topology, results can be fetched in real time. Such a data processing paradigm is very suitable for robot controlling. Most of the work in this paper focuses on designing and implementing application Topology and its corresponding IoT Cloud driver for robot collision avoidance. The bulk of the computation required for the collision avoidance algorithm is shifted to this layer. Once an application is deployed into the IoT Cloud, it can provide services to a large number of devices as long as they have the correct IoT Cloud drivers. By deploying multiple instances of the application or increasing the number of computation nodes for the application, data processing ability can be scaled accordingly.

The Batch/Storage Layer stores data from Stream Processing Middle Layer and provides Batch Processing/Data Mining services for the static data from various distributed databases. Since this paper mainly works on real time data processing this layer will not be used.

IV. IMPLEMENTATION OF THE COLLISION AVOIDANCE ALGORITHM

A. Application overview

Fig. 3 shows the overall design of the collision avoidance application. In the front-end Gateway layer, there is an IoT Cloud driver module which communicates with devices and converts data between message broker and local device. The driver is deployed on the Gateway site of the IoT Cloud System, which runs on a local desktop machine and is managed by the Gateway. As most robots run Robot Operation System(ROS)[22], here ROS is adopted as the device driver that interacts directly with the robots. The IoT Cloud driver will subscribe ROS topics to get the robot state, such as odometry, laser scans and so on, and then convert the data into messages that can be transmitted through a broker to the cloud. After finishing the data processing, the IoT Cloud will send back velocity commands through the message broker. The IoT Cloud driver will convert the

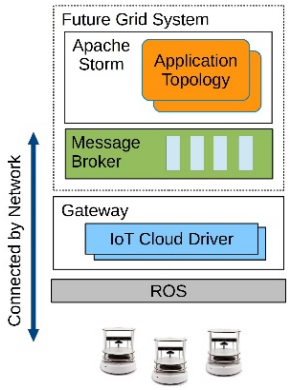


Fig. 3 Overview of the Collision Avoidance application

message into a ROS message, which is then sent to the correct ROS topic so that the robot can be controlled. This paper uses ROS as a device driver just for demonstration. Besides ROS, other device specific drivers can also be used, as long as they can provide APIs for data retrieval.

The IoT Cloud computation engine, together with the message broker servers, is deployed in the FutureGrid[1] cloud platform. The complicated collision avoidance algorithm is implemented as a Storm Topology running in the computation engine. The message broker of the IoT Cloud relays data from IoT Cloud driver and feeds it into Spouts of the control Topology. When the velocity command is calculated, the Topology will send the command through an output Bolt to the message broker, which then relays the message back to the cloud driver, and then to the robot.

B. IoT Cloud driver for collision avoidance

IoT Cloud driver is used to connect devices to the IoT Cloud Platform. For different types of devices, the Cloud drivers are different, but for the same type of device they can use the same Cloud driver and only need to spawn a new driver instance for each device. To perform collision avoidance, odometry, laser scan and pose array of the robot need to be sent to the cloud. All the information is published by the robot through ROS topics as shown in Fig. 4. So the IoT Cloud driver first subscribes these topics and gets the ROS messages. However these ROS messages are not viable for message brokers, such as Rabbitmq which is used in this work, and it is the cloud driver that converts these messages into custom defined data types that can be processed by the message broker and the Topology.

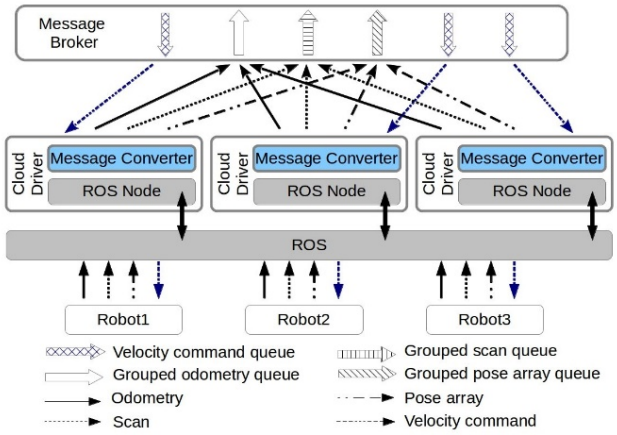


Fig. 4 IoT Cloud driver for collision avoidance

The next thing that an IoT Cloud driver needs to do is define IoT Cloud Channels for those ROS topics. For IoT Cloud application Topology, each input Spout or output Bolt is connected to a predefined IoT Cloud Channel according to the application and the message broker. All data is transmitted through these Channels. While the number of Spouts and Bolts in an application topology cannot be changed, the number of the robot that connects to the cloud may vary from time to time, so IoT Cloud Channels should be defined according to the robot information types rather than the robot entity. Thus the IoT Cloud driver will create an IoT Cloud Channel for each information type and publish converted messages to the corresponding Cloud Channel. To distinguish the messages sent from different robots, a unique robot ID generated by the Cloud Driver is attached to the message. The application Topology will get the correct robot state according to the ID. However, the Bolt of the application topology that publishes velocity commands back to robots will also publish all commands for different robots into one IoT Cloud Channel. It is the Gateway that creates a command queue for each cloud driver instance, and sends each message to the correct queue according to the robot ID attached in the message.

C. Topology design

All the algorithm and control logic for IoT Cloud-based collision avoidance are implemented in the Storm Topology. Before designing the application Topology, some details of the collision avoidance algorithm which is implemented in local mode should be explored. The collision avoidance algorithm introduced in section II is summarized in Fig. 5.

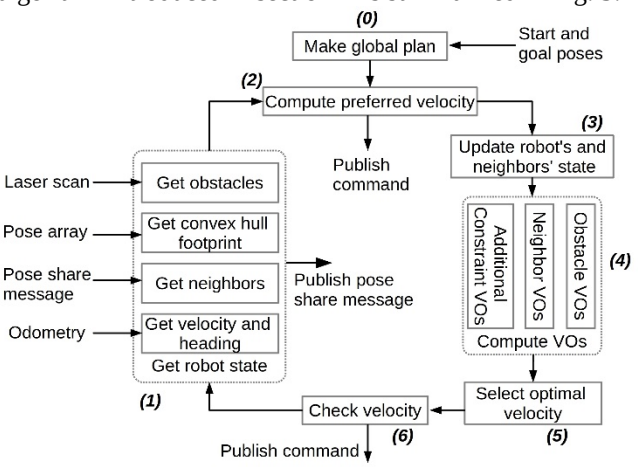


Fig. 5 The collision avoidance algorithm

The algorithm runs as a control loop which executes periodically. For a single loop, it starts by collecting the robot's newest state information, including getting obstacles from a laser scan, calculating convex hull footprint from pose array, getting neighbors from pose share messages and extracting velocity and pose of the robot from the odometry. The next step is to calculate preferred velocity from a global plan. This paper implements a very simple global planner that generates a straight path consisting of a number of way points from the start to the goal. If the robot has already reached the goal position and it only needs to adjust its heading, then a stopping velocity command or rotating command will be sent to the robot directly. Otherwise, the algorithm will update the robot's position and its neighbors' according to the predefined control period. With all the information up to date, VO lines from different aspects will be calculated. Such VO lines include those from neighbors, obstacles and various constraints, such as aforementioned

kinematic constraints, non-holonomic constraints and so on. Once all VO lines are obtained, the optimal velocity that is closest to our preferred velocity is selected using Clear Path algorithm. Finally the application will check the validation of the new velocity computed. If it is valid, then the velocity will be sent to the robot, otherwise the application will try the next way point and calculate a new velocity.

To implement the whole application into a Storm Topology, Spouts and Bolts that connect the Topology and the message broker should be designed first. As there are five types of information that need to be uploaded into the Topology, five Spouts need to be defined. These Spouts include odometry receiver Spout, scan receiver Spout, pose array receiver Spout, configuration Spout and pose share receiver Spout. The first three Spouts are used to get robot state information, while the configuration Spout receives basic parameters of the robot, such as control frequency, acceleration limits, maximum velocity, start pose and goal pose and so on, and the pose share receiver Spout is responsible for feeding information about all neighbors to the Topology. Two Bolts are required for publishing the computed velocity command and pose share messages to the message broker respectively. As the algorithm needs the neighbors' information, all robots in the scene should publish their state to a common IoT Cloud Channel periodically so that they can share their newest state with each other. All of these Spouts and Bolts are defined in a configuration file and the IoT Cloud platform will automatically generate them according to this file.

The rest components of the application in Fig. 5 can be implemented in different ways. For example, all of the rest components can be integrated into one Bolt, or each of the components can be implemented into a Bolt. Three possible Topologies are shown in Fig. 6.

All of the three Topologies are implemented in JAVA. Topology A integrates all components into one Bolt. As all messages are fed into the Bolt, the Bolt is so busy dealing with new messages that the overall delay of the velocity command is high. Topology C implements each component into separate Bolts and even calculates different types of VO lines in parallel. However robot state information has to be transmitted between several Bolts, resulting in the serialization/deserialization process along with the communication delay between computation nodes consuming much more time than the time that is saved by parallel computing. So the overall delay of Topology C is also very high. By reviewing the performance metrics of Topology A and C, it demonstrates Bolts that process input messages from Spouts require much less computation than the Bolts that calculate VO lines and velocity commands. So in Topology B, all components that process robot state information are combined into one Bolt and other components that calculate VO lines and velocities are wrapped into another Bolt. Such a design can reduce delays caused by data transmission between Bolts and, at the same time, isolate message processing from the main collision

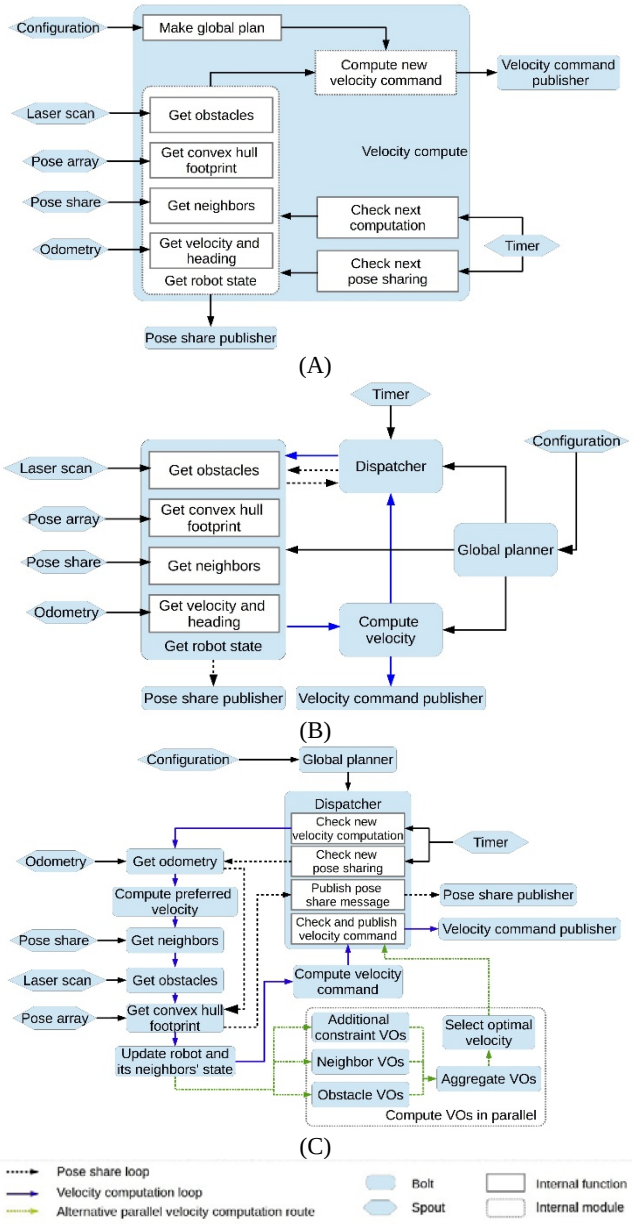


Fig. 6 Three possible topologies for collision avoidance application

avoidance algorithm.

Besides those Spouts and Bolts that interact with message broker, there are five more components in Topology B. To utilize collision avoidance control service, a robot should first send its parameters and start and goal poses to the Global Planner Bolt through its Configuration Spout. The Global Planner Bolt will then do the following jobs:

- Make a global path plan according to the start and goal Poses.
- Spawn a custom-defined JAVA Object called Agent that contains robot parameters, some algorithm related state variables, and the global plan generated before, and send it to Velocity Compute Bolt.

- Spawn a Control-Publish Time State Object that contains control period and pose share period, and also two variables to record the last time that the robot is controlled/published respectively. Besides, there are two Boolean variables that record whether the Topology is currently calculating velocity or publishing pose share message. This Object will be sent to the Dispatcher Bolt that triggers control or pose share processes according to the given period.
- Spawn a Pose Share Message Object that contains basic information to be shared. This object is sent to the Agent State Bolt for robot pose sharing.

Each of the three Objects spawned by the Global Planner Bolt will be stored as <robot Id, object> Hash Map in the destiny Bolt.

The Dispatcher Bolt will check those Control-Publish Time State Objects stored in the Bolt instance for controlling or pose sharing. Every 10ms it will receive a Tuple from the Timer Spout, which will trigger the Dispatcher Bolt to check whether it needs to emit a new Tuple to the Agent State Bolt to start a new controlling/publishing loop.

The Agent State Bolt implements modules that collect up-to-date robot information as shown in Fig. 5. If it gets a Tuple that tells it to calculate a new velocity command, then the Bolt will create a new Agent State Object and store all current robot state information in the Object then send it as a Tuple to the next Velocity Compute Bolt. Otherwise, if the Tuple asks it to share the robot information to others, the Bolt will fill a Pose Share Message Object with the current state information and send it to the Pose Share Publish Bolt for publishing the message to the message broker. After this Pose Share Message is published, Agent State Bolt also needs to send back a Tuple to the Dispatcher Bolt to tell it that the current job is done and a new Pose Share task for this robot can be accepted.

The Velocity Compute Bolt contains all other modules for velocity command calculation. After a new Agent State Object is received, this Bolt will select the correct Agent Object from the Hash Map that stores Agent Objects from Global Planner Bolt and then execute step 2 to step 6 in Fig. 5. If the calculated velocity command is valid, it will be sent to Velocity Command Publish Bolt to publish the command back to the cloud driver via a message broker. Just like Agent State Bolt, this Bolt has to send a Tuple back to the Dispatcher Bolt to tell it that the Topology is ready to receive the next calculation requirement for this robot.

As mentioned before, some of the Bolts may cache some runtime information about a robot. However each Bolt can run multiple instances in parallel, and how to make sure the proper Tuple is sent to the Bolt instance that caches the right robot information is very important to the process logic. In Apache Storm, the organization of connections between instances of different connected Bolts is called Grouping. Here, each Tuple, except the one emitted from Timer Spout, is attached with the robot id Field and Field Grouping based on the id Field is used. However, for Timer Spout, all Dispatcher Bolt instances need its periodical output as a time reference. Also the Agent State Bolt instances need to cache every robot's pose share information so that they can extract all neighbors for each robot. As such these two components use All Grouping.

The Topology shown in Fig. 6B can be used to control multiple robots and only needs them to send the required information and parameters to the Topology. From the Cloud Computing perspective, the IoT Cloud platform that runs this application Topology can provide robot collision avoidance control services to multiple robots. And with the ability to scale the platform, the application, or even a single Bolt in the Topology, such robot control framework can be used in Swarm Robotics that need to control a multitude of robots and at the same time retain the details of the robot model or the algorithm.

V. EXPERIMENT AND RESULTS

A. Application verification test

To verify the application developed in this paper, several experiments and tests were carried out.

As the algorithm implemented in this paper has already been tested in real world multi-robot collision avoidance, this paper will only use a software simulator to test the application. The simulator chosen is Simbad[23]. Simbad can simulate differential robot with laser scan range finder sensor. More importantly, Simbad is a light-weight simulator which is able to simulate more than 100 robots in one scene. Deployment of the whole experiment system is shown in Fig. 7.

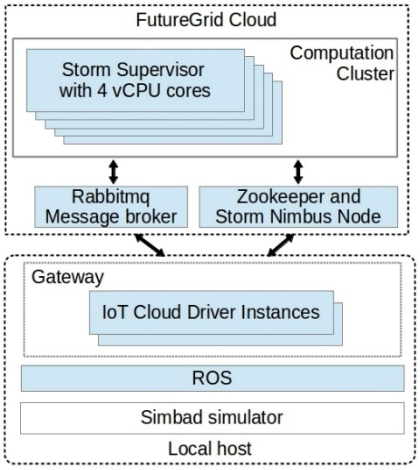


Fig. 7 Deployment of the application

Apache Storm and Rabbitmq Message broker are deployed in the FutureGrid Cloud Platform while the IoT Cloud Gateway is deployed with the Cloud Driver, ROS and the simulator being on a local desktop computer. The Simulator will publish the information of each robot to ROS and then IoT Cloud Driver will convert those ROS messages into custom-defined JAVA Objects that are used in the Topology. Configurations of the system are presented in Table 1.

Table 1 Hardware configuration of the system

	VMs in Cloud	Local host
CPU Model	Intel Core i7 9xx	Intel Core i7-2620M
CPU Frequency/Mhz	2933.436	2701
Cores	4	2
Thread per core	1	2
Memory/MB	8192	7900
OS	Ubuntu 12.04.4 (Linux 3.2.0)	Ubuntu 12.04.5 (Linux 3.13.0)
Hypervisor	KVM	None

The main task of this experiment is to test the availability of the application for large scale robot control. Since no SLAM (Simultaneous Localization and Mapping)[24] module is developed in Simbad and localization Pose Array cannot be generated, here Gaussian noise is added to the robot pose to create a fake localization pose array for the test. Pose array is published at a frequency of 10Hz and other information is published at a frequency of 20Hz.

Local test shows that the most computation-intensive component is the Velocity Compute Bolt, so velocity command delays for different number of robots with a different parallelism hint for Velocity Compute Bolt is measured. As shown in Table 1 there are 5 computation nodes with 20 cores in the cluster. To make sure each Bolt instance runs in parallel, the maximum number of parallel instances for Velocity Compute Bolt is limited to 5, while for Agent State Bolt it is set to 2 to see whether increased parallelization of the Agent State Bolt can bring better performance. Other components in the application Topology have only one instance for each. Also, to make sure the computation load is evenly distributed between the instances, the Filed Grouping strategy is replaced by custom defined Mod Grouping and an index value sequentially from 1 to maximum number of robots is assigned to each robot. This index is attached to all messages and Mod Grouping uses results of the index Mods the number of target Bolt instances to select which instance or task the message is sent to.

First off, NPC (Number of parallelism for Velocity Compute Bolt) is set to 5 and NPS (Number of parallelism for Get Robot State Bolt) is changed from 1 to 2. Testing the delays and collision times for NR (Number of robots) range from 5 to 50 to see how many robots the system can serve. Both the control frequency and robot pose share frequency are set to 20 Hertz, which means velocity command latency should be around 50 millisecond for the robots to avoid colliding with each other effectively. All robots are arranged on a circle with a radius of 6 meters and centered on the origin of the coordinate. These robots will go through the center to the antipodal position, then turn around and repeat the process. Each test runs for 300 seconds. Results are shown in Fig. 8(A1) and Fig. 8(B1).

Fig. 8(A1) and Fig. 8(B1) indicate that when the number of robots increases to 25, collisions will happen and the average velocity command delay increases to around 57 milliseconds. So for NPC less than 5, the maximum number of robots is set to 30. The test results are shown in Fig. 8(A2) to Fig. 8(B3).

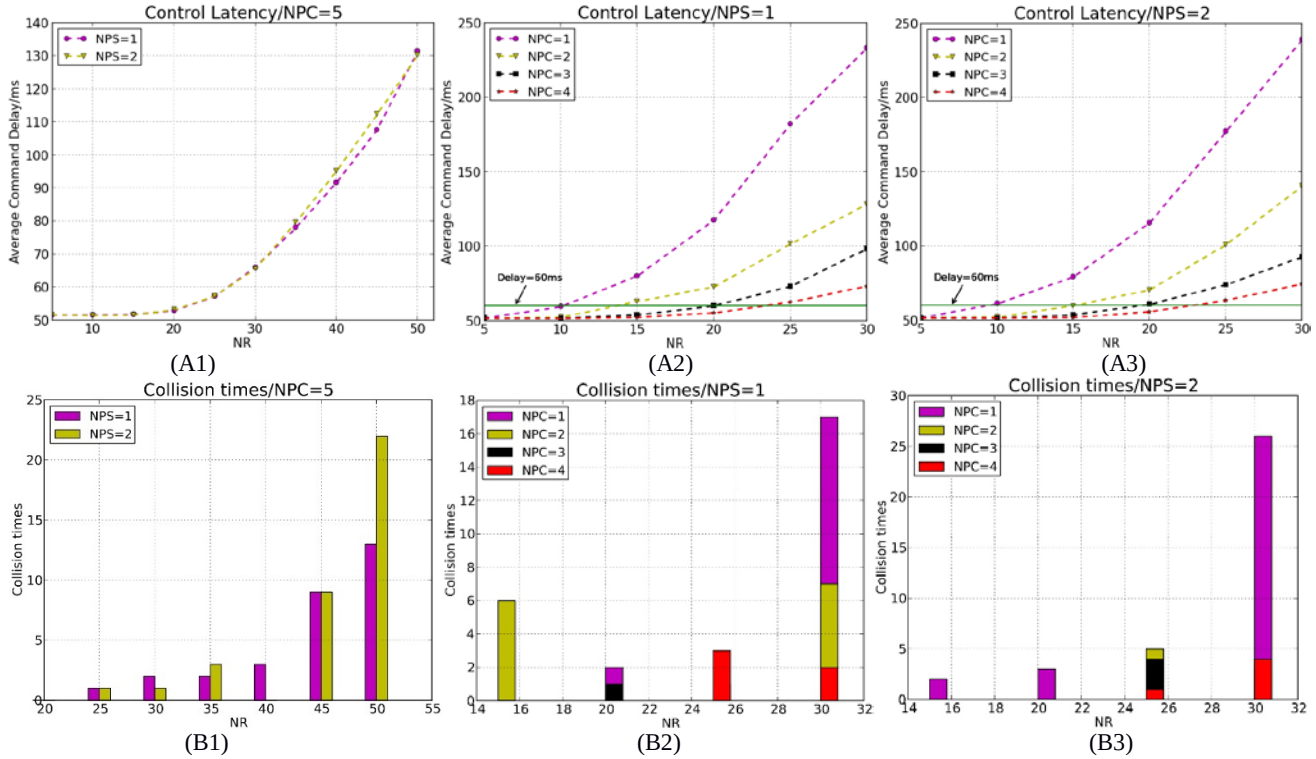


Fig. 8 Test results for different combinations of NPC (Number of parallelism for Velocity Compute Bolt) and NPS (Number of parallelism for Get Robot State Bolt). The first column of the figures shows command delays and collision times of robots with NPC=5, NPS varying from 1 to 2 and the maximum NR (Number of Robots) in the test set to 50. The rest of the figures show test results with NPC varying from 1 to 4, NPS varying from 1 to 2, and the maximum NR in the test to be 30.

Fig. 8(A2) to Fig. 8(B3) show that when the delay increases to around 60 milliseconds, collisions will occur. However in dense scenarios, collisions may still happen if the delay is less than, but still near to, 60 milliseconds. Also, increasing the parallelism of Agent State Bolt does not improve the performance. This is because computation load on the Agent State Bolt is very small (load capacity on this Bolt is less than 5%) and the overhead resulting from parallelization is almost comparable to the computation load on this Bolt. Thus increased NPS will generally bring no performance improvement in this test.

All results in Fig. 8 show that with the increase of parallelism for Velocity Compute Bolt, delay of the calculation for new commands decreases drastically, which proves that IoT Cloud can be used for real time robot control. Moreover, IoT Cloud-based applications can maintain good performance by simply scaling the computation resources in the Cloud when the number of robots increases. Such scaling ability with real time controlling provides a novel approach for Swarm Robotics or even Swarm Intelligence that includes more than just robotic area. Fig. 9 shows some snapshots of the simulation.

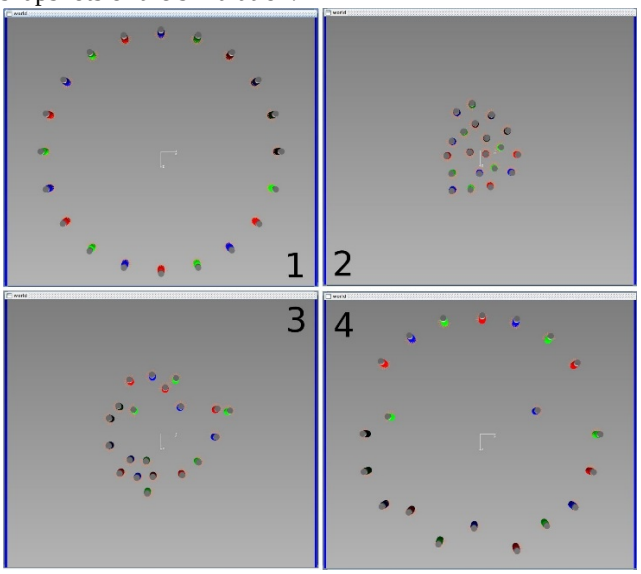


Fig. 9 Snapshots of the test

B. Performance test

Our previous test demonstrates that the application can control a swarm of robots to avoid collision. However the test cannot determine the overall performance of the application since robots are not set in a dense scenario. In this section robots are arranged very close to each other as shown in Fig. 10 to test the performance of the system.

Robots are placed in a square array all facing the origin of the coordinates and the antipodal position is set as the goal. If the number of the robots in a row/column is odd, all robots are shifted a little so that the start position and the goal position will not be the same for all robots. In addition, to keep the scenario dense all throughout the test, the control command is not executed by robots, so robots will not move during the test.

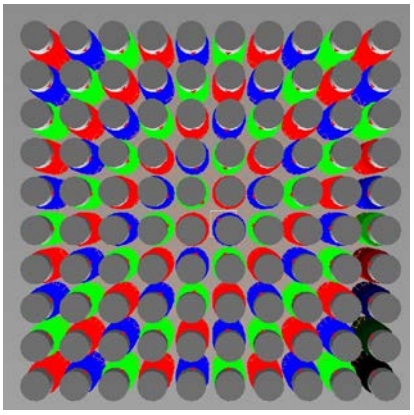


Fig. 10 Test scenario

Previous results show that except for Velocity Compute Bolt, other components in the Topology have load capacity less than 5%, so there is still much computation resources available in the cluster and increasing NPC to a reasonable number larger than 5 will not affect the test. Here NPC is set to 6, 8 and 10 with NPS varying from 1 to 2.

To get deep insight into the performance of key components in the application, detailed time consumptions in the control and pose share loops are measured. The timeline for each loop in the IoT Cloud is shown in Fig. 11. The pending process in Fig. 11 contains data transmitting between Bolts and queuing to wait for the next Bolt to become available. If the sum of all pending and computing time in a loop is less than the preset control/pose share period, there will be additional waiting time.

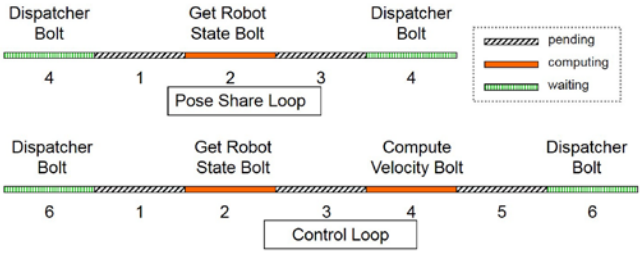
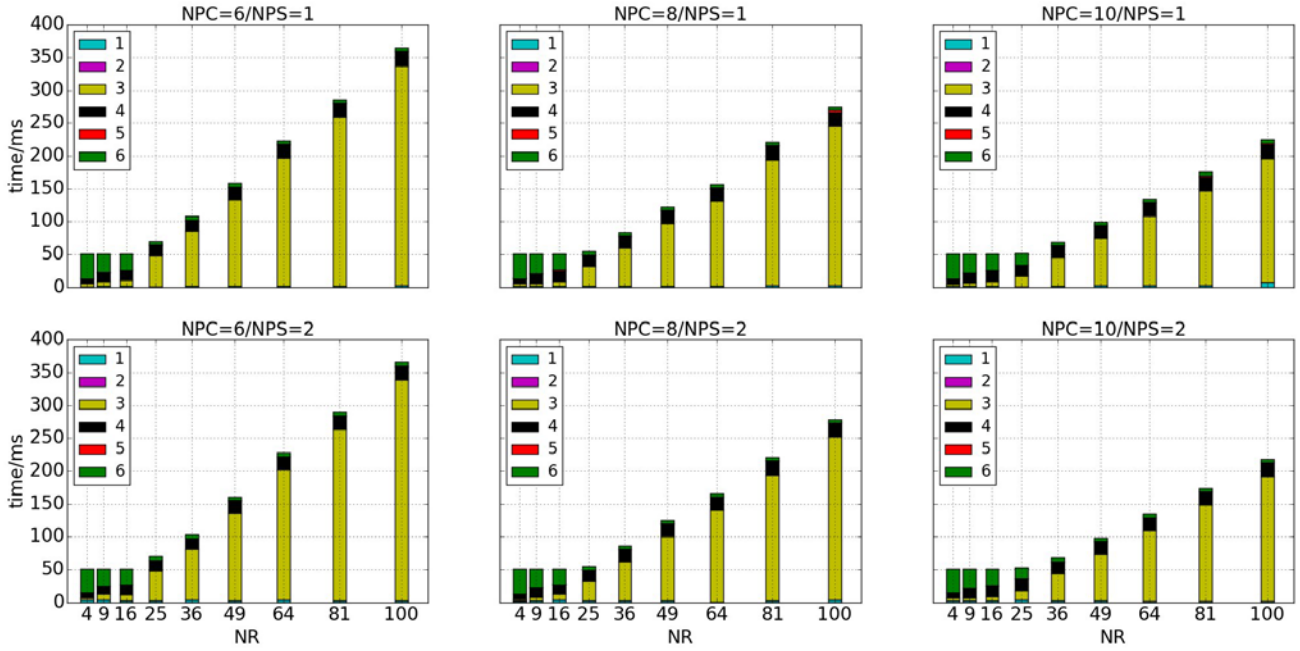


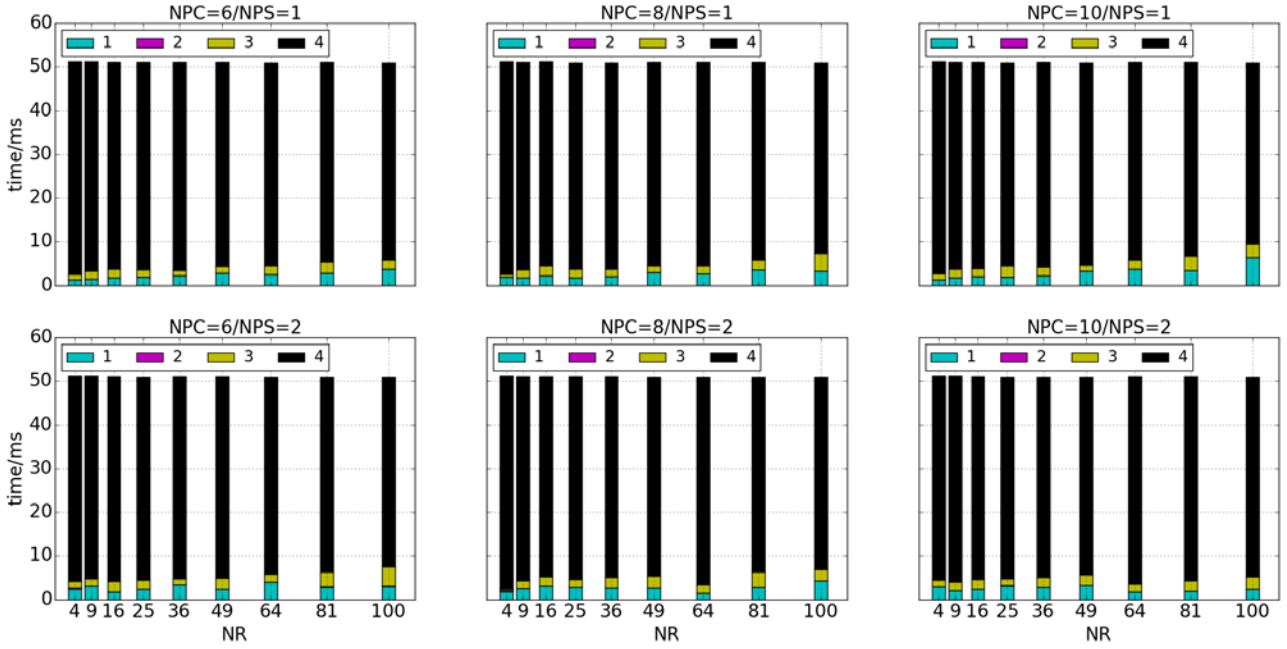
Fig. 11 Timelines for computation and pose share loops in the Topology

Test results are shown in Fig. 12. From Fig. 12A, it can be seen that the application spends most of its time calculating the velocity and waiting for the calculation of other robots. So by increasing NPC, more robots can run in parallel, which will reduce the time in process 3 and subsequently decrease the control latency as shown in Fig. 12A. However, similar to the previous test, increasing NPS will generally cause little degradation in the performance. But with the increasing of NPC such degradation can be ignored.

In the pose share loop as shown in Fig. 12B, time needed for processing and data transmitting takes only a very small portion of the overall control period. The result is the pose share period is always close to the period that is preset. Although there is some overlap between the two loops, the



(A) Control Loop time consumptions



(B) Pose Share Loop time consumption

Fig. 12 Time consumption in the Topology. Figure (A) shows time consumption for each part of the Control Loop as shown in Fig. 11 with different combinations of NPC (Number of parallelism for Velocity Compute Bolt) and NPS (Number of parallelism for Get Robot State Bolt). Figure (B) shows time consumption for each part of the Pose Share Loop with different combinations of NPC and NPS.

influence of pose share loop on the control loop can be generally ignored.

The overall velocity command delay is shown in Fig. 13. Here we see that after the velocity command is emitted from the Topology, it still needs some time to get to the robot. The extra delay in this process contains both communication latency and broker and ROS message routing latency. When NPC is 6, the maximum extra delay is round 100ms, and after increasing NPC to 10, more velocity command messages can be published in parallel, so the maximum extra delay reduces to around 70ms.

Since the number of parallel instances for Velocity Compute Bolt dominates the overall control delay for each robot, it is important to analyze the relationship between NPC and the maximum number of robots that the application can serve while keeping command latency close to the preset

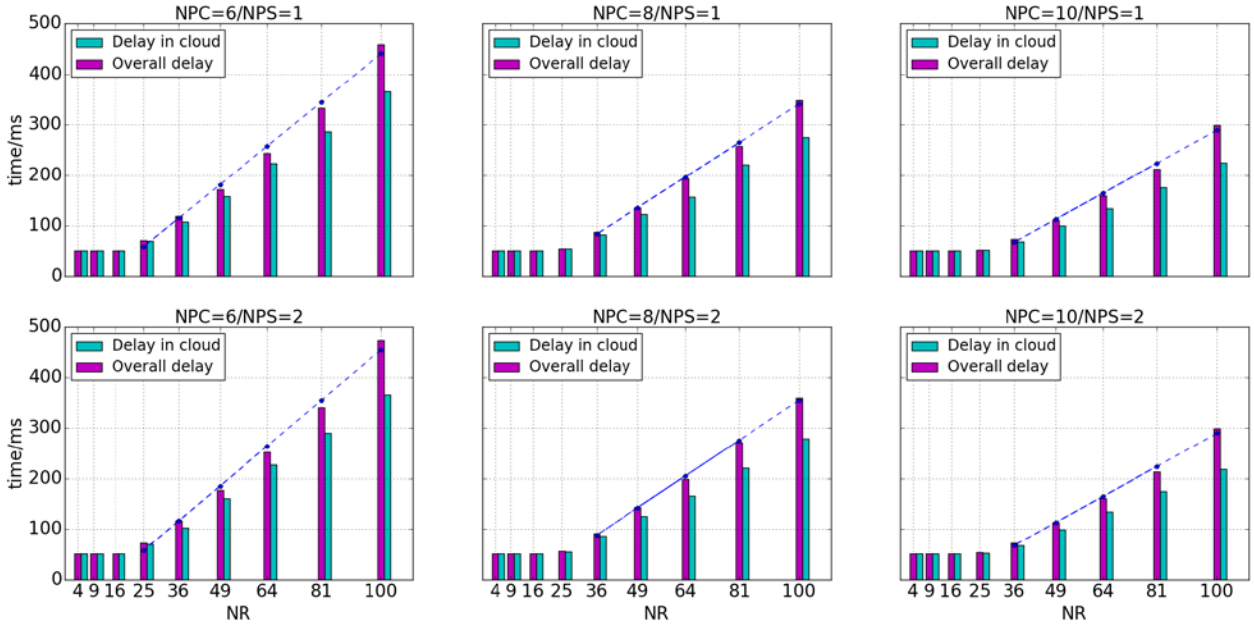


Fig. 13 Overall control latency. This figure shows the overall command delay and the delay caused by computation and communication in the IoT Cloud with different combinations of NPC (Number of parallelism for Velocity Compute Bolt) and NPS(Number of parallelism for Get Robot State Bolt).

control period. To do so, the relationship between the overall control latency and the number of robots should be analyzed first. Then the maximum number of robots for control latency that is around the preset control period can be determined. Fig. 13 shows that when the overall delay is longer than the control period, which is set to 50ms in this test, the relationship between NR and the delay is close to linear. Thus a linear function as shown in equation (1) can be used to formulate the relationship between NR and the overall delay.

$$\begin{cases} t = k * NR + b \\ t > T + \Delta T \end{cases} \quad (1)$$

While t is the overall control delay, NR is the number of robots, k and b are coefficients for the linear function, and ΔT is the execution period of the Time Spout. The reason for selecting $t > T + \Delta T$ is that only under this condition can the instance run in full load. which can reflect the computation capacity of the instance.

Table 2 Maximum number of robots that the application can serve with different NPCs

NPC	NPS	k	b	n*
6	1	5.09	-68.2	23
6	2	5.29	-74.57	23
8	1	4.02	-61.07	27
8	2	4.17	-62.11	26
10	1	3.46	-57.04	30
10	2	3.47	-57.43	30

Using Linear Regression Analysis, k and b can be calculated and the fitted curve is shown in Fig. 13. k and b for different NPC is listed in Table 2. By solving inequality (2), the maximum number of robots that the application is able to run for different NPCs can be found. The result is also in Table 2. This can be used to decide the computation resource that is required for controlling a certain number of robots. It is also the basis for system scaling and load balancing.

$$k * NR + b \le T \quad (2)$$

C. Discussion and future work

In applications and platforms that involve wide area network communication, the data transmitting latency is always the overhead that cannot be ignored. This has been demonstrated in Fig. 13. However the tests in this paper run all robots in one desktop machine, thus computation load, especially the graphic computation of the simulator, and communication load are centralized on one node. Such burdens can be greatly relieved in real robot systems as they do not need so much computation resource for simulation purposes, and communication can be distributed in several Gateways. That means more robots can be effectively controlled by the IoT Cloud. But if the number of robots increases to certain large values like 1000 or more, communication overhead still should be considered carefully.

Lastly, data transmitting delay in the cloud can also be tricky. As for Storm, currently tasks are distributed by the Nimbus node and cannot be set by program or manually by commands. Therefore connected tasks that distributed in different machines will suffer longer communication delay than those distributed in the same machine. These are the problems that need to be explored in the future.

VI. CONCLUSION

This paper provides a novel IoT Cloud-based computation framework for Swarm robotics. To demonstrate the viability of the framework for real time control of large numbers of robots, a local collision avoidance algorithm is implemented as an IoT Cloud application. Unlike other research work that tries to minimize computation cost by ignoring important real world factors, our setup adopts a complex algorithm that can reflect more details about the real world scenario. These computation-intensive tasks are transferred to the Cloud, so that robots or other intelligent entities can be simplified in both hardware and software. By offloading computation to the IoT Cloud, more complex entities can be studied in Swarm Robotics/Intelligence without trimming off the details of the entity. Such precise implementation of the entity model can lead to deeper insight into swarm characteristics.

By implementing and testing the collision avoidance algorithm on the IoT Cloud platform, scaling and real time controlling ability of the system is verified. The results demonstrate that by simply scaling the computation resources, one IoT Cloud application can provide service for more robots. Such features will greatly facilitate extending the population in Swarm robotics and also provide support for large-scale Swarm systems.

ACKNOWLEDGEMENT

The work carried out in this paper is partially funded by the CSC (Chinese Scholarship Council) of Chinese government. The authors also 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] G. C. Fox, S. Kamburugamuve, R. D. Hartman. Architecture and measured characteristics of a cloud based internet of things[C]. 2012 International Conference on Collaboration Technologies and Systems (CTS). 2012: 6-12.

[2] G. Beni. From swarm intelligence to swarm robotics[M], in Swarm Robotics: Springer, 2005, pp. 1-9.

[3] H. Woern, M. Szymanski, J. Seyfried. The I-SWARM project[C]. The 15th IEEE International Symposium on Robot and Human Interactive Communication. 2006: 492-496.

[4] M. Dorigo, E. Tuci, R. Groß, et al. The swarm-bots project[M], in Swarm Robotics: Springer, 2005, pp. 31-44.

[5] E. Şahin. Swarm robotics: From sources of inspiration to domains of application[M], in Swarm robotics: Springer, 2005, pp. 10-20.

[6] G. Mohanarajah, D. Hunziker, R. D'Andrea, et al. Rapyuta: A cloud robotics platform. IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING[J]. 2014, (To be published).

[7] D. Lorencik, P. Sincak. Cloud Robotics: Current trends and possible use as a service[C]. 2013 IEEE 11th International Symposium on Applied Machine Intelligence and Informatics (SAMI). 2013: 85-88.

[8] B. Kehoe, S. Patil, P. Abbeel, et al. A survey of research on cloud robotics and automation. IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING[J]. 2015, (To be published).

[9] K. Kamei, S. Nishio, N. Hagita, et al. Cloud networked robotics. IEEE Network[J]. 2012, 26(3): 28-34.

[10] L. Agostinho, L. Olivi, G. Feliciano, et al. A cloud computing environment for supporting networked robotics applications[C]. 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC). 2011: 1110-1116.

[11] J. Van Den Berg, S. J. Guy, M. Lin, et al. Reciprocal n-body collision avoidance[M], in Robotics research: Springer, 2011, pp. 3-19.

[12] P. Fiorini, Z. Shiller. Motion planning in dynamic environments using velocity obstacles. The International Journal of Robotics Research[J]. 1998, 17(7): 760-772.

[13] J. Van den Berg, M. Lin, D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation[C]. IEEE International Conference on Robotics and Automation ICRA 2008. 2008: 1928-1935.

[14] J. Snape, J. van den Berg, S. J. Guy, et al. Independent navigation of multiple mobile robots with hybrid reciprocal velocity obstacles[C]. IEEE/RSJ International Conference on Intelligent Robots and Systems IROS 2009. 2009: 5917-5922.

[15] J. Alonso-Mora, A. Breitenmoser, M. Rufli, et al, Optimal reciprocal collision avoidance for multiple non-holonomic robots[M]: Springer, 2013.

[16] D. Hennes, D. Claes, W. Meeussen, et al. Multi-robot collision avoidance with localization uncertainty[C]. Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems. 2012: 147-154.

[17] D. Claes, D. Hennes, K. Tuyls, et al. Collision avoidance under bounded localization uncertainty[C]. 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 2012: 1192-1198.

[18] S. J. Guy, J. Chhugani, C. Kim, et al. Clearpath: highly parallel collision avoidance for multi-agent simulation[C]. Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation. 2009: 177-187.

[19] L. Turnbull, B. Samanta. Cloud robotics: Formation control of a multi robot system utilizing cloud infrastructure[C]. Southeastcon, 2013 Proceedings of IEEE. 2013: 1-4.

[20] M. T. Jones. Process real-time big data with Twitter Storm. IBM Technical Library[J]. 2013.

[21] https://zookeeper.apache.org/.

[22] http://www.ros.org/.

[23] L. Hugues, N. Bredeche. Simbad: an autonomous robot simulation package for education and research[M], in From Animals to Animats 9: Springer, 2006, pp. 831-842.

[24] S. Thrun, J. J. Leonard. Simultaneous localization and mapping[M], in Springer handbook of robotics: Springer, 2008, pp. 871-889.