A Software Stack for Composable Cloud Robotics System

Yuan Xu[1,2(✉)], Tianwei Zhang[3], Sa Wang[1,2], and Yungang Bao[1,2]

[1] State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China
xuyuan@ict.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Nanyang Technological University, Singapore, Singapore

Abstract. Modern cloud robotic applications face new challenges in managing today's highly distributed and heterogeneous environment. For example, the application programmers must make numerous systematical decisions between the local robot and the cloud server, such as computation deployment, data sharing and function integration.

In this paper, we propose RobotCenter, a composable cloud robotics operating system for developing and deploying robotics applications. RobotCenter provides three key functionalities: runtime management, data management and programming abstraction. With these functionalities, RobotCenter enables application programmers to easily develop powerful and diverse robotics applications. Meanwhile, it can efficiently execute these applications with high performance and low energy consumption. We implement a prototype of the design above and use an example of AGV/UAV cooperative transport application to illustrate the feasibility of RobotCenter. In the experiment, we reduce the total energy consumption and mission completion time up to 41.2% and 51.5%, respectively.

Keywords: Cloud robotics · Runtime management · Data management · Programming abstraction · Composable system

1 Introduction

Over the past decade, we have witnessed two revolutionary hardware trends: the massive-scale cloud computing platforms and plentiful, cheap wireless networks. These technologies have triggered the killer applications of edge-cloud revolution – cloud robotics [6,27,30,31]. On one hand, robotic applications, such as Microsoft FarmBeats [46] and Dji FlightGub [12], enable robots to publish and share information with other edge robots or sensors. On the other hand, Rapyuta [37] and Amazon IoT Greengrass [8] provide a platform for robots to offload computation-intensive tasks on cloud servers. The rise in demands for those applications has led more researchers and practitioners to develop cloud robotics systems for computation offloading and data sharing.

M. Qiu (Ed.): ICA3PP 2020, LNCS 12453, pp. 705–720, 2020.
https://doi.org/10.1007/978-3-030-60239-0_48

The state-of-the-art cloud robotics systems mainly fall into two ends of the spectrum in the robotic community. One end is the monolithic systems, which tightly integrate edge robots with cloud servers for specific tasks, including automated logistics (e.g. Amazon Kiva [22]) and precision agriculture (e.g. Microsoft FarmBeats [7]). However, these systems are usually ad-hoc and hard to be extended with other kinds of robots or customized tasks [44].

The other end is the robotic middlewares (e.g. ROS [2], Player [20]). They provide an abstraction layer to facilitate coordination and message passing between different modules of robots [23]. All the modules are loose-coupled and easy to be integrated with third-party libraries, such as OpenCV [10], PCL [11] and so forth. Those flexible robotic middlewares have drawn much attention and became the de facto operating systems for developing robotic applications. Those middlewares work well for managing one or a few robots. However, there are still a lot of unresolved challenges when managing a group of heterogeneous robots. Cloud robotics application programmers have to take into consideration plenty of complicated resource management details by their own, e.g. availability, adaptability, fault-tolerance and reactivity.

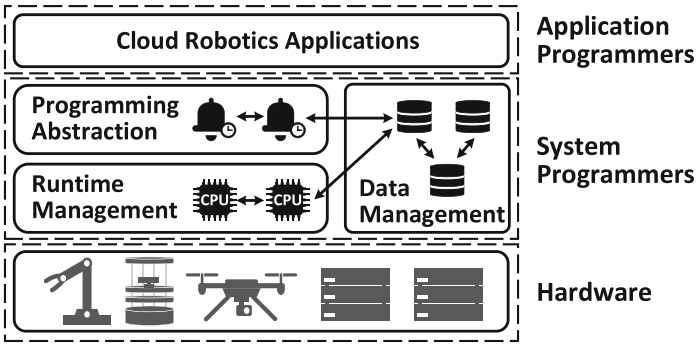

Fig. 1. The robot application environments and system architectures.

In this paper, we argue that the cloud robotics applications need a new software stack to manage and share multi-robots resources. Figure 1 shows the robot application environments and system architectures. The bottom layer is the hardware, containing computing processor, memory, sensors, etc. The middle layer is the software stack, developed by system programmers. The top layer is the robotics applications developed by application programmers. The software stack should provide at least three key functionalities: 1) **Runtime Management**: the stack should make decisions when (dynamic migration) and where (robots or cloud) to deploy user-defined jobs. 2) **Data Management**: the stack should maintain a bunch of state information about each robot, such as ID, heartbeats, maps and so forth, which can be shared among multiple robots. 3) **Programming Abstraction**: the stack should provide a high-level and easy-to-use interface for application programmers to manage the underlying robot resources.

We further propose RobotCenter, a software stack for *cloud robotics composable systems*. Specifically, (1) for *runtime management*, we implement `Avalon`, an adaptive resource management system, to provide the functionalities of computation offloading for better performance and energy usage. (2) For *data management*, we introduce Shared Knowledge Base (`SKBase`), a shared memory database to maintain the data for application programmers and provide data abstraction for availability. (3) For *programming abstraction*, we propose `ActionFlow`, a cloud robotics programming framework to provide modularity through a collection of pluggable planner modules that extend the software stack to support application-specific scheduling needs.

The combination of `Avalon`, `SKBase` and `ActionFlow` provide a composable cloud robotics system that manages distributed computation and data for cloud robotics applications. Programmers can develop their applications with modularized interfaces without considering availability, adaptability, fault-tolerance and reactivity. Our prototype show that the processing time of path panning process in cloud based multi-robot cooperative transport application can be reduced by 52%.

The rest of this paper is organized as follows: In Sect. 2, we discuss the main challenges and requirements for designing a cloud robotics OS. The mechanisms and functionalities of RobotCenter framework is presented in Sect. 3. In Sect. 4, we use an example of AGV/UAV cooperative transport application to illustrate the feasibility and experimental results of RobotCenter. Section 5 discusses some open problems and our future work. Section 6 concludes this paper.

2 Background

2.1 Cloud Robotics Application

This section first describes the characteristics of developing a cloud robotics application, and then discusses the new challenges and requirements for designing a cloud robotics OS.

Development Characteristics. Today, robotic applications have become inherently *distributed*, with data and computation spreading across edge robots and cloud servers. As a result, modern robotic application development has acquired two new characteristics:

1. **A distributed computation deployment.** Different from past robotic applications that run on a single robot, modern cloud robotics applications distribute computation across edge robots and cloud servers with wireless network. However, due to uncontrollable environmental factors, such as instability of wireless connections, fluctuation of communication bandwidth, and robot mobility, programmers must consider an adaptive computation scheduling policy under the highly unreliable and unpredictable data transmission.

2. **New reliable and persistence guarantees.** Different from past robotic applications that store robot's states in local file systems, modern cloud robotics applications shift the states storage from edge to cloud back-ends. The definition of the term "states" is the collection of states of a robot and its environment. Such a distributed data storage can incur more frequent faults, and make it harder to recover them. It is challenging to satisfy users' requirements of executing cloud robotics applications reliably without process crash or data loss.

Table 1. Cloud robotics OS requirements and challenges

Characteristic	Requirement	
Distributed computation deployment	**Availability**:	Remain usable with unreachable nodes
	Adaptability:	Respond with low latency under variable performance
Reliable and persistence guarantees	**Fault-tolerance**:	Ensure data is not lost on crashes
	Reactivity:	Propagate state updates automatically
Inherent application design	**Modularity**:	Simple to reuse at low cost

Application Development. As shown in Table 1, the two characteristics lead to a set of new challenges for programmers while developing cloud robotic applications. These challenges dictate the requirements for a cloud robotics OS which guide our stack design.

The first two challenges and requirements stem from the distributed computation deployment. Since mobile robots execute tasks in a highly unpredictable environment, the interaction between edge robots and cloud is influenced by many incontrollable factors. For example, As a robot keeps moving, its uplink and downlink bandwidth can fluctuates frequently, or even get disconnection, due to the absorption of electromagnetic waves or wireless interference with other devices. As a result, programmers must ensure that their applications are *available* so the robot can continuously execute tasks with unreachable nodes in the cloud. Meanwhile, the benefit from computation offloading can be decreased in the poor wireless environment, thus the cloud robotics applications should be *adaptive* to migrate between edge robots and cloud servers.

The next two challenges and requirements arise from new reliable and persistence guarantees. Robotic applications are used to be executed in a closed-loop streaming processing model, e.g. sensing data from the environment, interpreting

and making decision, navigating or modifying the environment. Each component generates large amounts of states that need to be stored in either edge robots or cloud. Consequently, cloud robotics applications must periodically checkpoint or log to the storage system for *fault-tolerance*, and keep *reactive* to automatically propagate the state updates from each component to persistent storage.

The final challenge and requirement is a consequence of inherent cloud robotics application design. *Modularity* plays an important role in the rapid development of applications, as it allows programmers to develop a new application from existing components. So, programmers must make their code easier to be integrated in other robotic applications to meet different demands of dynamic environment.

2.2 Cloud Robotics OS Abstractions

It is necessary to have a cloud robotics OS to meet the five requirements for developing applications. In this section, we describe the related work from three OS functionalities: runtime management, data management and programming abstraction. We further discuss the issues of current technologies to address these challenges.

Runtime Management. Cloud robotics OSes manage runtime processes of applications and decides when and where to deploy these computations dynamically. Recently, the RoboEarth project [3] introduces a centralized task controller to manage a group of robots [29]. It also uses Rapyuta [37] to offload computation to clouds and speed up computation-intensive robotic applications such as navigation [29] and SLAM [39,40]. Amazon recently released a cloud robotics platform, RoboMaker [9], which provides a cloud-based platform for robotic application development and simulation. Programmers can deploy their applications in the edge robot and cloud through IoT Greengrass [8].

These cloud runtime systems support lightweight containers that allow application programmers to easily execute customized processes in the cloud (*modularity*). They provide basic runtime management that monitors and updates states of each robot periodically (*reactivity*), detects and restarts crashed containers (*fault-tolerance*).

Issue #1: Both Rapyuta and RoboMaker systems are in short of *adaptivity* and *availability*. Specifically, they require programmers to set up and deploy containers without automatically computation migration for better performance or energy usage, and neglect the network disconnection due to unpredictable wireless network (Table 2).

Data Management. Cloud robotics OSes also manage the storage distribution of robots' states. For example, the KnowRob [15,45] and the RoboBrain [5] intend to build a robotic wikipedia to share information that guide robots to perform actions and interact with objects.

Table 2. Comparison of our stack to prior work for cloud robotics application development

OS	Availability	Adaptability	Fault-tolerance	Reactivity	Modularity
Rapyuta [37]			✓	✓	✓
RoboMaker [9]			✓	✓	✓
Cloud Database [5,45]			✓	✓	✓
ROS [2]					✓
SOA [32,35,36,40,42]					✓
Monolithic OS [7,22]	✓	✓	✓	✓	
Our Framework: RobotCenter	`SKBase` ($3.2)	`Avalon` ($3.1)	`SKBase` ($3.2)	`SKBase` ($3.2)	`ActionFlow` ($3.3)

These cloud database systems store the logic data in the relation database (e.g. Sesame [17]) and the binary data in the distributed storage systems (e.g. Apache HDFS [1]). The robot is regarded as a completely stateless mobile client that can query and analyze knowledge representation languages (e.g. Web Ontology Language [4]) that are related to objects and maps of applications. Application programmers can continuously checkpoint to these storage systems for *fault-tolerance* and periodically poll to update *reactive* state updates.

Issue #2: These cloud database systems do not meet the requirements of *adaptivity* and *availability*. While applications are implemented with stateless mobile clients, users cannot access the data in cloud storage under poor wireless bandwidth. Moreover, replicating data to the cloud increases the availability of the applications at the cost of responsiveness. It is ineffective for some tasks with strong real time constraints for fault recovery.

Programming Abstraction. The key role of an Cloud Robotics OS is to provide abstractions and APIs, hiding the underlying complex implementations of robotic functions and hardware features for the programmers. ROS (Robot Operating System [2]) makes a great progress towards programming abstraction through providing a series of common libraries for low-level device control and inter-process communication. Application programmers achieve functions via launching a set of concurrent long-running processes, called "nodes". The communication between two nodes can be either a two-way request/reply pattern called *rosservice* or a many-to-many publish/subscribe pattern called *rostopic*. Application programmers can easily achieve these functions by sending/subscribing to each function with ROS-type message. Besides, researchers in the robotic community have proposed many cloud robotics services to meet the needs of different applications, such as 3D Mapping [40,42] and grasp planning [32,35,36].

Issue #3: Unfortunately, ROS just provides the basic communication and hardware control abstraction. Application programmers need to implement all cloud robotics application requirements for themselves. Besides, current Service-Oriented Architectures (SOA) frameworks focus on the efficiency of algorithm implementations and cannot meet other OS requirements.

3 RobotCenter Framework

This section describes the mechanisms and functionalities of RobotCenter framework. We present how RobotCenter meets five requirements discussed in Sect. 2.

3.1 Runtime Management: Avalon

We designed a resource management framework, `Avalon`, to achieve *adaptability* in RobotCenter. `Avalon` implements an adaptive energy-efficient scheduler to migrate the computation across edge robots and cloud servers automatically under unpredictable wireless transmission latency. Although adaptive computation offloading problem is widely studied in mobile cloud computing domain (e.g. CloneCloud [19], MAUI [21] and ThinkAir [33]), they are not well suited for robotics applications because of several reasons. First, robotic applications are multi-process while mobile applications are multi-thread without exposing sockets. Second, the offloading decision model of robotic application depends on not only data transmission latency, task processing time and CPU frequency, but also some robotic factors, e.g. velocity and decision accuracy.

To ensure the responsiveness of various robotic applications, `Avalon` delegates control over scheduling to the framework with a customized energy-efficient utility model. Specifically, the energy-efficient utility function is characterized along two dimensions: *total completion time* and *total energy consumption*. The total completion time (TT) denotes the time a mobile robot completes the mission, which depends on the traveling distance and robot's velocity. In contrast, the total energy consumption (TE) means the energy consumed during the mission execution. Thus, we propose the energy-efficient cost (EEC) by the following definition.

Definition 1 (Energy-Efficiency Cost (EEC)): The energy-efficiency cost (EEC) is defined as the weight sum of total completion time and total energy consumption. Thus, the EEC of a robotic workload is given by

$$Z = \gamma^T TT + \gamma^E TE \tag{1}$$

where $0 \leq \gamma^T, \gamma^E \leq 1$ denotes the weights of total completion time and totally energy consumption for the workload execution of the mobile robot. In this equation, we normalize the measurement of TT and TE so that the value of Z will fall in [0, 1]. To provide a scalable and resilient core for enabling various mobile robots to efficiently perform the workload, we allow that different framework programmers can customize different weighting parameters in the decision making. For example, when a mobile robot has a long journey at a low battery state, the robot prefers to set a larger weight on TE to save more energy. When a mobile robot executes some applications that are sensitive to the delay, such as search and rescue, choosing a larger weight on TT would navigate the robot to the destination as soon as possible.

The utility function of local mode Z_{local} (robot only) and remote mode $Z_{remotel}$ (robot + cloud) is different due to the resource characteristics. Thus, the intersection of two utility curves determines the migration time to maximize performance. As discussed in recent works of mobile cloud offloading [18,26], the utility function Z_{local} and $Z_{remotel}$ depend on many factors, such as cpu frequency, transmission time, processing time of computations in local and remote server. Here, we introduce a novel factor that only exists in robot workloads, i.e. decision accuracy. Unlike mobile phones, robots can sacrifice the decision accuracy to reduce computation utilization under constraints [16,44]. For instance, the increase in planning resolution makes the voxels of map larger, so space is represented more coarsely and less accurately, then performance improves due to the less needed computation. However, the mission may fail in a complex environment under a low accuracy. Hence, `Avalon` attaches the accuracy constraints to utility function and helps programmers to determine on redirecting a computation replication with proper accuracy.

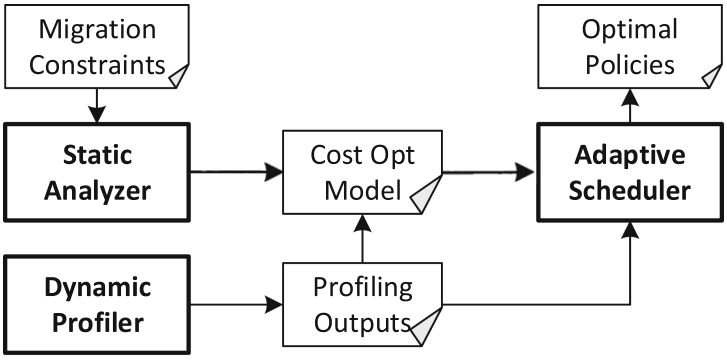

Fig. 2. `Avalon` architecture overview.

As Fig. 2 shows, the `Avalon` implements the utility model with three parts: *Static Analyzer*, *Dynamic Profiler* and *Adaptive Scheduler*. The *Static Analyzer* identifies the legal range of outputs in the utility model, according to a set of customized migration constraints. For example, maximum CPU frequency and maximum velocity limit the upper bound of frequency and speed scaling policy. The *Dynamic Profiler* collects the robot's states (e.g. CPU load, network latency) as the input of the model and sends to the cloud servers periodically. Finally, the *Adaptive Scheduler* finds the optimal decision by minimizing the utility function, including computation migration, edge robot's CPU frequency and planning accuracy control.

3.2 Data Management: SKBase

We propose `SKBase`, a shared knowledge database, to achieve *availability*, *fault-tolerance* and *reactivity* for data management. `SKBase` keeps multiple copies of

states on edge robots and cloud servers for fault recovery, and automatically propagate updates across these copies.

To ensure the *availability* of application effectively, `SKBase` provides shared memory among processes in both edge and cloud. Further, `SKBase` introduces a new data abstraction, called *shared data object* (SDO) as the sharing unit. SDOs support various data type, including simple primitives (e.g. string), collections (e.g. list) and ROS messages (e.g. geometry_msgs). Thus, application programmers can encapsulate robot's states without modifying original data type.

Due to various real time requirements for fault recovery of processes, SDOs should be deposited in different places. For example, the localization process should be auto-restarted with initial position on crash. The application might place the current position state on edge robot for low latency because the robot's position is still changing while moving. On the other hand, states like generated map could be placed on a cloud server for sharing with other robots. Thus, *SKBase* proposes a new data mapping interface to place copies in a cloud-based distributed storage system or edge robot's disk. Application programmers use this interface to link in-memory states to keys in different storage systems and all SDOs with the same key will be automatically synchronized.

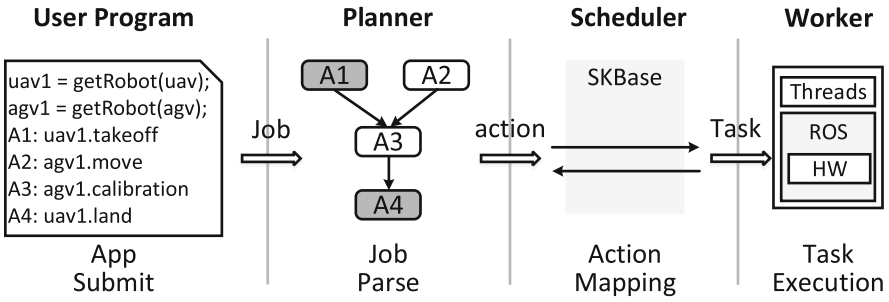

Fig. 3. `ActionFlow` execution process.

3.3 Programming Abstraction: ActionFlow

We propose `ActionFlow`, a ROS based *modularized* cloud robotics programming framework, which provides a RPC-like action abstraction for job scheduling. `ActionFlow` delegates data management and resource management to `SKBase` and `Avalon` respectively. Thus, RobotCenter meets all five cloud robotics application requirements.

`ActionFlow` encapsulates the SOA frameworks as a collection of pluggable modules that support application-specific scheduling needs. Figure 3 presents the basic components and execution workflow in `ActionFlow`. Specifically, application programmers submit their work to `ActionFlow` in the form of *jobs*, each of which consists of one or more *actions* that all run the same program. System

programmers design different planners to parse jobs into different actionDAGs. The scheduler then maps each action into a corresponding ROS task through `SKBase` and sends it to available robots. After the worker in each robot completes the task, it returns the results to the planner and updates the data. Each robot works in a stateless process, which means they are agnostic about the job collaboration, while the action scheduling is decided by the planner.

Another contribution of `ActionFlow` is the RPC-like action abstraction based on action mapping, which hides ROS relevant interfaces for interacting with robots. `ActionFlow` provides various APIs for different types of actions to simplify development of cloud robotics frameworks through storing mapping from its action interface to the a specific ROS task.

4 Use Case and Evaluation

We implemented a prototype[1] of the design above to illustrate the feasibility of RobotCenter. We use the example of AGV/UAV cooperative transport application. In this workload, both AGV (automatic guided vehicle) and UAV (unmanned aerial vehicle) are put in an obstacle-filled environment. The UAV takes off and sends its position to the AGV periodically. The AGV navigates to the designated position and calibrates the relative position with vision recognition to ensure correctness. At last, the UAV lands on the AGV to simulate cargo offloading. RobotCenter programmers can decompose the job into four actions with `ActionFlow` action interfaces, including (1) uav.takeoff(height), (2) agv.move(uav.position), (3) agv.calibration(uav.model), and (4) uav.land(agv.height).

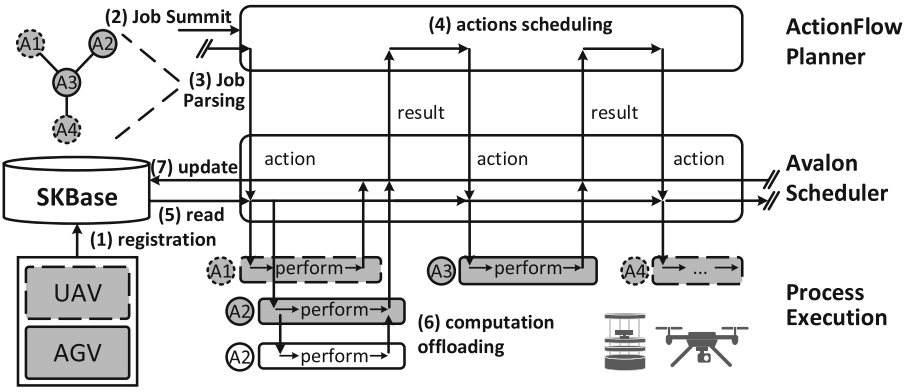

Fig. 4. A time-line sketch of cooperative transport. All white blocks construct Robotcenter base built in gateway, and grey blocks are launched in robots.

[1] The experiment video can be found in https://youtu.be/KeYyS6lZxo0.

In our experiment, We deployed the Dji Spark and turtlebot in our lab. The turtlebot is equipped with an Intel i5-2450M CPU @ 2.5 GHz, 1.8 GB of RAM netbook and a Dji guidance over the top board. The two robots were connected to a powerful gateway server (Intel i7-7700K CPU @ 4.2 GHz with 16 GB of RAM and a GeForce GTX 1080) with a passive 5 GHz band wireless network. In this application, we controlled the turtlebot to create a 2D occupancy grid map of our lab environment and upload it to the `SKBase` as a shared data object.

Figure 4 shows the runtime of the collaboration between the turtlebot and UAV. We describe each step executed (the numbers in Fig. 4 correspond to the numbers in the following list):

1. The AGV and UAV are registered to `Avalon` and the static knowledge is stored in `SKBase`.
2. A cooperative transport job is submitted through the application with some parameters. The application passes the job to the Planner in `ActionFlow`.
3. The Planner parses the job into an actionDAG, including six action primitives shown in Fig. 3.
4. The Planner forwards each action to the Scheduler in order and waits for the feedback of these actions.
5. The Scheduler queries `SKBase` through `ActionFlow` scheduler API to allocate the two robots and map each action to ROS task.
6. The Scheduler sends ROS tasks to the turtlebot and UAV with `Avalon` API. Notice that the worker in turtlebot detects that the processing time of the path planning node is too high and migrates it to the gateway server.
7. The `Avalon` worker executes each task in an ROS environment and returns the result and updates the related information in `SKBase`. The Planner determines the next action until the job is finished.

In the whole life cycle of above job, the 'move' and 'calibration' actions are important because they use `Avalon` to offload computation nodes and `SKbase` to share knowledge respectively. Thus, we discuss the detail implementation of these two actions in the next subsections.

4.1 Move Action

The function of action "move" aims to navigate the robot through an obstacle-filled environment to reach some arbitrary destination. Specifically, it plans an efficient collision-free path in the map and follows. For each movement, it simulates multiple trajectories based on some mechanical characteristics (velocity, acceleration limit), and identifies each trajectory whether conflicts with obstacles. Obviously, this action is a computation-intensive task that consumes much energy of the robot. Moreover, the robot has to sacrifice velocity to meet some constraints of each functional node, such as the accuracy loss in localization and the conflict possibility in obstacle avoidance. The reduction of velocity further prolongs the mission completion time in navigation task. In summary, the limited resource constrains both total energy and mission completion time of this action.

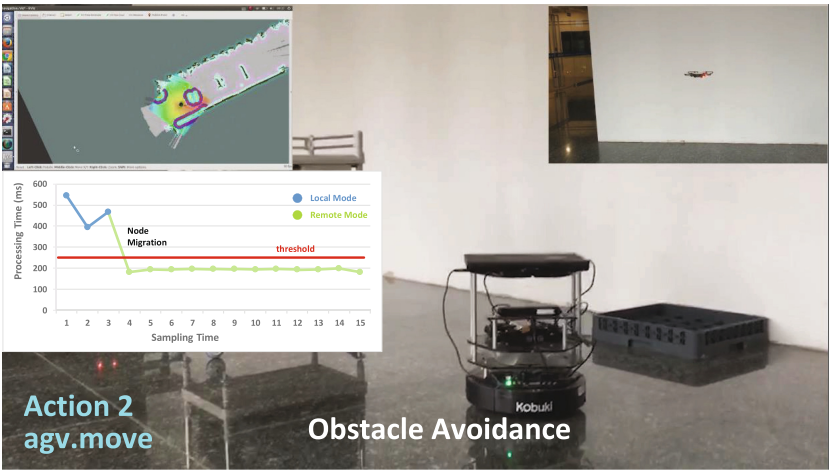

Fig. 5. The computation offloading example of "move" action.

Figure 5 shows the computation offloading example of "move" action in our experiment. We implement our "move" action with ROS navigation stack [13]. In our initial prototype, we simply consider processing time as the only factor to affect our utility function. We preset offloading threshold of processing time of local robot to 250 ms through *Static Analyzer*, which means while the processing time of any functional node in local robot is bigger than 250 ms, it would be migrated to the gateway server for energy and performance optimization through *Adaptive Scheduler*. The *Dynamic Profiler* monitors the processing time of each node in runtime.

While the turtlebot received the position of the UAV from `SKBase`, it launches a group of ROS functional nodes to setup action runtime environment, such as amcl, move_base node. From the figure, we can observe that the processing time of move_base node exceeds our threshold, the `Avalon` automatically migrates the move_base node to the remote gateway server and synchronizes related data (e.g. UAV position, node status) to the `SKBase`. Based on these operations, the processing time of move_base node reduces almost 64%. Besides, the total energy and mission completion time reduce up to 41.2% and 51.5% during the life cycle of "move" action execution.

4.2 Calibration Action

The function of action "calibration" is the process of identifying relative position between the turtlebot and the UAV based on the sensor data. Specifically, the turtlebot moves to the bottom of the UAV and receives the UAV's image through Dji guidance in five directions. To reduce unnecessary computation, once recognizing UAV in one direction, the turtlebot will only process the images from this

direction. All the received images are used to predict the UAV position relative to itself through an on-line image recognition model.

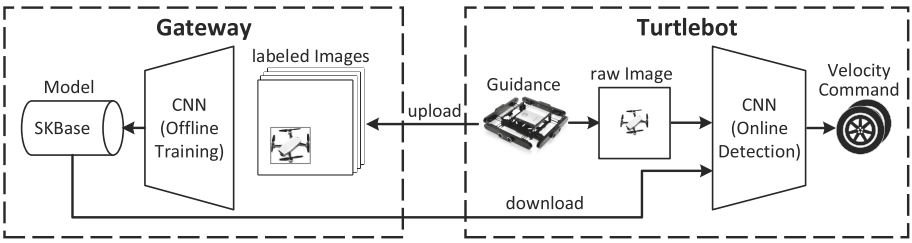

Fig. 6. The implementation of UAV-AGV "calibration" action.

Figure 6 shows the implementation of UAV-AGV "calibration" action. The edge gateway is responsible for receiving images from the turtlebot and we label the UAV position in each image. These preprocessing images will be used for off-line training in a convolutional neural network. The trained model is later saved in the `SKBase` and downloaded by the turtlebot for the on-line detection when the turtlebot makes "calibration" action. For each receiving raw image, the turtlebot identifies the related position of the UAV and then computes the velocity command for calibration.

We use TensorFlow framework [14] to implement an SSD-MobileNet [34, 41] as our image recognition model. The reason we choose the SSD-MobileNet is because the process of on-line detection is sensitive to latency. This model decomposes the standard convolutional layer into depth-wise convolutional layer and point-wise convolutional layer, which decreases the numbers of parameter, reduces the computation complexity, and further shortens the detection time. In our experiment, we trained 500 labeled images with the UAV through the TensorFlow Objection Detection interface [28] and reduced the loss up to 1.213 after 2700 iterative operations. In the turtlebot side, the detection time reduced up to almost 280 ms. Note that although the processing time of on-line detection is bigger than 250 ms, we did not migration this node because of the real-time constraint.

5 Discussion and Future Work

RobotCenter is our initial attempt to build a general-purpose and composable cloud robotics framework for robotic applications. While this system addresses most of drawbacks of current robotic systems, there are still some open problems listed below, which are our future work.

– **Multi-robot Challenge.** In our use case, we only consider the robotic application with two heterogeneous robots. While the number of robots scales up to

hundreds and access concurrently, RobotCenter must coordinate synchronized accesses to shared data in real time and ensures data sharing without conflicts. Thus, *scalability* and *consistency* will be new requirements and challenges for cloud robotics OS.

- **Mobility Management.** In our initial prototype of Avalon, we only consider the trade-off between processing time and planning accuracy. There are also some others factors that determine the total energy consumption and mission completion time, such as the robot's velocity and the CPU frequency of onboard computer. As a consequence, the velocity control and CPU frequency scaling will be added to our utility model in the future work.
- **Fault Tolerance.** When the computations in the robotic application deploy in both local robot and remote server, the mission failure will occur more frequently due to the unstable wireless communication and robot's mobility. For example, the network will be disconnected while the robot moves far away from the wireless access point. So, how to make RobotCenter robust to adapt for various real-world conditions is a big challenge.
- **Security and Privacy.** The computation offloading and data sharing in the cloud also raises a range of privacy and security concerns [24,25,38,43]. For example, robots may be hacked to send images or video data from private homes or corporate trade secrets to the cloud. Besides, an adversary can also take over a robot to disrupt functionality or cause damage. As a result, RobotCenter should also ensure a secure and private environment for end users.

6 Conclusion

In this paper, we proposed RobotCenter, rethink and reconstruct a novel cloud robotics system in a composable perspective. We described the challenges of designing a cloud robotics system in three levels: *runtime management*, *data management* and *programming abstraction*. We then provided the solutions to address each of these challenges.

We implemented a prototype of the design above three components and use an example of AGV/UAV cooperative transport application to illustrate the feasibility of RobotCenter. In the experiment, we reduced the total energy consumption and mission completion time up to 41.2% and 51.5%, respectively. We believe that the goals we outlined in this paper have a wide range of technical value for software systems and robotic communities. The RobotCenter software stack represents a promising direction for future research of robotics systems.

References

1. Apache Hadoop project. http://hadoop.apache.org/ (2009)
2. Open source robot operating system. http://www.ros.org/ (2009)
3. Roboearth project. http://roboearth.ethz.ch/ (2009)

4. Web ontology language. https://www.w3.org/OWL/ (2013)
5. The robo briain project. http://robobrain.me/ (2015)
6. Mit technology review: Robots that teach each other. https://www.technologyreview.com/s/600768/10-breakthrough-technologies-2016-robots-that-teach-each-other/ (2016)
7. Microsoft farmbeats. https://www.microsoft.com/en-us/research/project/farmbeats-iot-agriculture/ (2017)
8. Amazon aws iot greengrass. https://aws.amazon.com/greengrass/ (2018)
9. Amazon aws robomaker. https://aws.amazon.com/robomaker/ (2018)
10. Open source computer vision library. https://opencv.org/ (2018)
11. The point cloud library. http://pointclouds.org/ (2018)
12. Dji flightgub. https://www.dji.com/flighthub/ (2019)
13. Ros navigation stack. http://wiki.ros.org/navigation/ (2019)
14. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. In: Operating Systems Design and Implementation (OSDI) (2016)
15. Beetz, M., Beßler, D., Haidu, A., Pomarlan, M., Bozcuoğlu, A.K., Bartels, G.: Know Rob 2.0–a 2nd generation knowledge processing framework for cognition-enabled robotic agents. In: International Conference on Robotics and Automation (ICRA) (2018)
16. Boroujerdian, B., Genc, H., Krishnan, S., Cui, W., Faust, A., Reddi, V.J.: Mavbench: micro aerial vehicle benchmarking. In: MICRO (2018)
17. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: a generic architecture for storing and querying RDF and RDF schema. In: Horrocks, I., Hendler, J. (eds.) ISWC 2002. LNCS, vol. 2342, pp. 54–68. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-48005-6_7
18. Chen, X.: Decentralized computation offloading game for mobile cloud computing. IEEE Trans. Parallel Distrib. Syst. **26**(4), 974–983 (2015)
19. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: European Conference on Computer Systems (EuroSys) (2011)
20. Collett, T.H.J., MacDonald, B.A., Gerkey, B.: Player 2.0: toward a practical robot programming framework. In: Proceedings of the Australasian Conference on Robotics and Automation (ACRA) (2005)
21. Cuervo, E., et al.: Maui: making smartphones last longer with code offload. In: International Conference on Mobile Systems, Applications, and Services (MobiSys) (2010)
22. D'Andrea, R.: Guest editorial: a revolution in the warehouse: a retrospective on kiva systems and the grand challenges ahead. IEEE Trans. Autom. Sci. Eng. (T-ASE) **9**(4), 638–639 (2012)
23. Elkady, A.Y., Sobh, T.M.: Robotics middleware: a comprehensive literature survey and attribute-based bibliography. J. Robot. (2012)
24. Elkady, A.Y., Sobh, T.M.: A user-centric data protection method for cloud storage based on invertible DWT. IEEE Trans. Cloud Comput. pp. 1–1 (2017)
25. Elkady, A.Y., Sobh, T.M.: A user-centric data protection method for cloud storage based on invertible DWT. IEEE Trans. Cloud Comput. pp. 1–1 (2019)
26. Guo, S., Xiao, B., Yang, Y., Yang, Y.: Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In: IEEE International Conference on Computer Communications (INFOCOM) (2016)
27. Hu, G., Tay, W.P., Wen, Y.: Cloud robotics: architecture, challenges and applications. IEEE Network **26**(3), 21–28 (2012)

28. Huang, J., et al.: Speed/accuracy trade-offs for modern convolutional object detectors. In: Computer Vision and Pattern Recognition (CVPR) (2017)
29. Janssen, R., van de Molengraft, R., Bruyninckx, H., Steinbuch, M.: Cloud based centralized task control for human domain multi-robot operations. Intel. Serv. Robot. **9**(1), 63–77 (2015). https://doi.org/10.1007/s11370-015-0185-y
30. Kamei, K., Nishio, S., Hagita, N., Sato, M.: Cloud networked robotics. IEEE Network **12**(2), 432–443 (2012)
31. Kehoe, B., Abbeel, P., Goldberg, K.: A survey of research on cloud robotics and automation. IEEE Trans. Autom. Sci. Eng. (T-ASE) **12**(2), 398–409 (2015)
32. Kehoe, B., Matsukawa, A., Candido, S., Kuffner, J., Goldberg, K.: Cloud-based robot grasping with the google object recognition engine. In: International Conference on Robotics and Automation (ICRA) (2013)
33. Kosta, S., Aucinas, A., Hui, P., Mortier, R., Zhang, X.: Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: IEEE International Conference on Computer Communications (INFOCOM) (2012)
34. Liu, W., et al.: SSD: single shot multiBox detector. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9905, pp. 21–37. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46448-0_2
35. Mahler, J., et al.: Dex-Net 2.0: deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics. In: Robotics: Science and Systems (RSS) (2017)
36. Mahler, J., Matl, M., Liu, X., Li, A., Gealy, D.V., Goldberg, K.: Dex-Net 3.0: computing robust vacuum suction grasp targets in point clouds using a new analytic model and deep learning. In: International Conference on Robotics and Automation (ICRA) (2018)
37. Mohanarajah, G., Hunziker, D., D'Andrea, R., Waibel, M.: Rapyuta: a cloud robotics platform. IEEE Trans. Autom. Sci. Eng. (T-ASE) **12**(2), 481–493 (2015)
38. Ren, K., Wang, C., Wang, Q.: Security challenges for the public cloud. IEEE Internet Comput. **16**(1), 69–73 (2012)
39. Riazuelo, L., Civera, J., Montiel, J.M.M.: C2tam: a cloud framework for cooperative tracking and mapping. In: Robotics and Autonomous Systems (RSS) (2014)
40. Riazuelo, L., et al.: Roboearth semantic mapping: a cloud enabled knowledge-based approach. IEEE Trans. Autom. Sci. Eng. (T-ASE) **10**(3), 643–651 (2015)
41. Sandler, M., Howard, A.G., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenets: efficient convolutional neural networks for mobile vision applications. In: CoRR abs/1704.04861 (2017)
42. Schneider, T., et al.: Maplab: an open framework for research in visual-inertial mapping and localization. IEEE Rob. Autom. Lett. **3**(3), 1425–1428 (2018)
43. Shao, Z., et al.: Security protection and checking for embedded system integration against buffer overflow attacks via hardware/software. IEEE Trans. Comput. **55**(4), 443–453 (2006)
44. Stoica, I., et al.: A Berkeley view of systems challenges for AI. In: CoRR abs/1712.05855 (2017)
45. Tenorth, M., Perzylo, A.C., Lafrenz, R., Beetz, M.: Representation and exchange of knowledge about actions, objects, and environments in the RoboEarth framework. IEEE Trans. Autom. Sci. Eng. (T-ASE) **10**(3), 643–651 (2013)
46. Vasisht, D., et al.: A cloud robot system using the dexterity network and Berkeley robotics and automation as a service (Brass). In: USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2017)