Resource Manager for Scalable Performance in ROS Distributed Environments

Daisuke Fukutomi*, Takuya Azumi†, Shinpei Kato‡, Nobuhiko Nishio*
* Graduate School of Information Science and Engineering, Ritsumeikan University, Japan
† Graduate School of Science and Engineering, Saitama University, Japan
‡ Graduate School of Information Science and Technology, The University of Tokyo, Japan
Email: tommy@ubi.cs.ritsumei.ac.jp, takuya@mail.saitama-u.ac.jp, shinpei@pf.is.s.u-tokyo.ac.jp, nishio@is.ritsumei.ac.jp

***Abstract*—This paper presents a resource manager to achieve scalable performance in Robot Operating System (ROS) for distributed environments. In robotics, using ROS in distributed environments via multiple host machines is trending for large-scale data processing, for example, cloud/edge computing and the data communication of point clouds and images in dynamic map composition. However, ROS is unable to manage the resources (e.g., the CPUs, memory, and disks) on each host machine. Therefore, it is difficult to use distributed environmental resources efficiently and achieve scalable performance. This paper proposes a resource management mechanism for ROS distributed environments using a master-slave model to execute ROS processes efficiently and smoothly. We manage the resource usage of each host machine and construct a mechanism to adaptively distribute the load to be balanced. Evaluations show that scalable performance can be achieved in ROS distributed environments comprising ten host machines using a real application (SLAM: simultaneous localization and mapping) processing large-scale point cloud data.**

I. Introduction

Recently, sensors and applications have been increasing applied in robotics fields, e.g., autonomous driving systems [1], [2] and industrial robotic development [3]. Accordingly, software frameworks to utilize rich libraries and improve the development efficiency have attracted attention [4]. Consequently, Robot Operating System (ROS) [5] is used as the de facto standard [6] because it corresponds to open source libraries, e.g., OpenCV for image processing and the Point Cloud Library (PCL) [7]. Considering the separation of the sensor processing and control systems, ROS supports the distribution of processes to multiple host machines.

In ROS distributed environments, several studies [8], [9] have considered multiple robots as resources for edge computing, and processing large-scale data (e.g., point clouds and images) such as cloud computing. However, ROS distributed environments do not manage resources, e.g., the CPUs, memory, and disks of each host machine such as robots and computers. Thus, process allocation between host machines in a distributed environment is imbalanced, meaning that their resources cannot be used efficiently; therefore, these studies using ROS distributed systems have inadequate scalability.

Prior studies [10], [11] and the introduction of the Data Distributions Service (DDS) [12] to ROS2 [13], which is the next generation of ROS, are presented to improve fault tolerance and availability in ROS distributed environments. However, these approaches have no effect from the perspective of resource management and scalability; therefore, efficient processing cannot be achieved.

This paper presents a resource manager for ROS distributed environments. To efficiently and smoothly operate in a ROS distributed environment, the proposed mechanism uses a master-slave method to consider the number of resources used by each host machine. During any imbalanced allocation of processes between host machines, the resources manager migrates the process to another host machine that has a margin in that resource. In the case of migration, if the process refers to the internal state, e.g., the estimation of a vehicle's position in simultaneous localization and mapping (SLAM) of the previous cycle, the internal state is lost. Therefore, to back up the state information while applying the ROS communication method, we construct a mechanism that can be applied to stateful processes.

This paper makes the following three contributions:

1) The constructed ROS resource manager prevents deviation processing in ROS distributed environments, making it efficient at allowing processing using large-scale data in ROS distributed environments.
2) Stateful process migration is realized by applying the ROS communication system. The resource manager can be used suitably in ROS distributed environments.
3) The resource manager can be used for scalable performance in real applications, e.g., SLAM.

This paper is organized as follows: A brief overview of the system of ROS distributed environments is provided in Section II. The design and implementation of the proposed resource manager are presented in Section III. Section IV presents the experimental results. Section V presents the related studies. Section VI concludes the paper and presents future work.

II. System model

ROS is a software platform that provides libraries and tools for robotic development. A ROS application is developed in units called packages, each package comprises nodes corresponding to Linux processes, which subdivide the functions of an application. Figure 1 shows a system model of the proposed ROS resource manager. The ROS resource manager is implemented as a middleware that runs on ROS to manage all applications.

The ROS resource manager acquires and manages the resource information of all the host machines in the distributed environment. Furthermore, it allocates and migrates each node based on the resource information. By adopting this system model, the resource information of the host machines in the

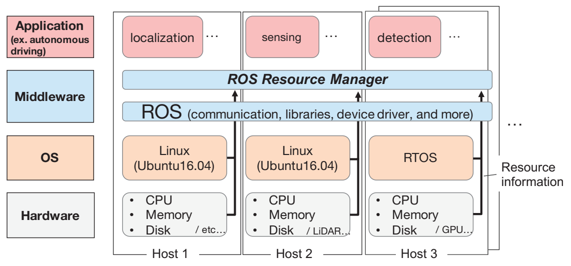

Fig. 1: System model of the ROS resource manager.

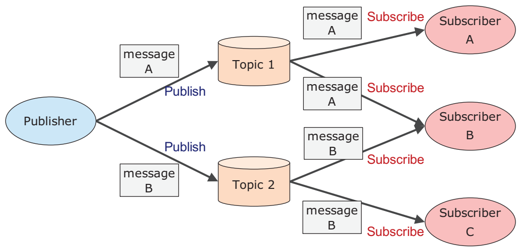

Fig. 2: Publish/subscribe model.

distributed environment can be managed and the processes can be efficiently allocated without changing the source code of the application. Hardware driver nodes such as sensors and actuators connected to the host machine must be launched on the host machine where the hardware resides unless the application code is changed. As the resource manager acquires the resource utilization from the OS, it can consider the computing resources used by the driver nodes when allocating other processes. In ROS, the data obtained from a sensor are delivered as though the processes that use sensor information can be launched on any host machine. The resource manager allocates these processes to the appropriate host machine without hardware constraints.

We describe the two methods of communication between ROS nodes to explain the ROS resource manager. The first means of communicating between ROS nodes are topics, which perform publish/subscribe-type communication as shown in Fig. 2. In ROS, the publisher/subscriber model shown in Fig. 2 is equivalent to a node. A topic comprises a bus of data specifying data types and contents, and the publisher and subscriber perform one-way asynchronous messaging via specified topic names. The second means of communicating between ROS nodes are services, which provide communication with respect to the server-client architecture. Services provide synchronous communication that enables the bidirectional transmission and reception of information with two types of data: request and response.

III. Design and Implementation

A. Design choice

We designed and implemented a ROS resource manager by adopting the master-slave method. This method is also used for Yet Another Resource Negotiator (YARN) [14], which is a representative resource management mechanism in distributed environments. Master-slave has high affinity with ROS adopting the publish/subscribe model as a communication method

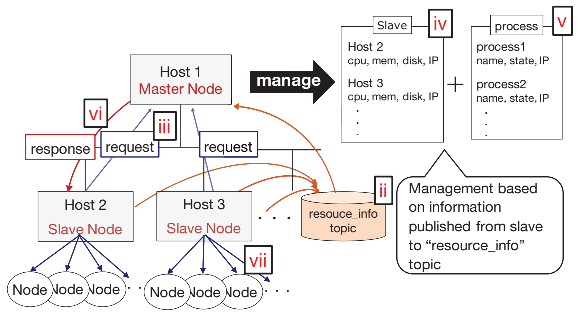

Fig. 3: Design of the ROS resource manager

because slaves provide resource information to the master, and the master has unilateral control of assigning processes using the slave's resource information. Therefore, it is optimal for constructing a resource manager herein. The design of the proposed resource manager is shown in Fig. 3, where topic communication is used for resource information, whereas service communication is used for processing assignments. Using topics to communicate resource information asynchronously from slave to master, it is possible to reduce the communication load with the master. Services are used for assigning processes because it is a synchronous communication method that ensures the reliable allocation of processes.

B. Resource manager construction

In this paper, we define the resources of the host machine as being the CPU, memory, and disk. As shown in Fig. 3, master-slave management is realized by launching the master node on one host machine and launching slave nodes on the other host machines. We refer to the host machine executing the master node as the master and any number of host machines executing a slave node as the slaves. The resource manager and process allocation associated with the proposed method are presented in the following steps corresponding to those shown in red in Fig. 3.

i. Set the upper limits of the utilization rates of the CPU, memory, and disk on each host machine to a slave node.
ii. Each slave host machine publishes the current resource utilization and the upper limits of the resource utilization rates to the "resource_info" topic.
iii. If the current CPU, memory, and disk usage rates do not exceed each of the resource limits, the slave node requests a processing assignment from the master node.
iv. The master node subscribes to the "resource_info" topic registers the host machine to the management list if it is a new host machine, and updates the resource information of the management list if it is an existing host machine.
v. The master node manages processes assigned to slaves via the first-in, first-out scheme.
vi. When a slave node requests a process allocation, processes are assigned to the slave with the highest resource margin determined by the host machine management list.
vii. Call processes (nodes) on the slave node.

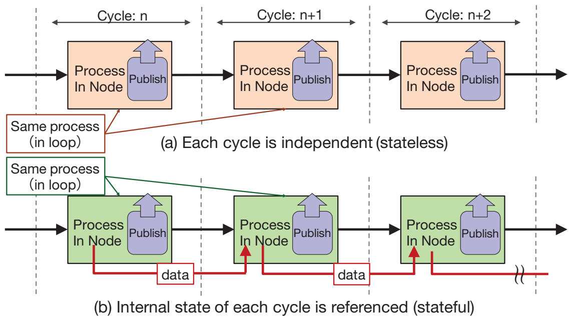
Cycle: n
Cycle: n+1
Cycle: n+2
Process In Node
Publish
Same process (in loop)
(a) Each cycle is independent (stateless)
data
(b) Internal state of each cycle is referenced (stateful)

Fig. 4: State of the ROS processes.

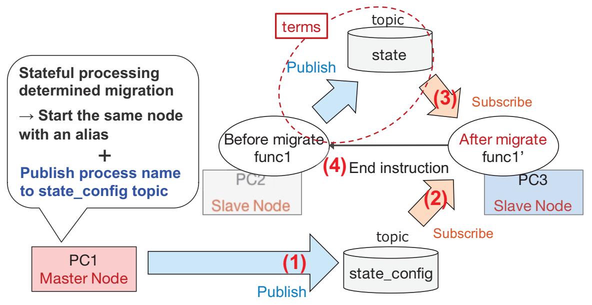

Fig. 5: Stateful node migration behavior.

With the above flow, each ROS node can be launched while managing the resources of each host machine constituting the distributed environment. The proposed resource manager allows the process imbalance between the successive host machines to be reduced. Additionally, it contributes to the prevention of failures resulting from resource shortages.

The details of the process allocation using the resource information are as follows: The slave resource margin is calculated using Eq. (1), with $memusage$, $cpuusage$, and $diskusage$ are the memory, CPU, and disk usage rates, respectively. Herein, the administrator determines the arbitrary weighting variables w_{mem}, w_{cpu}, and w_{disk}, which take values ranging from zero to one. Since a ROS node cannot determine the number of resources used until actual processing is performed, the slave node with the lowest score is regarded as the slave with the largest resource margin.

$$score = w_{mem} \times memusage + \\ w_{cpu} \times cpuusage + w_{disk} \times diskusage \qquad (1)$$

There are two reasons why each slave node determines whether the upper resource limit has been reached independently when publishing resource information (steps ⅱ and ⅱi). First, the allowable usage rates for individual slave nodes are assumed to be different. The second reason is to reduce the number of requests from slaves to the master and the calculations that compare the current values and the upper resource limits. With regard to scalability, e.g., increasing the number of host machines, reducing the calculations on the master node is important.

C. Process migration method between host machines

When using the resource manager described in Section III-B, the resource utilization rate can reach the preset upper limit of the slave in some cases. In such a case, we propose a migration method that reallocates the processes to another slave. The master can judge whether the processes allocated to a particular host have reached the given upper resource limit. Then, the processes assigned to a slave that has reached its upper resource limit can be migrated to another slave, thereby distributing the load. When the resource utilization of all host machines reaches the preset upper limit, the allocation of new processes is stopped and processes are not migrated; in this way, migration does not occur frequently.

The two patterns shown in Fig. 4 are used to migrate ROS nodes. First, a stateless case is shown in Fig. 4(a). In this case, the publishing/subscribing of the information of the node at a fixed cycle is unaffected by the previous cycle, examples being a node that continues to publish the same message at a fixed cycle or a node that delivers raw data obtained from a sensor. The stateful situation is shown in Fig. 4(b). In this case, the publishing/subscribing of information at a fixed cycle is influenced by the previous period. An example is a node that delivers a timestamp showing the time that the node started processing or the estimation of a vehicle's position in SLAM. We explain the method for process migration in stateless and stateful situations.

1) The stateless case: Nodes with the same name cannot be launched. If another node with the same name is activated, the previously operating node is terminated. Therefore, we can migrate stateless nodes between host machines by assigning the node with the same name to other host machines. The advantage of this method is that it is not necessary to rewrite nodes that exist in packages and applications.

2) The stateful case: All states are lost when the same name node is launched as a stateless case because stateful data depend on the previous cycle that are held and used by variables. Consequently, we propose a method with the following two constraints: (i) information regarding the state must be published in every cycle; (ⅱ) in the function called by the topic published by the master node, the variable with the published data must be initialized. These constraints must be satisfied by application developers. State data are often used by other nodes; therefore, publishing this information in every cycle in accordance with constraint (i) is not a burden for application developers.

Figure 5 shows the sequence of process migration. First, the master node launches the same node as a migration destination with an alias name (e.g., "original name" + "Unix time") and publishes the state_config topic as a trigger for node initialization, as shown in Fig. 5(1). The migration destination node subscribes the state_config topic published by the master node [Fig. 5(2)]. Then, a function for subscribing the information published by the migration source node is called. In the called function, the system initializes a stateful variable [Fig. 5(3)] with the published information and issues an instruction to terminate the migration source node [Fig. 5(4)]. As a result, it is possible to initialize variables according to the previous cycle information and migrate the process between host machines even for stateful nodes.

TABLE I: Evaluation environment for the operation of the resource manager.

OS	Ubuntu 16.04
Kernel	Linux kernel 4.4.0
Memory Size	16 GB
ROS version	ROS Kinetic Kame
Quantity (slave)	3 Units

TABLE II: Evaluation environment for the ROS scalability.

OS	Ubuntu 16.04
Kernel	Linux kernel 4.4.0
Memory size	4 GB
ROS version	ROS Kinetic Kame
Quantity (slave)	10 Units

IV. Evaluation

A. Evaluation environments

In the evaluation, the impact on the resource management of the CPU, memory, and disk usage rates calculated using Eq. (1) were $w_{mem} = 1$, $w_{cpu} = 0$, and $w_{disk} = 0$. In particular, the utilization rate of the memory affects the process allocation, making it easy to check the validity of the evaluation and the fact that memory use is an obstruction in real applications. Furthermore, the upper limit of the memory utilization in each host machine was set to 80%, a level at which the processes operate stably, and the operation cycle was set to 10 Hz, which is the standard ROS cycle. The operation is the same in the proposed mechanism even in an environment where the number of slaves is increased; however, it is difficult to verify the operation because the time-lapse diagram of the state of the resource becomes complicated. Therefore, we constructed the following two evaluation environments:

1) an environment for evaluating the operation of the proposed resource manager;
2) an environment for evaluating the scalability (i.e., the number of processes that can be processed) of a ROS using the proposed resource manager.

In the first evaluation environment, three host machines were prepared (their specifications are listed in Table I), one of which served as both the master node and the slave node, whereas the remaining two host machines functioned as slave nodes. In the second evaluation environment, the slave node was launched on each of the 10 host machines (their specifications are listed in Table II).

B. Behavior of the ROS resource manager

This section shows the operation of the resource manager based on the proposed method. The evaluation environment is summarized in Table I. The evaluation items are 1) the process allocation to a host machine with the highest resource margin and 2) process limitation of the resource manager.

1) Process allocation to a host machine with the highest resource margin: Processes are allocated by the resource manager to the slave with the highest resource margin. The memory utilization rate, which is the upper given resource limit in the evaluation, is limited to 80%. Figure 6 shows the elapsed time and the resource utilization rate when three

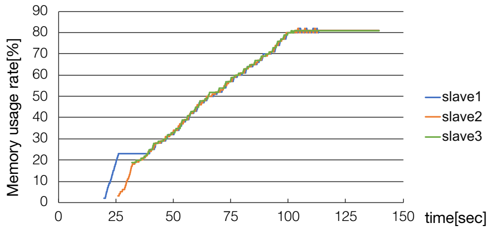

Fig. 6: Process allocation on the resource manager.

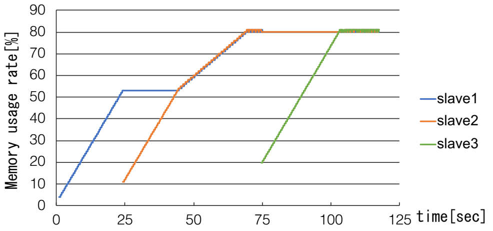

Fig. 7: Process allocation when slave is added halfway.

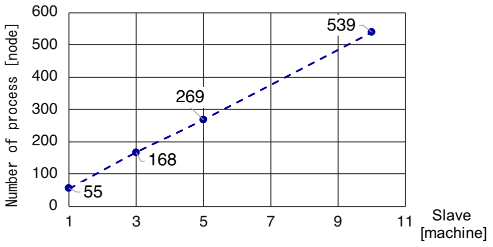

Fig. 8: The number of processes that can be allocated in accordance with an increase in the number of slave nodes.

slave nodes are launched nearly at the same time. Here, the processes allocated to each slave provide sufficient memory to use 8-80 MB per process up to the resource limit of each host machine. Figure 6 shows that each slave performs processes while maintaining the load balance and that the processes are allocated continuously until the upper limit of the set resource rate is reached. Figure 7 shows the time lapse and resource utilization rate when three slaves are launched with a time difference. In this evaluation, the process allocated to each slave provides sufficient memory to use 32 MB per process before reaching the upper resource limit of each host machine. Based on the addition of slaves at 25 and 75 s in Fig. 7, we were able to determine that processes were allocated to the slave with the highest resource margin.

2) Process limitation of the resource manager: Here, we consider the scalability when allocating processes to distributed environments using the resource manager. The evaluation environment is summarized in Table II. In a distributed environment, the scalability indicates that the number of processes that can be processed is as proportional as possible to the number of slaves. In addition, the master subscribes the resource information from the slave. The master node calculates the score of the slave and is responsible for process allocation. Therefore, because the processing load of the master node increases as the number of slaves increases, the processing limits of this resource manager is also evaluated.

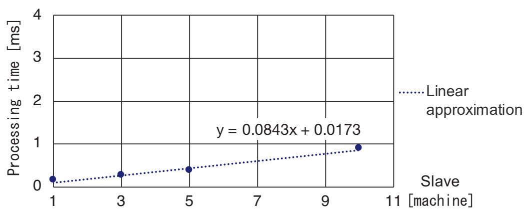

Fig. 9: Master processing time and approximate graph accompanying the slave increase.

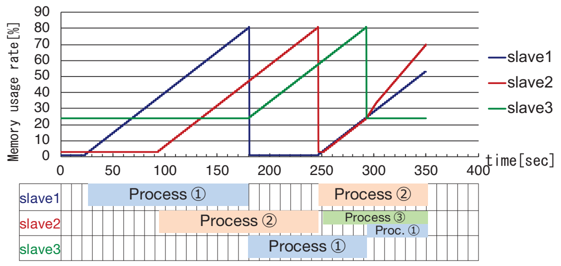

Fig. 10: Process migration when a given resource limit is reached.

Using the resource manager, we assigned a memory of 40 MB per process to each slave as the upper limit of each slave's resources. The number of processes that can be allocated with the increase in slaves is shown in Fig. 8. This figure shows that, as the number of slaves is increased, the number of processes that can be assigned increases proportionally. As a result, it was shown that the managed processing in distributed environments using the resource manager exhibits scalability. Figure 9 shows the processing time involved in one cycle of the master node as the number of slaves increased. The internal processing of the master node, which searches the slave with the minimum score, involves $O(n)$ calculations. Therefore, Fig. 9 plots the processing time of the master with an increase in the number of slaves and the linear approximation formula $y = 0.0843x + 0.0173$. This formula means that the processing time within 100 ms per cycle required by the resource manager is satisfied regardless of launching 1,000 slaves. However, because we cannot ensure that this number is sufficient for an accurate evaluation, in the future, a larger distributed environment must be constructed and investigated.

C. Process migration when the specified resource limit is reached

We show that process migration occurs when the upper limit of the given resource is reached. In this case, three processes constantly obtaining the memory were allocated sequentially. Figure 10 shows the elapsed time and resource utilization rate with three slaves activated. As shown in Fig. 10, the task of initiating process 1 was assigned to slave 1, which had the lowest resource utilization rate. Process 2 was assigned to slave 2 with a lapse of 100 s. After 180 s, in slave 1, because the memory utilization rate exceeded the set upper limit of 80%, process migration was performed. At this time, the slave chosen for the migration was the one with the highest resource margin at that time. In this case, it was confirmed that processes were assigned to slave 3 and that an appropriate operation was performed. With regard to process 2, Fig. 10 shows that processes will be migrated to the slave with the highest resource margin. After 300 s, because process 1 has migrated to slave 2, it can be seen that two processes are performed on slave 2 and that the inclination of the graph is large.

TABLE III: Execution time of SLAM application.

Environment	Execution time (min)
One host	109.5
10 hosts (ideal case)	2.5
10 hosts (with Resource manager)	3.6

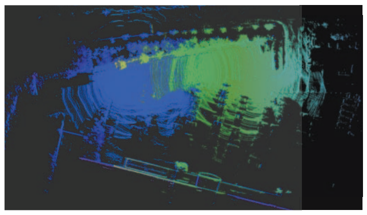

Fig. 11: 3D maps created by different host machines.

D. Evaluation using real applications

As an evaluation of real applications, we generated 3D point cloud maps using a SLAM application using the Normal Distributed Transform algorithm proposed by Magnusson [15], which requires large data processing. When processing SLAM, while estimating its own position, a map is generated by sequentially adding sensor data to its calculated position. Therefore, the self-position estimation result is a state in processing.

1) Scalable processing with SLAM: SLAM performs 3D map generation for a distance of 1 km. Table III shows the processing time for SLAM when performed by one host machine and when performed by 10 host machines with the same data divided into 10 pieces. The "10 hosts (ideal case)" entry in Table III is an ideal case in which processing is assigned manually to 10 host machines one each, and the "10 hosts (with Resource manager)" entry is the result of assigning the processes using a resource manager. When using 10 hosts compared to one host, the processing time becomes 1/10 or less because the amount of data per unit decreases and the memory-access efficiency improves. We also observed that the processing speed of the resource manager is slower than that of the ideal case. This is because when processing is performed with the resource manager, each process is allocated to the host with the highest resource margin. Therefore, it is not necessarily optimal. However, if the number of host machines is increased further, the resource manager can automatically allocate the processes and it is possible to handle the data realistically. Further, the resource manager manages the resources of each host machine and maintains the state; therefore, it is also superior with regard to fault tolerance.

2) Process migration in SLAM: We allocated the SLAM process so that migration occurs using the resource manager and evaluated the stateful processing migration. Figure 11 shows a superimposition of the 3D map (blue) created on the host machine before the handover and the 3D map (green) created on the destination host machine. As a result, each host machine received the point cloud data of the same vehicle. This demonstrates that the resource manager also works for real applications with stateful processing migration.

V. Related work

We discuss prior studies of continuous processing in ROS distributed environments. Lauer et al. [10] constructed a fault-tolerant processing mechanism using ROS internode communication. This study proposed a primary-backup replication system that can continue processing even if one host machine fails in a ROS distributed environment. Furthermore, connections between nodes are realized via a single ROS master on one host machine, which is a single point of failure in the current version of ROS. This problem could be solved by applying a library to check and restart the entire launched UNIX process using the DMTCP package [11], [16] in ROS. It is interesting that these approaches solve the lack of availability in ROS distributed environments. However, ROS2, the next generation of ROS, adopts DDS to solve the problem of availability by distributing the ROS master function. ROS2 makes it possible to guarantee the quality of service by adopting DDS [17]. However, these methods do not manage the resources used by each node running on the host machine. As a result, process imbalance can occur between the host machines in ROS distributed environments. In addition, because their system did not have the feature to scale-out, the system could not utilize the resources effectively.

We consider other resource management mechanisms for distributed environments. Volpe et al. [18] presented Coupled Layer Architecture for Robotic Autonomy (CLARAty). CLARAty is a robotic framework designed to improve the modularity of system software to combine autonomy and control interaction more strongly. CLARAty decomposes robotic software into two layers: a decision layer and a functional layer. The decision layer determines the activity and model and the function layer realizes the function for each object, such as the individual hardware. In this way, even in a distributed robotic environment, each resource is used in an optimal state for the activity model determined in the decision layer. However, it is difficult to separate this robot module system into two layers to use for large-scale data processing, such as that handled in a cloud because it is a hardware-oriented system. In addition, YARN is a resource management mechanism used in Hadoop [19], which is a scale-out framework. YARN performs resource management and task scheduling in a distributed environment adopting the master-slave method but is executed as a Java process, and the processing of the application itself also processes tasks on a Java virtual machine. Therefore, the processes that can be processed are limited to Java processes and YARN cannot be used as a ROS resource manager.

VI. Conclusions

This paper has been presented a master-slave resource manager for ROS distributed environments. In our proposed resource manager, the master monitors the utilization of the CPUs, memory, and disks on each slave and allocates processes to host machines with the highest resource margin. In addition, when resource utilization reaches a preset upper limit, the system migrates processes to other host machines to use the distributed resources efficiently. In the case of process migration under the constraint of publishing data with a state, we delivered a topic as a trigger to subscribe data on the master node. This enabled migration of processing that crossed the host machine even for stateful nodes. In the evaluation, we demonstrated that scalability can be realized in a ROS distributed environment by allocating the processes efficiently to each host machine. In the future, we will examine GPUs and efficient process allocation taking into account network resources that are not managed by the current resource manager. Furthermore, we will use more host machines for real applications such as generating 3D maps composed of point clouds.

References

[1] S. Kato, S. Tokunaga, Y. Maruyama *et al.*, "Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems," in *Proc. of ICCPS*. IEEE Press, 2018, pp. 287–296.

[2] Baidu, "Apollo," http://apollo.auto/, retrieved Sept. 2nd, 2018.

[3] E. Dean-Leon, K. Ramirez-Amaro, F. Bergner *et al.*, "Robotic technologies for fast deployment of industrial robot systems," in *Proc. of IECON*. IEEE, 2016, pp. 6900–6907.

[4] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, 2012.

[5] M. Quigley, K. Conley, B. Gerkey *et al.*, "ROS: an open-source Robot Operating System," in *Proc. of ICRA workshop on open source software*, vol. 3, 2009, p. 5.

[6] S. Cousins, B. Gerkey, K. Conley *et al.*, "Sharing Software with ROS [ROS Topics]," *IEEE Robotics Automation Magazine*, vol. 17, no. 2, pp. 12–14, 2010.

[7] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *Proc. of ICRA*. IEEE, 2011, pp. 1–4.

[8] T. Nägeli, L. Meier, A. Domahidi *et al.*, "Real-time Planning for Automated Multi-View Drone Cinematography," *ACM Transactions on Graphics*, vol. 36, no. 4, pp. 132:1–132:10, 2017.

[9] M. Nathan, S. Shaojie, M. Kartik *et al.*, "Collaborative mapping of an earthquake-damaged building via ground and aerial robots," *Journal of Field Robotics*, vol. 29, no. 5, pp. 832–841, 2012.

[10] M. Lauer, M. Amy, J.-C. Fabre *et al.*, "Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS," in *Proc. of HASE*. IEEE, 2016, pp. 94–101.

[11] T. Jain and G. Cooperman, "DMTCP: Fixing the Single Point of Failure of the ROS Master," https://roscon.ros.org/2017/presentations/ROSCon%202017%20DMTCP.pdf, ROScon 2017.

[12] G. Pardo-Castellote, "OMG Data-Distribution Service: architectural overview," in *Proc. of 23rd ICDCSW*. IEEE, 2003, pp. 200–206.

[13] Open Source Robotics Foundation, "ROS2," https://github.com/ros2/ros2/wiki, retrieved Sept. 2nd, 2018.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proc. of SoCC*. ACM, 2013, pp. 5:1–5:16.

[15] M. Magnusson, "The Three-Dimensional Normal-Distributions Transform: an Efficient Representation for Registration, Surface Analysis, and Loop Detection," *Orebro university*, 2009.

[16] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Proc. of IPDPS*. IEEE, 2009, pp. 1–12.

[17] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *Proc. of EMSOFT*. ACM, 2016, pp. 5:1–5:10.

[18] R. Volpe, I. Nesnas, T. Estlin *et al.*, "The CLARAty architecture for robotic autonomy," in *Proc. of AeroConf*. IEEE, 2001, pp. 121–132.

[19] The Apache Software Foundation, "Apache Hadoop," http://hadoop.apache.org/, retrieved Sept. 2nd, 2018.