Effective Crash Recovery of Robot Software Programs in ROS

Yong-Hao Zou and Jia-Ju Bai

***Abstract*— Modern robot systems use various software programs to autonomously perform different kinds of tasks. However, due to the risks of possible faults and errors, a robotic software program can inevitably crash in some cases, causing that the robot system fails to perform the current task. Thus, for robustness, the crashed program should be correctly recovered to continue the failed task. For this purpose, ROS provides a default *restart* method to automatically restart crashed programs. However, our case studies of typical ROS programs show that the restart method can perform incorrect crash recovery, and it can even cause the robot to perform dangerous behaviors, because this method loses the program's important data that was stored before the crash and is used after recovery. To solve this problem, we develop a practical approach named RORY, to perform effective crash recovery of robot software programs in ROS. RORY uses a hybrid checkpoint-replay method, and it is generic to different ROS programs by considering ROS properties. We evaluate RORY on 6 common ROS programs, and show that RORY performs correct crash recovery in both virtual and realistic environments with modest overhead. The comparison experiments indicate that RORY outperforms the restart, checkpoint-alone and replay-alone methods.**

I. Introduction

Nowadays robot systems have been widely used in aviation, manufacturing, transportation, biology and many other areas. To enhance generality and scalability, a modern robot system uses various software programs to autonomously perform different kinds of tasks, such as map building, human-robot interaction and navigation. However, due to the risks of possible faults and errors (such as transient faults, configuration errors and code bugs), a robotic software program can inevitably crash in some cases, causing that the robot fails to perform the current task. This problem is especially serious for the robot systems working in unmanned environments (such as forests and deserts), because the developers cannot conveniently perform manual recovery. Thus, for robustness, the robot system should automatically and correctly recover the crashed software program to continue the failed task.

To this end, ROS [1] provides a default *restart* method [2] to automatically restart crashed programs that run as ROS nodes. This restart method has been widely used to perform crash recovery of common ROS programs. However, our case studies of two typical ROS programs in Section II-C show that, the ROS restart method can perform incorrect crash recovery, and it can even cause the robot to perform dangerous behaviors, as it loses the program's important data (such as the already built map) that was stored before the crash and is used after recovery. Implementing a customized recovery method for a specific ROS program is feasible, but this method is hardly generic to other ROS programs.

To solve this problem, our basic idea is to introduce two classical recovery methods *checkpoint* and *replay* to recover ROS programs. Achieving this idea has two key challenges:

C1) The checkpoint and replay methods both have their own limitations. Specifically, the effectiveness of the checkpoint method heavily relies on the checkpoint frequency; and replaying program inputs is quite time-consuming when the program crashes after running for a long time.

C2) The crash recovery should be generic to different ROS programs. For this purpose, ROS properties (such as the node model for process execution and message passing for inter-node communication) should be fully considered in recovery.

To solve these challenges, we develop a practical approach named RORY, to perform effective crash recovery of robot software programs in ROS. RORY uses a hybrid checkpoint-replay method to ensure recovery correctness and improve efficiency, and it is generic to different ROS programs by considering ROS properties. Specifically, when a program node runs, RORY records its important data at fixed intervals to create checkpoints, and collects the messages received by the node during each interval. When the node crashes, RORY performs crash recovery using four steps. First, RORY automatically controls the robot to enter a safe status (such as stationary status). Second, RORY isolates the node by stopping its message passing, to avoid disturbance to the recovery process caused by messages from/to other nodes. Third, RORY restarts the node, restores the last checkpoint, and replays the collected messages received by the node during the last checkpoint interval. Finally, RORY finishes the recovery by releasing the node isolation to make the node run normally again. Compared to the ROS restart method, RORY can effectively recover the program's important data that was stored before the crash, to improve recovery correctness. In this paper, we make three main contributions:

- We study the questions proposed by ROS developers and make case studies of typical ROS programs, and find that the ROS restart method can perform incorrect crash recovery in practical use.
- We develop a new approach named RORY to perform effective crash recovery of robot software programs.
- We evaluate RORY on 6 common ROS programs in both virtual and realistic environments, and RORY performs correct crash recovery with modest overhead. The comparison experiments show that RORY outperforms the restart, checkpoint-alone and replay-alone methods.

Yong-Hao Zou and Jia-Ju Bai are with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. Jia-Ju Bai is the corresponding author.

The rest of this paper is organized as follows. Section II introduces the motivation of our work. Section III introduces RORY. Section IV presents the evaluation. Section V introduces the related work, and Section VI concludes this paper.

II. Motivation

We first introduce ROS and its restart method, and then motivate our work by a study of developers' questions about ROS programs and case studies of typical ROS programs.

A. ROS and Its Restart Method

ROS (Robot Operating System) [1] is an open-source software platform of developing and running robotic programs. Because ROS provides many easy-to-use tools and libraries that can simplify the task of creating complex and robust robot behaviors, it has been widely used in research and industry. For extensibility, ROS runs each process of a software program as a *ROS node*. Each node communicates with other nodes via message passing, which is implemented by publish-subscribe style. For example, when *Node A* needs to send a message to *Node B*, *Node A* publishes a specific topic, and *Node B* subscribes to this topic.

Due to the risks of possible faults and errors, a ROS node can inevitably crash in some cases, causing the robot fails to perform the current task. Thus, for robustness, ROS provides a default restart method to automatically restart the crashed node for continuing the task. To enable this restart method, the user needs to set a specific flag *respawn="true"* in the node's launch file.

B. Study of Developers' Questions about ROS Programs

To understand the effectiveness of the ROS restart method, we perform a manual study of the questions proposed in the "ROS Answers" website [3]. We select this website, because it contains many questions proposed by ROS developers for implementing and running ROS programs. We collect the questions from Jan. 2014 to Dec. 2019 (6 years) containing the keyword "crash", and manually select the questions focusing on the crashes of ROS nodes, resulting in 518 questions. We manually read the content of these questions, and find that 60 of them explicitly complain that the robot works abnormally after using the ROS restart method to perform crash recovery. Figure 1 shows such an example[1] of the *move_base* node. This question complains that the node loses correct parameters after restart, causing the robot to move incorrectly in the map.

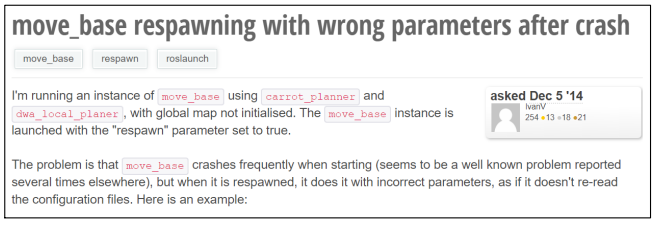


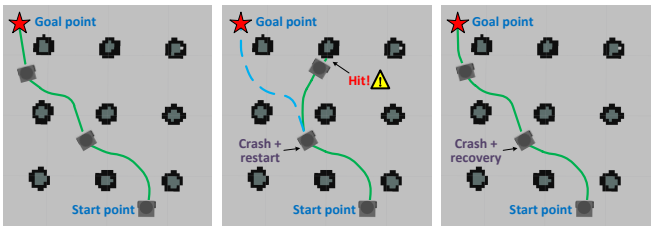
Fig. 1. A question about the ROS restart method for crash recovery.

[1]https://answers.ros.org/question/198950/move_base-respawning-with-wrong-parameters-after-crash/

C. Case Studies of Typical ROS Programs

To validate the effectiveness of the ROS restart method in practical use, we make case studies of two typical ROS programs, namely *move_base* [4] and *hector_mapping* [5]. They are executed on a virtual robot TurtleBot3 Waffle in a robot simulation framework Gazebo 9.0 [6]. We run the *move_base* program to perform route planning, and run the *hector_mapping* program to build the environment map.

Case 1: Route planning. As shown in Figure 2(a), we create a 2D environment containing nine pillars, and let the robot automatically move from a start point to a goal point. We run *move_base* to automatically plan the route on the given map and avoid hitting the pillars. Figure 2(a) shows that the robot successfully finishes the task in the normal case. Then, we enable the ROS restart method, and cause *move_base* to crash by killing it while the robot moves. As shown in Figure 2(b), we observe that the ROS automatically restarts the crashed node, but the robot moves on a wrong route and dangerously hits a pillar. Indeed, *move_base* uses the goal point's location data and the robot's current-location data to plan the route, but such important data stored before the crash is lost after restart, and thus *move_base* cannot use such data to correctly plan the route.



(a) Normal (b) Restart (c) RORY

Fig. 2. Case study of route planning.

Case 2: Map building. As shown in Figure 3(a), we create a 2D environment and manually control the robot to move from a start point to a goal point. While the robot moves, we run *hector_mapping* with a virtual lidar to build a 2D map of the environment. Figure 3(a) shows that the robot successfully finishes the task in the normal case. Then, we enable the ROS restart method, and cause *hector_mapping* to crash by killing it while the robot moves. As shown in Figure 3(b), we observe that the built map is incomplete, and the map data stored before the crash is all lost.

(a) Normal (b) Restart (c) RORY

Fig. 3. Case study of map building.

In fact, the ROS restart method can perform correct crash recovery for the case that the program's data stored before the crash is never used after recovery. This requirement can be satisfied for the programs that never store data, such as

the *chassis* program that simply transfers commands to the chassis without data storage. However, many common ROS programs need to store data for subsequent work, such as *move_base* and *hector_mapping* in our case studies. Thus, for these programs, the ROS restart method can perform incorrect crash recovery. For this reason, it is important to design a new and generic approach that can perform effective crash recovery of robot software programs. In Figure 2(c) and Figure 3(c), we compare our approach RORY to the ROS restart method, and find that RORY produces better results.

III. RORY Design

Checkpoint and replay are two classical crash-recovery methods that have been widely used in common long-running programs (such as distributed-system software and network-service applications). The checkpoint method records program data at fixed intervals to create checkpoints during program execution. When the program crashes, it first restarts the program and then restores the last checkpoint. But the effectiveness of the checkpoint method heavily relies on the checkpoint frequency. It may miss much latest program data when the checkpoint frequency is low, and may introduce much overhead when the checkpoint frequency is high. The replay method continuously records the inputs of the program during program execution. When the program crashes, it restarts the program and replays the recorded inputs to the program. But replaying inputs is quite time-consuming when the program crashes after running for a long time.

Inspired by the above two methods, we develop a new and practical approach named RORY, to perform effective crash recovery of robot software programs in ROS. RORY exploits a hybrid checkpoint-replay method to relieve the limitations of the checkpoint and replay methods alone, in order to ensure recovery correctness and improve efficiency. Besides, RORY is generic to different ROS programs by considering ROS properties.

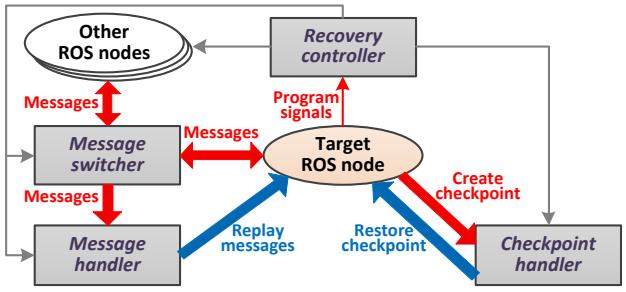


Fig. 4. Overall architecture of RORY.

Figure 4 shows the overall architecture of RORY, which consists of four parts. The recovery controller is used to detect the crash of the target ROS node and control the process of crash recovery; the message switcher is used to control message passing of the ROS node; the message handler is used to record and replay messages of the ROS node; the checkpoint handler is used to create and restore checkpoints for the ROS node. In RORY, each of these parts is implemented as a simple and light-weight ROS node. Based on this architecture, RORY consists of two phases:

P1: Runtime monitoring. While the target ROS node runs normally, RORY records its important data at fixed intervals to create checkpoints, and records messages received by the ROS node during each interval. Figure 5 shows the basic procedure of this phase.

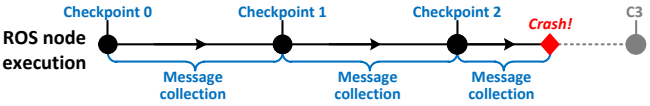


Fig. 5. Procedure of runtime monitoring.

Checkpoint creation. RORY first instruments the key variables that contain the program's important data at compile time, and then uses the checkpoint handler to record their values via the ROS *parameter server* [7] for creating checkpoints at fixed intervals during program execution. Note that these key variables are specific to the target program. For example in the *move_base* code, the variables storing the robot's current-location data and goal-location data are key variables; in the *hector_mapping* code, the variables storing the robot's current-location data and already built map data are key variables. After the user manually provides the key variables in a specific configuration file, RORY instruments them automatically.

Message recording. RORY uses the message switcher to forward messages between other ROS nodes and the target ROS node, and uses the message handler to records the messages received by the target ROS node via the *Rosbag* tool [8]. Specifically, for each message, RORY records its timestamp, message content and published topic. When a new checkpoint is created, RORY drops all recorded messages and starts new message recording.

P2: Crash recovery. When the target ROS node crashes during execution, namely its exceptional signal is caught by the recovery controller, RORY automatically performs crash recovery according to the program data recorded in runtime monitoring, with four steps shown in Figure 6.

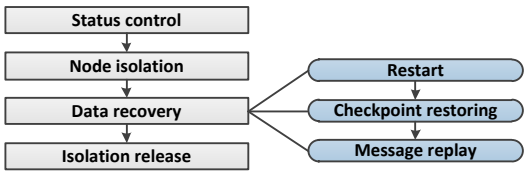


Fig. 6. Steps of crash recovery.

S1: Status control. RORY uses the recovery controller to notify the message switcher, message handler and checkpoint handler of starting crash recovery, and to communicate with other ROS nodes to make the robot enter a safe status. The safe status is specific to the robot's tasks and physical environment. In many cases, the robot is considered to be safe when it is stationary with performing no behavior. Thus, the recovery controller sends messages to the ROS nodes controlling robot movement (e.g., the *move_base* and *chassis* nodes) to keep the robot stationary.

S2: Node isolation. RORY isolates the crashed ROS node by stopping its message passing. This step is necessary,

because other ROS nodes are unaware of that the node has crashed and they can still communicate with the node via messages, causing two possible side effects: 1) other ROS nodes may send messages to the crashed node, which may disturb its recovery process; 2) other ROS nodes may be disturbed by the messages sent by the crashed node during its recovery process. Specifically, the message switcher disables message passing between the crashed node and other ROS nodes, to isolate the node.

S3: Data recovery. RORY first restarts the crashed ROS node, and then uses the last checkpoint and recorded messages to recover its important data. To restore checkpoint, when the node is initialized, the checkpoint handler retrieves these key variables stored in the ROS parameter server. To replay messages, after the node is initialized, the message handler sends the messages recorded in the *Rosbag* tool to the node. Note that in ROS, each message is sent through a specific topic, and a node receives this message by subscribing to this topic. Thus, to replay messages to the node, the message handler creates a shadow topic for each original topic to which the node subscribes, and lets the node subscribe to all such shadow topics. Through these shadow topics, the recorded messages are replayed strictly according to their timestamps. For example, before the node crashes, suppose that it receives *MSG1* at the time point *P1* and then receives *MSG2* at the time point *P2*. Accordingly, when replaying *MSG1* and *MSG2* during crash recovery, RORY first replays *MSG1* and then replays *MSG2* after waiting for *P2-P1* time.

S4: Isolation release. RORY releases the isolation of the crashed ROS node by enabling its message passing. Then, the node can normally communicate with other ROS nodes to continue its task. Specifically, the message switcher enables message passing between the crashed node and other ROS nodes, to release the isolation.

IV. Evaluation

A. Experimental Setup

To validate the effectiveness of RORY, we perform evaluation on a virtual robot TurtleBot3 Waffle in the robot simulation framework Gazebo 9.0 and a physical robot based on the TurtleBot2 platform. Figure 7 shows the physical robot, which has a iClebo Kobuki chassis [9], a SICK TIM551-2050001 2D lidar [10] and a ASUS Xtion PRO LIVE 3D camera [11]. For the virtual and physical robots, we test 6 common ROS programs with ROS Melodic Morenia in Ubuntu 18.04:

- *hector_mapping* [5]. It implements a SLAM approach that can be used without odometry. We use it to build a 2D map with the lidar, while the robot moves.
- *RTAB-Map* [12]. It implements a RGB-D, stereo and lidar graph-based SLAM approach with an incremental appearance-based loop closure detector. We use it to build a 2D map with the camera, while the robot moves.
- *ORB_SLAM2* [13]. It is a real-time SLAM library that computes the camera trajectory and a sparse 3D reconstruction. We use it to reconstruct a 3D model with the camera, while the robot moves.

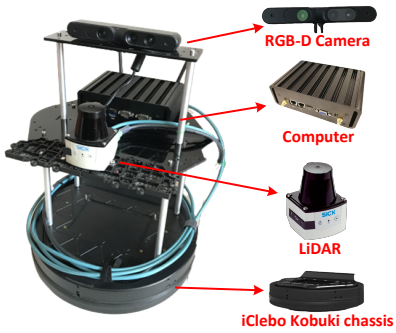


Fig. 7. The physical robot used in the experiment.

- *move_base* [4]. It provides an implementation of attempting to reach the goal with a mobile chassis, given a goal in the world. We use it to perform route planning for the robot navigation.
- *AMCL* [14]. It implements an adaptive Monte Carlo localization approach, which uses a particle filter to track the position of a robot against a known map. We use it to perform localization for the robot navigation.
- *LaMa_local* [15]. It is a fast scan matching approach to mobile robot localization supported by a continuous likelihood field. We use it to perform localization for the robot navigation.

We configure the virtual robot to work in two virtual environments *Office_small* and *Workshop* selected from the 3DGEMS datasets [16]. We place the physical robot in a realistic room containing an obstacle, and we subsequently refer to this environment as *Real_room*. In each environment, we manually control the robot to move from a start point to a goal point, and run *hector_mapping*, *RTAB-Map* to build a 2D map of the environment and run *ORB_SLAM2* to build a 3D model of the environment. Then, with a complete 2D map of the environment, we run *move_base* for route planning and run *AMCL* and *LaMa_local* for localization, to make the robot automatically move from a start point to a goal point. In the evaluation, we set the interval of creating checkpoints in RORY as 2 seconds. The user can conveniently change this interval as needed.

B. Crash Recovery

For each tested ROS program, we first run its node normally without crash, as the normal case. Then, we manually kill the node to simulate a crash caused by a transient fault; when the program node runs for 9 seconds, we run the ROS restart method and RORY to perform crash recovery of this node, respectively. Finally, we compare the robot's behaviors and produced results in the normal and crash cases, to check whether the two methods perform correct crash recovery.

For *hector_mapping*, *RTAB-Map* and *ORB_SLAM2*, we check whether the map or model produced in the crash case is complete compared to the normal case; for *move_base*, *AMCL* and *LaMa_local*, we check whether the robot reaches the goal point in the crash case. We run each program in each environment five times. Table I shows the results. The symbol ✓ means the recovery is correct in all the tests, and the symbol × means the recovery is incorrect in all tests.

TABLE I

RESULTS OF CRASH RECOVERY.

Program	Office_small		Workshop		Real_room	
	Restart	*RORY*	*Restart*	*RORY*	*Restart*	*RORY*
hector_mapping	×	✓	×	✓	×	✓
RTAB-Map	×	✓	×	✓	×	✓
ORB_SLAM2	×	✓	×	✓	×	✓
move_base	×	✓	×	✓	×	✓
AMCL	×	✓	×	✓	×	✓
LaMa_local	×	✓	×	✓	×	✓

From Table I, we find that the ROS restart method performs incorrect crash recovery in all the tests, as it loses the program's important data that was stored before the crash and is used after recovery. Specifically, for *hector_mapping* and *RTAB-Map*, the built map is incomplete, as the map data collected before the crash is all lost; for *ORB_SLAM2*, the reconstructed model is incomplete, as the model data collected before the crash is all lost; for *move_base*, *AMCL* and *LaMa_local*, the robot does not reach the goal, and it even dangerously hits the wall or objects in some tests, as the goal point's location data and the robot's current-location data stored before the crash are lost. On the contrary, RORY performs correct crash recovery for all the tests, as it recovers the program's important data that was stored before the crash.

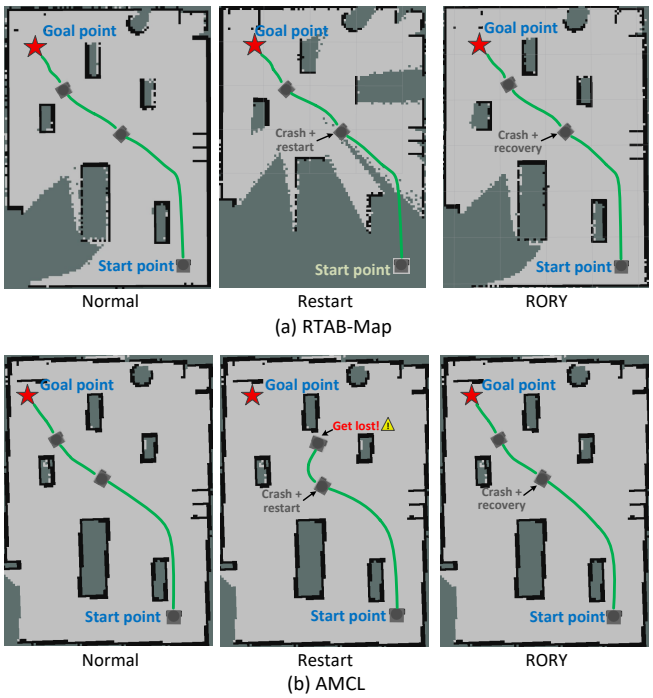


Fig. 8. Results in the *Workshop* virtual environment.

Figure 8 shows the recovery results of *RTAB-Map* and *AMCL* in the *Workshop* virtual environment. In Figure 8(a), the built map is incomplete when the ROS restart method is used, while the built map is more complete when RORY is used. In Figure 8(b), the robot gets lost when the ROS restart method is used, while the robot reaches the goal when RORY is used. In summary, RORY produces better results than the ROS restart method in crash recovery.

TABLE II

RUNTIME OVERHEAD OF RORY.

Program	Original		RORY	
	CPU	*Memory*	*CPU*	*Memory*
hector_mapping	19.3%	440M	31.4% (1.6x)	487M (1.1x)
RTAB-Map	15.9%	520M	19.3% (1.2x)	577M (1.1x)
ORB_SLAM2	25.3%	907M	29.3% (1.2x)	1237M (1.4x)
move_base	5.2%	422M	6.4% (1.2x)	481M (1.1x)
AMCL	5.6%	420M	6.5% (1.2x)	477M (1.1x)
LaMa_local	14.5%	282M	16.9% (1.2x)	371M (1.3x)

C. Runtime Overhead

The main runtime overhead of RORY is caused by runtime monitoring of the program node. To quantify this overhead, we measure the CPU utilization and memory cost of the target program node and RORY during execution. We only perform this experiment on the physical robot in the environment *Real_room*, because we believe that the results of the physical robot is more representative to quantify the realistic overhead. We test each program five times and then calculate the average value of CPU utilization and memory cost. Table II shows the measurement results. We find that the runtime overhead of RORY is about 1.2x on average, which is modest in practical use.

D. Comparison Experiment

When performing crash recovery, RORY uses a hybrid checkpoint-replay method. To show the value of this hybrid, we implement a checkpoint-alone method and a replay-alone method, and compare them to RORY. Then, we evaluate these two methods and RORY on the 6 tested ROS programs in the three environments. We run each program with each method in each environment five times. As in Section IV-B, we manually kill the node to cause a crash when it runs for 9 seconds, and use the three methods to perform recovery.

Table III and Table IV show the recovery correctness and time usage, respectively. The symbol ✓ means the recovery is correct in all the tests, the symbol × means the recovery is incorrect in all tests, and the symbol ? means the recovery is correct in some tests but incorrect in the other tests. From Table III and Table IV, we make two observations:

First, the checkpoint-alone method fails to perform correct crash recovery in the most tests. Because this method loses the program's important data stored between the last checkpoint and crash point, it cannot retrieve such data in crash recovery. However, we also observe that the checkpoint-alone method performs correct crash recovery in several tests of *AMCL* and *LaMa_local*. Indeed, *AMCL* and *LaMa_local* both use a probabilistic algorithm for localization, and thus in some cases, they can still produce correct results even though history data is lost. Compared to the checkpoint-alone method, RORY additionally records and replays the messages that the program receives between the last checkpoint and crash point, which makes data recovery more complete; but RORY spends a longer time than the checkpoint-alone method in recovery, due to replaying messages.

TABLE III

RECOVERY CORRECTNESS IN THE COMPARISON EXPERIMENT.

Program	Office_small			Workshop			Real_room		
	Checkpoint	*Replay*	*RORY*	*Checkpoint*	*Replay*	*RORY*	*Checkpoint*	*Replay*	*RORY*
hector_mapping	×	✓	✓	×	✓	✓	×	✓	✓
RTAB-Map	×	✓	✓	×	✓	✓	×	✓	✓
ORB_SLAM2	×	✓	✓	×	✓	✓	×	✓	✓
move_base	×	✓	✓	×	✓	✓	×	✓	✓
AMCL	?	✓	✓	?	✓	✓	?	✓	✓
LaMa_local	?	✓	✓	?	✓	✓	?	✓	✓

TABLE IV

RECOVERY TIME IN THE COMPARISON EXPERIMENT.

Program	Office_small (ms)			Workshop (ms)			Real_room (ms)		
	Checkpoint	*Replay*	*RORY*	*Checkpoint*	*Replay*	*RORY*	*Checkpoint*	*Replay*	*RORY*
hector_mapping	2145	9560	3997	2141	9534	3971	2478	9939	4157
RTAB-Map	5	11214	2903	3	11582	2892	9	12991	4397
ORB_SLAM2	1	9960	3410	1	9460	3280	3	9910	3070
move_base	1	9459	1597	1	9318	1547	1	9155	1478
AMCL	1	9603	1615	2	9343	1649	8	10162	2235
LaMa_local	1	9766	2334	1	9634	2301	3	9667	2267

Second, the replay-alone method performs correct crash recovery for all the tests, but its recovery time is much longer than that of RORY. Indeed, for most ROS program nodes, their inputs are just messages, and thus replaying the recorded messages can perform correct recovery. However, the replay-alone method records and relays all the messages that the program receives between the start time and the crash point, and these messages must be replayed in order according to their timestamps. Thus, in theory, the recovery time of the replay-alone method should be equal or longer than the node's execution time before crash. Compared to the replay-alone method, RORY records and replays the messages that the node receives only between the last checkpoint and the crash point, and thus it spends less time on replaying messages. Thus, plus the time usage of restoring checkpoint, which is often little in practice, RORY spends a shorter time than the replay-alone method on crash recovery.

V. RELATED WORK

Existing approaches that can perform crash recovery of robot software programs are based on the restart, redundancy or checkpoint methods.

Based on the ROS restart method, some approaches [17]–[19] perform fault tolerance of ROS programs. However, our case studies in Section II-C show that the restart method can perform incorrect crash recovery, because it loses the program's important data that was stored before the crash and is used after recovery.

Some approaches [20]–[23] are redundancy-based, namely they run one or more redundant nodes of the target ROS program. They forward the messages received by the original node to these redundant nodes at runtime. When the original node crashes, these approaches automatically replace the crashed node with a redundant node to continue the work. However, realistic robot systems often have limited system resources and computing capacity, and thus running redundant nodes may be quite expensive when the target program is large and complex.

Two recent approaches [24], [25] exploit the checkpoint method to perform crash recovery of the ROS master. However, like the checkpoint-alone method that we implemented in Section IV-D, they may lose the program's important data stored between the last checkpoint and crash point. On the contrary, RORY uses a hybrid checkpoint-replay method to improve the completeness of data recovery, and it is also generic to different ROS programs.

VI. CONCLUSION

For robustness of robot systems, when a robot software program crashes, it should be correctly recovered to continue its task. To this end, ROS provides a default restart method for crash recovery. However, our case studies on typical ROS programs show that this method can perform incorrect crash recovery, because it loses the program's important data that was stored before the crash and is used after recovery. To solve this problem, we propose a practical approach named RORY, to perform effective crash recovery of robot software programs in ROS. RORY uses a hybrid checkpoint-replay method, and it is generic to different ROS programs by considering ROS properties. The evaluation on 6 common ROS programs shows its effectiveness.

RORY can be improved in some aspects. For example, the time usage of replaying messages in RORY is still a bit long, as RORY replays all the recorded messages strictly according to their timestamps. We plan to reduce this time usage by analyzing the dependency between recorded messages and dropping the messages that are useless for recovery. Besides, we only use RORY to recover single-robot programs at present. We plan to apply RORY to multi-robot programs, by considering synchronization between multiple robots.

ACKNOWLEDGMENT

We thank anonymous reviewers for their comments, and thank Shi-Min Hu and Julia Lawall for their suggestions on improving the paper. This work was supported by the Natural Science Foundation of China under Project 62002195.

REFERENCES

[1] "ROS: an open-source software platform for building robot applications," https://www.ros.org/.

[2] "Configure the ROS launch file to restart crashed node," https://wiki.ros.org/roslaunch/XML/node.

[3] "Questions and answers of using the ROS," https://answers.ros.org/questions/.

[4] "move_base: a package for route planning and movement control for the robot," http://wiki.ros.org/move_base.

[5] "hector_mapping: a scalable SLAM approach for the robot," http://wiki.ros.org/hector_mapping.

[6] "Gazebo: a robot simulation framework," http://gazebosim.org/.

[7] "Parameter server for storing and retrieving data in the ROS," http://wiki.ros.org/Parameter%20Server.

[8] "Rosbag tool in the ROS," http://wiki.ros.org/rosbag.

[9] "SICK TIM551-2050001 2D lidar," https://www.sick.com/us/en/detection-and-ranging-solutions/2d-lidar-sensors/tim5xx/tim551-2050001/p/p343045.

[10] "iClebo Kobuki chassis," http://kobuki.yujinrobot.com/.

[11] "ASUS Xtion PRO LIVE 3D camera," https://www.asus.com/ 3D-Sensor/Xtion_PRO_LIVE.

[12] "RTAB-Map: an application of real-time appearance-based mapping," https://github.com/introlab/rtabmap.

[13] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[14] "AMCL: a probabilistic localization application for a robot moving in 2D," http://wiki.ros.org/amcl.

[15] E. Pedrosa, A. Pereira, and N. Lau, "Efficient localization based on scan matching with a continuous likelihood field," in *Proceedings of the 2017 International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2017, pp. 61–66.

[16] "3DGEMS: 3D Gazebo model dataset," http://data.nvision2.eecs.yorku.ca/3DGEMS/.

[17] S. Zaman, G. Steinbauer, J. Maurer, P. Lepej, and S. Uran, "An integrated model-based diagnosis and repair architecture for ROS-based robot systems," in *Proceedings of the 2013 International Conference on Robotics and Automation (ICRA)*, 2013, pp. 482–489.

[18] S. Zaman and G. Steinbauer, "Automated generation of diagnosis models for ROS-based robot systems," in *Proceedings of the 24th International Workshop on Principles of Diagnosis*, 2013, pp. 92–98.

[19] A. Lutac, N. Chechina, G. Aragon-Camarasa, and P. Trinder, "Towards reliable and scalable robot communication," in *Proceedings of the 15th International Workshop on Erlang*, 2016, pp. 12–23.

[20] M. Lauer, M. Amy, J.-C. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Engineering adaptive fault-tolerance mechanisms for resilient computing on ROS," in *Proceedings of the 17th International Symposium on High Assurance Systems Engineering (HASE)*, 2016, pp. 94–101.

[21] M. Amy, J.-C. Fabre, and M. Lauer, "Towards adaptive fault tolerance on ROS for advanced driver assistance systems," in *Proceedings of the 47th International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2017, pp. 29–35.

[22] M. Lauer, M. Amy, J.-C. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Resilient computing on ROS using adaptive fault tolerance," *Journal of Software: Evolution and Process*, vol. 30, no. 3, 2018.

[23] H. Ahn, S. C. Ahn, J. Heo, and S. Y. Shin, "Fault tolerant framework and techniques for component-based autonomous robot systems," in *Proceedings of the 2011 International Symposium on Applied Computing (SAC)*, 2011, pp. 566–572.

[24] T. Jain and G. Cooperman, "DMTCP: fixing the single point of failure of the ROS master," 2017.

[25] P. Kaveti and H. Singh, "ROS Rescue: fault tolerance system for Robot Operating System," *arXiv preprint arXiv:1910.01078*, 2019.