Rorg: Service Robot Software Management with Linux Containers

Shengye Wang Xiao Liu Jishen Zhao Henrik I. Christensen*

***Abstract*— Scaling up the software system on service robots increases the maintenance burden of developers and the risk of resource contention of the computer embedded on robots. As a result, developers spend much time on configuring, deploying, and monitoring the robot software system; robots may utilize significant computer resources when all software processes are running. We present Rorg, a Linux container-based scheme to manage, schedule, and monitor software components on service robots. Although Linux containers are already widely-used in cloud environments, this technique is challenging to efficiently adopt in service robot systems due to multi-tasking, resource constraints and performance requirements. To pave the way of Linux containers on service robots in an efficient manner, we present a programmable container management interface and a resource time-sharing mechanism incorporated with the Robot Operating System (ROS). Rorg allows developers to pack software into self-contained images and runs them in isolated environments using Linux containers; it also allows the robot to turn on and off software components on demand to avoid resource contention. We evaluate Rorg with a long-term autonomous tour guide robot: It manages 41 software components on the robot and relieved our maintenance burden, and it also reduces CPU load by 45.5% and memory usage by 16.5% on average.**

I. Introduction

Robotic research and application are advancing at a high pace in recent years, and service robots have started to enter the consumer market and assist human at home and offices [1]. Whereas deploying service robots is becoming easier, the robot systems themselves are becoming more complex [2]. From our previous experience, a tour guide robot for human-robot interaction (HRI) research in a university office building consists of about 65 software programs for face recognition, voice recognition, navigation, and various other tasks [3]; autonomous driving vehicles have even more components for localization, pedestrian detection, mission planning, motion planning, and so on [4]. As a robot system evolves and expands, the complexity of the software components is becoming a more pressing issue that in turn, challenges scaling up.

Developing software for robot applications has two primary challenges: maintenance burdens and resource limitations. First, when the number of software components increases, organizing, deploying, and monitoring them becomes tedious and error-prone [5], [6], [7]. Second, computationally intensive programs use much of the computing resources in the on-board computer [8], [9]. They challenge scaling up since a service robot can only carry a computer with moderate processing power and memory capacity due to power and weight considerations. Efforts have been put into tackling these issues, but the infrastructure side of robot systems has only received limited attention.

*Department of Computer Science and Engineering, University of California, San Diego. La Jolla, CA 92093, USA. {shengye, x1liu, jzhao, hichristensen}@ucsd.edu

Fig. 1. TritonBot [3]: a Rorg-powered long-term autonomous tour guide robot that chats with people and navigates to introduce places of interest in a university office building. We use Rorg to manage the 41 software components in TritonBot. Rorg only runs the components that are needed and shuts them down when they are idle to save computing resources.

Recently, the Linux container has become a popular tool to deploy software in data centers and the cloud [10], [11], [12]. This technique allows developers to pack software and dependencies into self-contained images, deploy them onto different targets with minimal configuration, and isolate them from the host system to varying degrees. This technique also opens an opportunity to mitigate the maintenance burden and resource limitation in service robots (Section II). However, few research and commercial service robots currently benefit from Linux containers due to three primary reasons:

- Most of the current research and commercial service robots use certain middleware or framework [13], [14], [15] to facilitate loosely-coupled architecture design, and among them ROS [13] is the most popular representative. While ROS-based software can run inside containers in theory, there is limited experience to containerize the robotic software.
- Service robots have numerous software components with complex relationships, and not all of them are in use at the same time. Although existing tools can leverage Linux containers to provide interfaces for one component to start/stop/monitor another, few take the runtime pattern of programs into account.
- Few Linux container orchestration tools are designed for environments with tight overall resource budget, as they target for data centers and cloud platforms [12] where almost unlimited computing resources are available.

Our goal is to enable efficient software organization

and scheduling on autonomous service robots in a high-performance and scalable manner. We propose Rorg, a toolkit that leverages Linux containers to manage and schedule software on service robot. Rorg makes three key contributions:

- Rorg adopts a Linux container engine, Docker [10] to run robot software in individual containers. Rorg works with ROS [13], the de facto standard robotic middleware. It provides default configurations to run ROS software without modification, and we provide an example setup of a fully-functional tour guide robot application with Rorg.
- Rorg organizes the robot software into multiple Linux containers and models the static and dynamic relationships between them. Rorg provides an interface to create, query, update, delete, start, stop, and restart these containers manually or programmatically.
- Rorg allows the software components to time-share computing resources: It pauses or shuts down inactive services to save computing resources but reactivates them when they are required for an upcoming task. Rorg monitors the system and keeps the resources utilization at a reasonable level.

We evaluate Rorg on a long-term autonomy tour guide robot, TritonBot (Figure 1). Rorg and its early prototype helped us to keep the robot working for over six months. The tour guide robot went through 126 software version updates and 71 configuration changes. Rorg now manages 41 services (containers) on the robot. The robot consumes 89.4% of CPU time and 3.41 GB of memory on average without Rorg. With Rorg, the average CPU drops to 48.7% and the memory usage is 2.85 GB. Although web service and data center applications have wildly adopted container technology, Rorg addresses the specific challenge on service robots: multi-purpose systems with limited computing resources but have high performance/responsiveness requirements.

The rest of this paper is organized as following: Section II discusses related work and the unique challenges of service robot software management. Section III presents the design of Rorg and introduces its features to ease software management and avoid resource contention. Section IV evaluates Rorg with a real tour guide service robot and measures its performance, and finally Section V concludes the paper.

II. Background and Related Work

Linux containers (or operating-system-level virtualization in general) have received much attention in data centers and cloud applications recently [16]. Relying on operating system kernel features, Linux containers provide isolated runtime environment with minimal performance overhead. They also provide a convenient approach to build, deploy, and run applications on different machines. Unlike hypervisors or virtual machines that run fully virtualized kernels, Linux container leverages the kernel of the host and thus is much lighter weighted [17]. Popular Linux container engines include Docker [10], LXD [18], and others. Despite different branding and user interface, they all exploit the same underlying Linux kernel features such as namespaces and control groups, and thus offer similar performance.

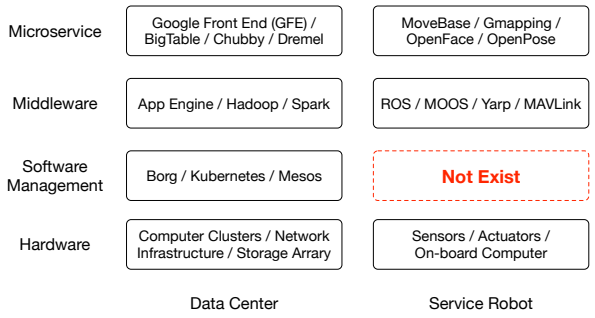

Fig. 2. Robot systems are similar to data centers. Each hierarchy in a service robot can find its counterparts in a data center, except for the "software management system." We built Rorg to fill in the gap and address the unique challenges of software management in service robots where resources are sacred.

Prior research on applying Linux containers to robots. Previous research applied Linux containers to robotics to some extent. White et al. use Linux containers to run ROS with Docker [19], but their examples do not illustrate actual challenges on building and maintaining a moderate-scale service robot system. Mabry et al. exploit Docker to deploy software on a maritime robot [20], but their approach only applies to application-specific robot and does not scale up to fit the more complex software system of service robots. Avalon [21] is a crowd-robot scheduler that distributes tasks to multiple robots; it also leverages Linux containers to encapsulate the applications, but it does not address challenges on a standalone service robot. Cognitive Interaction Toolkit (CITk) [22] adopts Linux containers to construct and run robot simulation experiments; a later work RoboBench [23] is a benchmark suite based on CITk that reproduces robotic simulation on workstations. SwarmRob [24] is a toolkit to share experimental heterogeneous robots using Linux containers. However, none of CITk, RoboBench, or SwarmRob was tested on physical robots.

Difference between service robots and data center applications where Linux containers are used. Robot software systems are similar to data center applications in some aspects. Modern applications in data centers are often built with the "microservices" pattern: individual microservices communicate and cooperate with each other to deliver greater functionality [25] — the same pattern also exists in ROS-based robot system design. Besides, other hierarchies in a robotic system can also find their counterparts in data center applications, as shown in Figure 2. However, most service robots do not have a "management system" counterpart to data centers, and that is where Rorg comes in.

In data centers, the main challenge for a software management system is redundancy. At Google, Borg [26] runs two extra instances of each microservice in their planet-scale computer system to tolerate the failure of one instance while updating another instance [25]; it distributes the instances across different physical locations to increase reliability. Similarly, Kubernetes [12] and Mesos [27] are popular open-source software management systems that combine Linux containers with sophisticated scheduling to provide load-balancing, to avoid single point failures, and to scale-up when

needed. However, the challenges of software management in service robots are distinct to that in data centers, as discussed in Section I.

In summary, no existing solutions address the unique challenges of software management on service robots — multi-purpose systems with limited computing resources yet have high performance/responsiveness requirements.

III. Rorg Design

To address the aforementioned challenges, we propose Rorg, a software manager toolkit for service robots. In essence, Rorg is a set of programs that receives requests from developers or programs, and creates, starts, pauses, stops, and removes software components on a service robot at an appropriate time. Rorg consists of three design principles: First, Rorg is effortless to use; it containerizes robotic applications (ROS-based in particular) with minimal configuration. Second, Rorg is scalable; it targets managing moderate and large-scale robot applications up to hundreds of programs or ROS nodes. Third, Rorg avoids the risk of computer resources contention in the robot; it eliminates unnecessary computation and improves robot responsiveness.

A. Linux Containers for Robotic Applications

Linux container interface for robotic applications. Rorg leverages Docker [10] as its underlying backend to containerize robotic software, but it provides default configurations to run ROS-based software with minimal configuration since currently ROS is the most popular robotic middleware. Because ROS applications are not designed to cross network address translation (NAT) devices, Rorg by default uses the host computer's network stack for the containers to exclude default communication barriers that come with Docker and extra performance overhead in multiple networking stacks. In addition, Rorg provides a "driver" option to configure microservices that talks to sensors/actuators with higher privilege to access host peripherals. Rorg monitors these programs for the unexpected restart, and it asks for extra confirmation when restarting these programs manually. Last but not least, Rorg still opens all the lower-level Docker parameters to the users, but Rorg alerts users to potential configuration error. In short, Rorg makes Linux container easier to use for robotic applications without sacrificing its original functionality.

Efficient container image hierarchies. Rorg enables efficient deep-hierarchy container image building. Docker allows the user to pack a program along with its dependencies into a standalone "image"; these images can form a hierarchy to save build time or disk space when they share common parts. Figure 3 shows part of the images hierarchy in a service robot. A deep hierarchy allows each application image to use an appropriate base image to maximize libraries reuse and minimize the chance of conflicts, but it also leads to a challenge: the developer needs to build the images in a correct order for the end image to reflect the updates in the hierarchy. Rorg provides a script to automate the task: it sorts all of the images in a topological order of the dependency graph, pulls external dependencies, and then builds dependent images before the children images. Therefore, Rorg retains Docker's optimization such as caching while building the images, but it builds the images correctly without a larger infrastructure such as a continuous integration system.

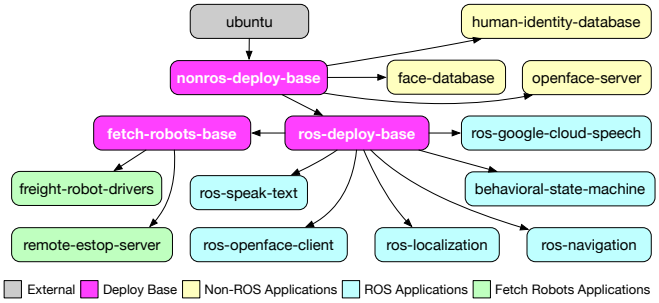

Fig. 3. Container images (and their hierarchy) that support a robot receptionist and tour guide [3]. By carefully arranging the hierarchy, each application can use a most appropriate "base image" and save build time and disk space without creating conflicts. Rorg provides tools that help the developer to build the images with a deep hierarchy correctly.

Linux containers for software development. Rorg leverages Linux containers to simplify robotic software development. Rorg comes with a script that prepares a seamless development environment inside a container, which is based on the same base image for deployment. The unified environment eliminates the inconsistency between development and deployment, which helps the developer to prepare for deployment at the very beginning. Since the development environment is encapsulated inside a container, the developer can create a fresh environment with minimal effort in case the development environment is contaminated.

Configuration history tracking. Rorg keeps the full history of an application for later review or postmortem analysis. It records the full log of the changes to Docker container in machine-readable "protobuf" [28] format. Also, Rorg leverages Docker's "mount" feature to map a directory on the host machine to a container, so that the developers can store runtime configurations like "`roslaunch`" files to a version control system (VCS) and attach them to the container. The history in VCS combined with the Rorg log allows the developers to recover the state of a robot to any previous checkpoint. As an example, we provide our TritonBot tour guide system along with Rorg to demonstrate these practices.

Configuring Rorg. The benefits of Rorg are not free — the developer needs to write extra code to configure Rorg. Since Rorg runs all programs inside Docker containers, deploying a robotic program with Rorg requires creating a Docker image and defining runtime parameters. Seemingly an extra effort, the image blueprint (`Dockerfile`) and the runtime configuration actually serve as a document to reproduce the execution environment, which helps organizing robotic software from another perspective; many readily available ROS images and examples further make this process easier [19].

The above Rorg features exploit most of Docker's potentials to build, ship, and deploy ROS-based robotic software; we plan to extend Rorg support to other robotic middleware in the future. These features keep indivudal robotic software

components organized and lighten developer's burden. The next section will discuss Rorg's effort to organize robotic software system in a whole and avoid resource contention.

B. Scalable Robotic Software Organization

Basic element in Rorg. The basic element in Rorg is a *service*. Usually, a service represents one container instance: for example, each of the localization, navigation, and face recognition software is a service. The "service" concept is consistent with "microservice" in data center applications, and it is similar to ROS "nodes" or Linux processes in terms of granularity. Rorg does use non-regular service to simplify its semantics (for example, `developer` is a meta-service that represents a developer's actions, which will be discussed later in this section), but regular services are created with Docker images and runtime configurations. Rorg provides interfaces to create, query, update, and remove a service programmatically or through a command-line interface.

The relationship between the Rorg elements. Although each service differs from each other in terms of the role in the system, Rorg sees them as identical in orchestration. A service may "request" another: a *request* is a relationship between services — the requester service will use the requested services until it releases the request. Rorg only keeps requested services alive and ceases services that are not requested by any other services. For example, when the robot needs to talk, a *behavioral* service can request the *speak-text* service before invoking voice-synthesize. For an active service with implicit dependencies, Rorg automatically requests the dependent services. For example, a *navigation* service must always request *localization* service because the robot cannot navigate correctly without awareness of its position. A newly created service is not active until another service "requests" it. When terminating a service, Rorg also automatically releases all its owned requests to prevent "requests leak."

C. Time-sharing Computing Resources

Rorg avoids resources contention on the on-robot computer by time-sharing computing resources. Because the robot carries limited computing power, the resource-contented computer should allocate the appropriate amount of resources on programs' demands. With the robotic software informing Rorg of the components usage, Rorg enables the required services and ceases or pauses inactive services, and thus reduces resources usage.

Figure 4 demonstrates a simplified Rorg-powered tour guide robot example that detects human faces, chats with people, and offers tours to the visitors. The robot takes five consecutive actions in its work cycle; Rorg manages services accordingly to avoid resources contention. ① In the beginning, the human developer requests the *behavioral* service that controls the overall behavior of the robot through a command-line program. The requester is set to a *developer* meta-service that is always assumed active. ② Then the robot becomes autonomous, and *behavioral* requests *face-recognition* to detect human faces. ③ When the robot sees

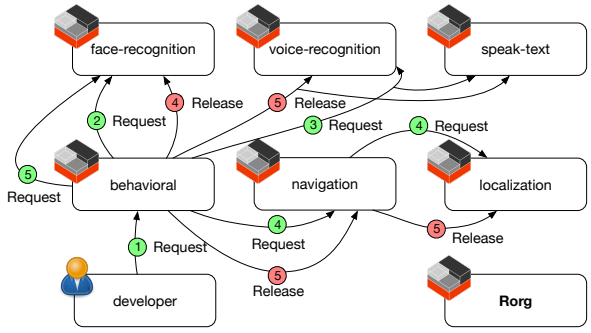

Fig. 4. Services in a receptionist and tour guide robot [3]. Each service is a Linux container; Rorg itself also runs inside a container, but it is not a service. The services contact Rorg to start or stop their peers: ① The developer makes the robot autonomous. ② The robot starts to detect visitors. ③ The robot chats with a visitor. ④ The robot moves around to guide a tour. ⑤ The robot returns to its standby state.

a visitor, *behavioral* requests *voice-recognition* and *speak-text* to chat with the visitor. ④ When the visitor decides to take a tour with the robot, the *behavioral* requests *navigation*, which implicitly requests *localization* (for robot pose estimation) to move the robot around. The *behavioral* also releases its previous request to *face-recognition*, as the robot does not detect human face during the tour. ⑤ After the tour, *behavioral* releases all of the services but requests *face-recognition* to return to the initial state. When *navigation* becomes inactive, Rorg automatically releases *localization* as it was only requested by *navigation*. Note *localization*, *navigation*, and *face-recognition* are all computationally intensive. In such setup, the robot does not waste CPU time on *face-recognition* during navigation, nor does it assign any resources for *localization* during the waiting state.

Programming for time-sharing. Using Rorg for time-sharing computing resources is easy in general, but it is complicated for certain services. For example, the developer can specify an implicit dependency between *navigation* and *localization* when creating *navigation* service, thus *localization* will always start when *navigation* becomes active. Rorg provides several interfaces and client libraries, including ROS-service interfaces, general remote procedure call (RPC) interfaces [29], Python, and C++ libraries to help developers to write code to send/cancel Rorg requests for services with dynamic behavior (e.g., *behavioral* service). Besides, we are investigating more automated methods to make Rorg more programmer-friendly.

Alternatives for avoiding resource contention. It is worth pointing out Rorg's time-sharing method is not the only way to reduce resource usage on service robots. A common option is to offload computation, but it is limited by latency constraints and privacy concerns on service robots. Another option is to start and stop individual Linux processes, but Linux containers are much cleaner (e.g. the developers are free from accidentally leaving orphan processes running) and provide more functionality (e.g. pausing a process without sending `SIGSTOP`). A third option is to design an event-driven system architecture — it can avoid unnecessary computation without shutting down any components — but such option usually involves a total system overhaul; besides, by

TABLE I

GENERAL TRITONBOT MAINTENANCE WORKFLOW WITH AND WITHOUT RORG.

Maintenance Task	Without Rorg	With Rorg
Deploy a new software component.	Copy source files to the robot; install the dependencies; compile the software; fix potential dependencies errors; write a startup script.	Write and test a `Dockerfile`; build and upload the container image to a "Docker registry"; use a Rorg command-line tool to create a new service.
Update a software component.	Remove original software and residuals but keep configuration and reusable data files; install the new version; update configuration files.	Rebuild and update the original `Dockerfile`; run a Rorg command-line tool to refresh the update.
Rollback a software component.	Remove original software and residuals; reinstall the old version; rollback configuration files.	Rollback configuration files; run a Rorg command-line tool to rollback.
Update software configuration.	Locate and update the configuration in scattered places; kill the program process tree; restart the program.	Update the configuration in a unified location; run a Rorg command-line tool to restart the service.
Develop software.	Install the same libraries and tools on the robot to the developer workstation; clean up or even reinstall the system in case of library contamination; before deployment, fix the inconsistency between deployment and development environments.	Use the same container image (with libraries and tools) to develop on workstations or robots; recreate the container with Rorg to restart from fresh; not to worry about inconsistency between deployment and development environments.

often ceasing services, Rorg provides an opportunity for them to restart refresh frequently, as Google points out "a slowly crash looping task is usually preferable to a task that hasn't been restarted at all [25]."

IV. EVALUATION

We test Rorg using our TritonBot system, a real long-term autonomous service robot [3]. This section evaluates Rorg from two aspects: First, to show Rorg's effectiveness on software maintenance, we compare our daily workflow before and after introducing Rorg to manage the software in TritonBot. Second, to show how Rorg reduces the risk of resource contention, we compare the CPU and memory usage with and without Rorg.

A. Experimental Setup

Rorg targets at service robots that perform different tasks at different times. In previous work we built TritonBot to serve as building receptionist and a tour guide robot [3]. TritonBot stands to face the building entrance and continuously detects faces with its camera. When TritonBot sees a visitor, it will greet the person by name if it could recognize the face, or it will ask the visitor's name and associate the name with the face. TritonBot also offers trivia questions and navigates with the visitors and introduces places of interest in the building; it uses a leg tracker to make sure the visitor is following it. The behavior of TritonBot is controlled by a finite state machine.

TritonBot is built on a commercial mobile robot platform "Freight Robot" [30]. It senses and interacts with the environment using a camera, a directional microphone, a loudspeaker, three laser scanners, and a mobile platform with two differential-drive wheels. The core of the platform is a computer composed by an Intel i5-4570S CPU (4 cores, 4 threads) operating at 3.20GHz, 16 GB memory, and 1 TB solid-state storage. Seemingly outdated, the computer was a medium-configuration when the two-year-old robot was manufactured. We believe TritonBot represents service robots in the middle of their lifecycles. TritonBot's software system consists of 65 ROS programs, and it heavily leverages open-source and third-party software to support its tasks. For example, Openface [31] pipelines human faces detection and the face similarity calculation; an open-source leg tracker [32] tracks people around the robot; Cartographer [33] provides simultaneous localization and mapping (SLAM) for the robot, and the ROS navigation stack "move base" [34] navigates the robot around. TritonBot also uses cloud service like Google Cloud Speech API [35] to transcribe speech audio to text.

B. Managing Software System

The complexity of TritonBot system puts a lot of maintenance burden on the developers. Many of these software components depend on different libraries (the "dependency hell"), and there are many runtime parameters, configurations, and other supporting files associated with each of these components. As the robot system scales up, keeping the system running becomes a tedious and error-prone task.

Table I compares our workflows of general maintenance tasks with and without Rorg. With Rorg, we manage our robotic software in a much cleaner and organized manner. Since the maintenance effort is a subjective concept and hard to quantify, we leave the decision of whether Rorg helps the developers in operating and evolving the system to the readers.

At this time, TritonBot is running 65 ROS nodes as 41 Rorg services. We group some tightly-coupled ROS nodes together as a single Rorg services (for example, the cartographer node and the accompanying occupancy-grid-map converter). In the past year, we pushed 126 software version updates and 71 configuration changes to TritonBot with the help of Rorg.

C. Avoiding Resources Contention

Many of the programs in TritonBot are resources intensive. When the robot is operating without Rorg, the average CPU usage is often around 90%, the memory utilization occasionally reaches 100% due to a potential memory leak. Therefore, TritonBot is on the borderline of resource contention where insufficient performance slows down the robot. As a result, the response latency of the robot is creating an unsatisfactory user experience — the robot only moves a few seconds after it says "please follow me."

We evaluate the decrease in resource usage with Rorg using three setups: simulation, emulation, and deployment.

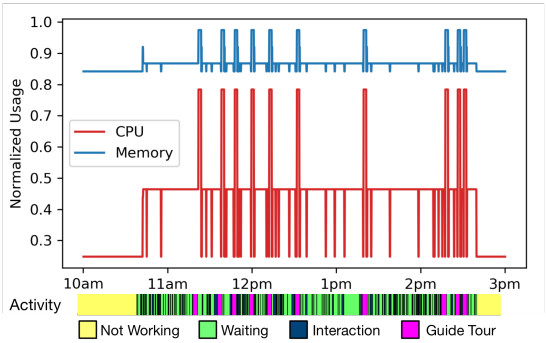

Fig. 5. Rorg simulation result of TritonBot using a trace collected on February 6, 2018. We normalize the CPU and memory usage to a baseline where all the components are always active. Only calculating active time, Rorg reduces CPU usage by 52.6% and memory usage by 12.5%.

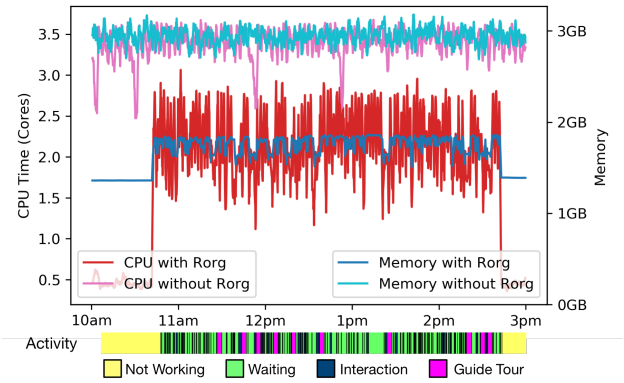

Fig. 6. Rorg emulation result of TritonBot. The activity of the robot at every moment is shown along with the CPU/memory usage. Only taking active time into account, Rorg brings 37.2% reduction to CPU usage and 40.2% reduction to memory usage.

Simulation. In the simulation, we feed the state machine trace (state transition log) to a simulator and assign empirical CPU and memory usage (medium number collected from a month-long deployment) to each of the services. The simulation experiment is fast and the result is always consistent, but it does not reflect the dynamic resources consumption by a program. We run the state machine trace collected on TritonBot back on February 6, 2018, a typical day for TritonBot [3]. The robot was deployed from 10:40 am to 2:40 pm that day, worked for four hours in total. It greeted the visitors 188 times and guided ten tours. As shown in Figure 5, the simulation result indicates 52.6% reduction to CPU usage and 12.5% reduction to memory usage by introducing Rorg to TritonBot.

Emulation. In the emulation, we run the actual components on a workstation with the same specs as the robot embedded computer, but we play back sensor data and discard the control commands sent to the actuators. The emulation experiment reflects the dynamic resource allocation of the programs, and it does not require TritonBot to move when we are profiling the system. The baseline system has all of the Rorg services running, but the system with Rorg only runs the services that are required at the emulated moment. Figure 6 shows the results from emulation, which reflects the fluctuation in CPU and memory usage when a service is starting or stopping. Rorg reduced CPU usage by 37.2% and memory usage by 40.3% in average in the emulation on average.

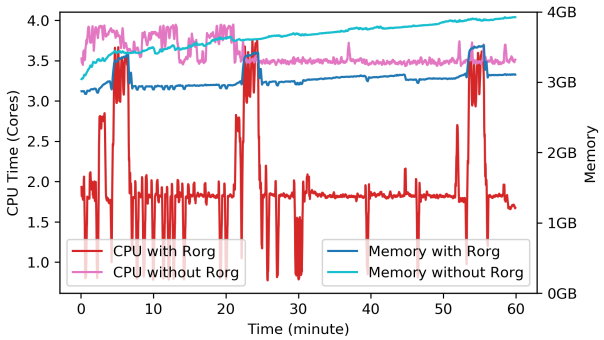

Fig. 7. Rorg deployment result of TritonBot. We deployed the robot for one hour and collected performance data. Although the events do not align in the baseline and in the Rorg experiments, the robot gave three tours and greeted people 30 times in both scenarios. With Rorg there is 45.5% reduction in CPU usage and 16.5% reduction in memory usage on average.

Deployment. We also deploy Rorg on the real TritonBot platform to observe its performance. Reduction of resource usage in deployment is the golden standard to evaluate Rorg's performance. However, due to the dynamic nature of the environment, we can only guarantee similar experimental conditions in a relatively short period. Figure 7 shows the variation of resources usage in a 60 minutes TritonBot deployment. We had the robot guided three tours and greeted people 30 times during the experiment. Rorg reduced CPU usage by 45.5% and memory usage by 16.5% during the deployment on average.

The CPU usage improvements are consistent in three experiments, but the memory usage differs. We found that the simulation and emulation correctly reflect CPU usage by switching on and off CPU-intensive tasks like face recognition and navigation, but the memory usage is less accurate because of difference of sensor/actuator drivers and data playback. Nevertheless, simulation provides a fast way to evaluate the theoretical benefit of Rorg, and emulation provides a fair comparison between the baseline and the system with Rorg.

V. Conclusion

Linux containers are widely used in data center and cloud platforms to help developers to deploy their systems. This paper presents Rorg, a tool for lifecycle/resource management on service robots. Rorg uses Linux containers to ease developer's effort to orchestrate microservices on service robots, and it enables efficient time-sharing resources to avoid resources contention. We tested Rorg using a service robot application — TritonBot [3], a receptionist and a tour guide robot. Experimental results by simulation, emulation, and deployment show that Rorg reduces 45.5% CPU and 16.5% memory usage. We release Rorg as an open-source software (available at `https://github.com/CogRob/Rorg`). We also provide configurations to build, deploy, and run TritonBot as an example of using Rorg.

Linux containers have the potential to play a more critical role in service robots. Our future goal is to apply Linux containers and other system engineering methods to create a more reliable and scalable robot system infrastructure and bridge the gap between robots in research and in real life.

REFERENCES

[1] H. I. Christensen, Ed., *A Roadmap for US Robotics: From Internet to Robotics*, Nov. 2016.

[2] Y. M. Youssef and D. Ota, "A general approach to health monitoring & fault diagnosis of unmanned ground vehicles," in *2018 International Conference on Military Communications and Information Systems (ICMCIS)*. IEEE, 2018.

[3] S. Wang and H. I. Christensen, "Tritonbot: First lessons learned from deployment of a long-term autonomy tour guide robot," in *Proceedings of the 2018 IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, August 2018, pp. 158–165.

[4] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, Nov 2015.

[5] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The brics component model: a model-based development paradigm for complex robotics software systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1758–1764.

[6] N. Hochgeschwender, S. Schneider, H. Voos, H. Bruyninckx, and G. K. Kraetzschmar, "Graph-based software knowledge: Storage and semantic querying of domain models for run-time adaptation," in *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Dec 2016, pp. 83–90.

[7] N. Hochgeschwender, G. Biggs, and H. Voos, "A reference architecture for deploying component-based robot software and comparison with existing tools," in *2018 Second IEEE International Conference on Robotic Computing (IRC)*, Jan 2018, pp. 121–128.

[8] M. Foughali, B. Berthomieu, S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet, "Formal verification of complex robotic systems on resource-constrained platforms," in *FormaliSE: 6th International Conference on Formal Methods in Software Engineering*, 2018.

[9] J. Wienke and S. Wrede, "Autonomous fault detection for performance bugs in component-based robotic systems," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2016, pp. 3291–3297.

[10] Docker Inc. (2018, Nov.) Docker is an open platform to build, ship and run distributed applications anywhere. [Online]. Available: https://www.docker.com

[11] Canonical Ltd. (2018) Snapcraft: Snaps are universal linux packages. [Online]. Available: https://snapcraft.io/

[12] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016.

[13] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[14] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the orocos project," in *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, vol. 2. IEEE, 2003, pp. 2766–2771.

[15] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "Genom3: Building middleware-independent robotic components," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 2010, pp. 4627–4632.

[16] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[17] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172.

[18] Canonical Ltd. (2018, Nov.) What's lxd? [Online]. Available: https://linuxcontainers.org/lxd

[19] R. White and H. Christensen, "Ros and docker," in *Robot Operating System (ROS): The Complete Reference (Volume 2)*, A. Koubaa, Ed. Springer International Publishing, 2017, pp. 285–307.

[20] R. Mabry, J. Ardonne, J. N. Weaver, D. Lucas, and M. J. Bays, "Maritime autonomy in a box: Building a quickly-deployable autonomy solution using the docker container environment," in *OCEANS 2016 MTS/IEEE Monterey*, Sept 2016, pp. 1–6.

[21] Y. Xu, Z. Yan, S. Wang, C. Yang, Q. Xiao, and Y. Bao, "Avalon: Building an operating system for robotcenter," *CoRR*, vol. abs/1805.00745, 2018. [Online]. Available: http://arxiv.org/abs/1805.00745

[22] F. Lier, J. Wienke, A. Nordmann, S. Wachsmuth, and S. Wrede, "The cognitive interaction toolkit–improving reproducibility of robotic systems experiments," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 400–411.

[23] J. Weisz, Y. Huang, F. Lier, S. Sethumadhavan, and P. Allen, "Robobench: Towards sustainable robotics system benchmarking," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016, pp. 3383–3389.

[24] A. Pörtner, M. Hoffmann, and M. König, "Swarmrob: A toolkit for reproducibility and sharing of experimental artifacts in robotics research," *arXiv preprint arXiv:1801.04199*, 2018.

[25] B. Beyer, C. Jones, J. Petoff, and N. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated, 2016.

[26] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[27] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[28] Google Inc. (2018, Apr.) Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. [Online]. Available: https://developers.google.com/protocol-buffers/

[29] Google LLC. (2018, Nov.) grpc: A high performance, open-source universal rpc framework. [Online]. Available: https://grpc.io/

[30] M. Wise, M. Ferguson, D. King, E. Diehr, and D. Dymesich, "Fetch & freight: Standard platforms for service robot applications," in *Workshop on Autonomous Mobile Service Robots, International Joint Conference on Artificial Intelligence*, July 2016.

[31] B. Amos, B. Ludwiczuk, and M. Satyanarayanan, "Openface: A general-purpose face recognition library with mobile applications," CMU-CS-16-118, CMU School of Computer Science, Tech. Rep., 2016.

[32] A. Leigh, J. Pineau, N. Olmedo, and H. Zhang, "Person tracking and following with 2d laser scanners," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 726–733.

[33] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d lidar slam," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1271–1278.

[34] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 300–307.

[35] Google LLC. (2017, May) Cloud speech api – speech to text conversion powered by machine learning. [Online]. Available: https://cloud.google.com/speech