Distributed Fault Diagnostic for Multiple Mobile Robots Using an Agent Programming Language

Márcio G. Morais and Felipe R. Meneguzzi and Rafael H. Bordini and Alexandre M. Amory
Informatics Faculty, PUCRS University, Porto Alegre, Brazil

Abstract—**Programming autonomous multi-robot systems can be extremely complex without the use of appropriate software development techniques to abstract the hardware heterogeneity from the complexity of distributed software to coordinate autonomous behavior. Moreover, real environments are dynamic, which can generate unpredictable events that can lead the robots to failure. This paper presents a highly abstract cooperative fault diagnostic method for a team of mobile robots described on a programming environment based on ROS (Robot Operating System) and the Jason multi-agent framework. When a robot detects a failure, it can perform two types of diagnostic methods: a local method executed on the faulty robot itself and a cooperative method where another robot helps the faulty robot to determine the source of failure. A case study demonstrates the success of the approach on two turtlebots.**

I. Introduction

Autonomous mobile robots are being increasingly employed in real-world applications and places such as homes, hospitals, and shopping malls. Such robots have different types of sensors, actuators, and processing elements, resulting in a typically highly heterogeneous hardware platform. Moreover, the services that the robots provide are also becoming more sophisticated, enabling some degree of autonomy even in unknown or partially known environments. Consequently, it is becoming unfeasible to program such sophisticated systems as a single monolithic application. For this reason, several software architectures were created to enable modular software design, where the modules can interact with each other by passing data and messages [1], [2]. This approach allows the design of parallel applications, making software complexity manageable and resulting in highly maintainable and reusable software modules.

When the robots face the real world, many unpredicted situations can occur. Carlson et al. [3] demonstrate that reliability in the field is typically low, between 6 to 24 hours of mean time between failures (MTBF), even for tele-operated robots. This low reliability requires constant human intervention, which hinders the purpose of using robots in the first place. One common way to increase the reliability is by redundancy (in hardware or software). However, this increases the cost and design complexity which can also hamper the use of mobile robots for cost-sensitive applications. Instead of building a single very expensive robot with extensive use of redundancy to increase reliability, it might be more efficient, economic, and reliable to have multiple simpler robots collaborating in a given task. Multiple robots provide parallelism, redundancy, and tolerance to individual robot failure.

The motivation of this work is to provide a programming infrastructure which enables hardware abstraction and code reuse for the described fault plans, increasing the robot's fault diagnosis capability and its overall reliability. The fault plans presented in this paper are described in a highly abstract way using an agent programming language (at the top layer) and a robotic software framework (at the middle layer). Thus, our main contribution in this paper is a method of cooperative high-level fault diagnostic using an agent programming language where two or more robots can collaborate to improve the fault diagnostic, abstracting hardware details from fault diagnostic procedure, enabling fault plan reuse in a heterogeneous robotic system, and potentially enabling a more effective fault isolation and recovery methods.

The rest of this paper is organized as follows. Section II surveys two main software programming environments for robotics: robotic software frameworks (RSF) and multi-agent systems (MAS). Section III reviews both programming environments in terms of their existing fault tolerance features. Section IV describes the main contribution of this paper, including a case study of the integration of RSF and MAS to describe more abstract fault plans. Section VI gives some final remarks and discusses directions for future work.

II. Robot Programming Approaches

Abstract and structured programming approaches become necessary as robotic applications require additional complexity in behavior, increased autonomy, and the ability to adapt to dynamic events. In this context, robotic software architectures require more scalability, reusability, efficiency and fault tolerance [2] to cope with unforeseen events once they are deployed in a real-world environment. For these reasons, modern robotic software architectures have a number of common key features [2], such as: a distributed architecture and inter process communication; high modularity to improve code reuse and scalability; robustness and fault tolerance to avoid a single fault causing the entire system to crash, and to allow the system to achieve its goals with the available resources; and the capacity to deal with events in real time and with high efficiency.

Robotic Software Frameworks (RSFs) typically consist of software modules that can interact with each other via inter-process communication, as well as a set of tools for usual software development tasks such as package creation, building, execution, simulation, debugging, and logging. Importantly, RFSs also include the most common drivers for "off-the-shelf" robotics sensors/actuators, well-known libraries for modeling kinematics, physics, localization, mapping, navigation, gripping among other common robotics capabilities. There are several RSFs available such as Orocos [4], MIRA [5], ROS [6], among many others that have been surveyed in [1], [2]. In those surveys, ROS stands out because it is fast becoming the *de facto* open source platform used for applications in robotics, with a large number of resources, libraries, drivers, and tools incorporated. Therefore, we have selected ROS to use as the lower level software platform in our research to interface high-level agent programming and low-level robotic algorithms and hardware.

Recent work on Multiagent Robotic Systems (MARS) [2] compares software infrastructure for MAS and for RSF. The authors suggest advantages when both software systems are used in robotics, enabling the programming of high-level behaviors. For this reason we selected Jason [7] as an extension of the well-known AgentSpeak(L) language [8] as the programming abstraction for goal-directed behavior in our multi-robot programming platform. AgentSpeak is an agent programming language that follows the Beliefs, Desires, and Intentions (BDI) architecture [9]. In BDI agents, reasoning is based on three main mental components: beliefs (representing an agent's knowledge), desires (representing goals to be achieved) and intentions (representing commitments to particular desires and courses of actions to achieve them). The authors suggest the book [7] for more details on how to program plans with AgentSpeak

III. Reliability in Robotics

As mentioned before, the typical MTBF of robotic systems is very low. According to Carlson et al. [3], it is between 6 to 24 hours. Crestani and Godary-Dejean [10] also report low MTBF for robots used in military, urban, and rescue missions. Most common faults are related to actuators [3].

System-level reliability methods generally implement at least these three main steps: fault detection; fault diagnosis; recovery. Most papers on fault tolerance for robotics propose some new fault detection / diagnostic approaches for mobile robots, which were surveyed in [11]. However, the focus of this paper is on a system-wide and integrated diagnostic approach for multi robots. The rest of this section reviews the existing approaches for RSF and MAS.

The RSF survey papers [1], [2], [10] raise criticism about fault tolerance in robotics: despite the obvious need for fault tolerance, only a few RSFs implement any FT mechanism. Elkady and Sobh [1] evaluates several RSFs in terms of criteria such as fault detection and recovery capabilities. Most of them have no explicit fault handling capability except for the support of software exceptions, logging, and playback. Melchior and Smart [12] describe a recovery-oriented framework for mobile robots which requires no outside intervention. The authors focus on general programming faults that can cause programs to halt or crash. In this case, the recovery mechanism can restart the process before a complete system failure. Kirchner et al. [13] propose a self-healing architecture for multi-robot systems based on ROS. It enables the developer to add components for monitoring, diagnostic, and recovery. The authors present no quantitative or qualitative evaluation.

IV. Case Study on Cooperative Fault Plans for Multiple Robots

This section presents the case study used to evaluate the proposed cooperative fault plans for multiple robots. Section IV-A describes the case study, Section IV-B details the RFS/MAS interface, and Section IV-C presents the robot's local and cooperative fault plans.

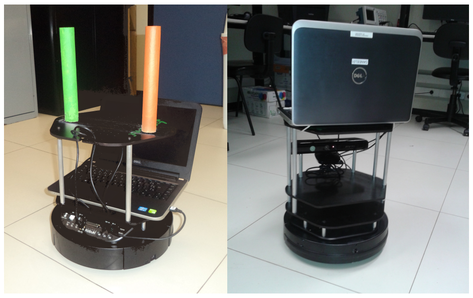

(a) Faulty (b) Helper

Fig. 1: Faulty and Helper robots used in the experiments.

A. Case Study and Fault Model

The photo in Figure 1 illustrates our experimental set up with the two Turtlebots[1]. Both robots (called, respectively, **Faulty** and **Helper**) have similar hardware: a laptop computer (Core i7 with 8GByte RAM), Kobuki mobile base with 2 differential wheels with wheel encoders. The Helper robot has a Kinect sensor array, and the Faulty robot has two colored cylinders on its top, for the purposes of identification and pose estimation. Both robots have the same software, which consists of the software environment presented in Section IV-B and the fault plans described in Section IV-C.

Faulty only has wheel encoders to monitor its movements, it has neither depth sensor nor camera, which would be useful to disambiguate faults by itself. Helper has wheel encoders, depth sensors, and camera, to safely navigate on an environment. Kinect is a cheap and effective depth sensor for robotics, although it has a limited field of view (57° horizontally and 43° vertically) and it has a minimum viewing distance of approximately 80 cm. Thus, obstacles outside the field of view or closer to the minimum distance cannot be detected by the robot, potentially causing undesired collisions. Moreover, Kinect is not able to detect reflective surfaces and glass walls/windows. Due to these limitations, we require a fault plan to identify the root cause of a navigation failure.

We consider the following fault scenarios within the fault plans. The Faulty robot is expected to move a certain distance and one of the following outcomes can happen: (*i*) the robot is able to perform the action; (*ii*) there is a fault on the wheel encoders. (*iii*) there is a fault in both actuators of the wheels; (*iv*) there is a fault in the actuator of one of the wheels;

Note that there are multiple causes in the situation where a robot cannot achieve a certain navigation goal. For instance, the robot may not move forward because a wheel is stuck or the actuator is defective; the motor is actually working but the wheel encoder is not detecting the resulting wheel movement (e.g., broken wire). Each of these causes is ambiguous since they may lead to the same outcome: *the robot cannot reach the desired destination*. The purpose of the fault plan is to automatically identify which of the possible faults actually happened so that an adequate recovery strategy can be started.

[1]http://www.turtlebot.com/.

In some situations, Faulty cannot identify the source of a fault with its own sensors, since the sensors themselves can be the source of the fault. In this case, the robot asks for help from other robots, which in our scenario is exemplified by the Helper robot. The Helper "lends" its perceptions so that the Faulty robot can disambiguate and confirm the source of the fault, enabling an effective fault recovery. For instance, during an action where a robot needs to perform some movement, if their odometry system reports no progress, the robot is not able to determine if the odometry system is the source of the fault itself or whether the odometry system is correctly reporting some fault in the motion system. A similar situation happens when an actuator has fault. This fault produces an unexpected movement and Faulty is not able to isolate the source of the fault that can be a fault in one of the actuators or the encoders reporting wrong values. In these situations, an inspector robot plays a key role in the fault isolation process since it can help the faulty robot by visually locating it and detecting any movement, then giving feedback during the fault isolation procedure. Feedbacks of movement during the fault isolation procedure help Faulty to know whether it is moving or not and, in case of movement, whether the robot can move in a straight line or not.

The experiment consists of generating each of these faults to evaluate the diagnostic correctness of the proposed cooperative fault plans presented in Section IV-C.

B. ROS Layers

Figure 2 illustrates the RSF layer describing the ROS nodes used in the described case study. This figure is divided into three parts representing the types of ROS nodes: drivers, ROS application, Jason.

A set of ROS nodes has been implemented to perform different tasks according to the needs of the case study. In functional layer, freenect_node and Kobuki_node, provide the interface to the hardware; In execution layer, motion_decomposer, visual_synthesizer, and fault_injector are responsible for the execution of actions and generation of information to decision layer; In decision layer, Jason agents connected to ROS take decisions based on information coming from execution layer.

1) Functional Layer: The functional layer is the lowest layer within the three-tier architecture. It is responsible for implementing an interface between the layers of the highest level and the hardware through elementary operations performed on sensors and actuators. This section presents an overview of the rosnodes from functional layer, they are already available as part of ROS packages.

freenect_node [2] is a ROS node which is a ROS driver for kinect hardware, that acquires data from Kinect and publishes messages on several topics related with camera calibration, image, depth and combined information such as registered depth camera (aligned with RGB camera) and point clouds.

Similar to freenect_node, kobuki_node is a ROS wrapper for the iClebo Kobuki [3] mobile base driver. kobuki_node publishes messages on several topics about hardware diagnostics, events (for instance, a bumper hit) and odometry. It also subscribes to topics related to commands to actuators and its internal odometry system.

2) Execution Layer: The execution layer is responsible for defining a software infrastructure where the exchange of information between layers or modules of the system is performed. Among its responsibilities are the use of raw data aiming to provide resources and services to the system, and the interconnection between the functional layer and the decision layer.

The motion_decomposer ROS node translates the agent's high-level basic motion actions as "turn(45)" into ROS messages to the nodes of the functional layer. This node subscribes to the topic /jason/actions and handles incoming messages whose high-level action is "move" or "turn".

The fault_injector is a ROS service that intercepts messages from other nodes to kobuki_node and changes the value of the parameters simulating the behavior of the active fault.

This visual_synthesizer ROS node gives to the Helper's agent a perception of the presence of Faulty and its coordinates, in order to locate Faulty or give it a feedback during a cooperative fault diagnosis procedure. The perception about Faulty is given in form of its position, pose and depth.

The colored cylinders are detected by using OpenCV functions to extract blobs[4] according to the desired colors. *InRange* function is used to filter color pixels out by checking if image color pixels lie between lower and upper boundaries. Morphological operations are performed to remove small objects and fill small holes. Thus, the remaining shapes in the resulting image are the candidate blobs.

The procedure to determine the best pair of blobs is executed in two steps. First, the pairs of blobs whose distance is greater than 25 cm or less than 22 cm (22.7 cm is the known distance between cylinders) of distance between them are excluded. Second, among the remaining pairs, the pair whose distance is closer to 22.7 cm is selected.

Once the colored cylinders have been identified, the centroid of their areas is calculated and their depths extracted from the point cloud; the distance *depth* between the robots is the mean value from the depth of the centroid of the region of each cylinder $c1$ and $c2$ and it is extracted directly from the point cloud.

$$depth = \frac{c1 + c2}{2}$$

Position is given in degrees from the distance between the center of the captured image and the middle-point of the pair of the cylinders. Once it is known that Kinect has a horizontal field of view of 57°, it is possible to map angular offsets up to 28.5° left or right, from the image center to middle point of the pair of colored cylinders. Thus, *angle* is given as follow:

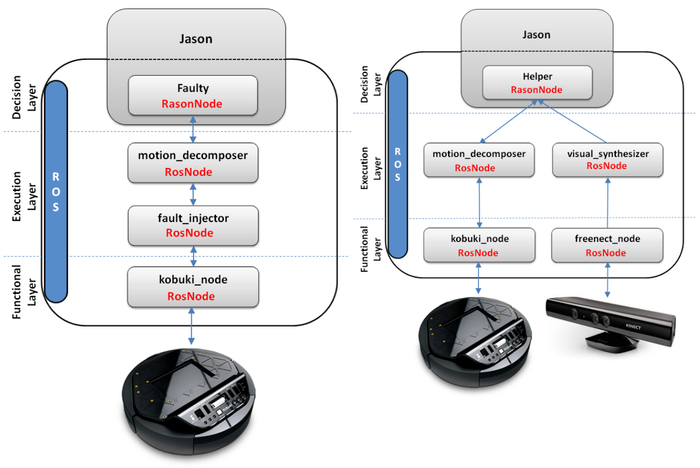
$$angle = \frac{qrc\_center\_w - (\frac{img\_width}{2})}{\frac{img\_width}{2}} \times 28.5$$

Where *qrc_center_w* is the middle-point of a pair of colored cylinders and *img_width* is the total number of columns of the image frame.

[2]freenect_node is a libfreenect-based ROS driver for the Microsoft Kinect. libfreenect is available at http://openkinect.org

[3]http://kobuki.yujinrobot.com

[4]In computer vision, blobs are regions in a digital image that differ in properties, such as brightness or color, compared to surrounding regions.



(a) Faulty (b) Helper

Fig. 2: Nodes organization for Faulty(a) and Helper(b).

Since the distance H between the colored cylinders is known (22.7 cm), and the distance from the Helper to the centroid of the colored cylinders has been obtained from the depth point cloud, it is possible to estimate the pose of Faulty in relation to Helper by estimating the value of the cosine of β, as follows:

$$\cos(\beta) = \frac{hypotenuse}{adjacent\ cathete} = \frac{H}{c2 - c1}$$

Where $c1$ and $c2$ are the distances from the Helper to the centroid of the cylinders.

3) Decision Layer: In order to integrate ROS and Jason, we developed a standardized interface that allows Jason agents to connect to ROS and communicate with ROS nodes in charge of sending information and executing actions. The interface comprises two internal classes in Jason to become Jason agents into ROS nodes named RasonNodes. Thus, decision layer has Jason running agents with fault plans which have only high-level actions and perceptions which are not hardware-specific. The plans are explained in Subsection IV-C.

Figure 2 represents the internal software organization of our two robots, Faulty (Figure 2a) and Helper (Figure 2b). In this example, a single agent represents a single robot with one computer. However, it is possible to model a single robot with multiple computers or even multiple agents. For instance, an humanoid robot can have an agent for each arm, another agent for the legs, etc. Multiple robots can be modeled by instantiating multiple agents in the robot's computers. Jason agents running on different computers can communicate via JADE [14].

C. Agents' Plans

This section describes the fault plans used to isolate and diagnose the faults described in Section IV-A. Figures 3 and 4 present behavior trees which illustrate the robots plans. A Behavior Tree is a model of plan execution used robotics to describe switchings between a finite set of tasks in a modular way. Each node represents a task and a complex task could have several child tasks. There are two key node types in Behavior trees: selector and sequencer. The selector,

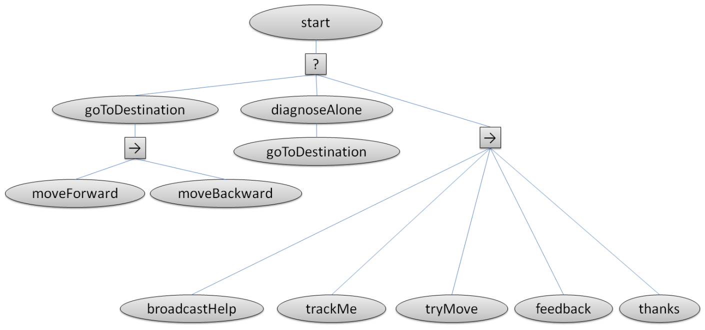


Fig. 3: Faulty's behavior tree.

represented by a "?" symbol, executes the next sibling node when the execution of the current node fails. The sequencer node, identified by a "→" symbol, executes the next sibling node when the current node has its execution successfully finished.

1) Faulty's Plans: Figure 3 shows a behavior tree that describes the Faulty's plans used to reach the goal as well as plans to deal with its faults. Initially, the faulty robot tries to reach a destination using the goal `goToDestination`. When Faulty detects it is not able to reach the destination (due to the failure of `goToDestination` action), it executes its local fault plan `diagnoseAlone`. The local fault plan for the fault scenarios described in Section IV-A is empty once Faulty has no devices to disambiguate the source of the fault. When the second attempt to move fails, it starts a cooperative fault detection strategy by broadcasting a help request (`broadcastHelp`). When Faulty receives a positive response from Helper, it asks Helper to track it (`trackMe`) and, when Helper is ready to track it (`tryMove`), Faulty tries to move again. After completing the movement, Faulty asks Helper for a feedback on its movement and evaluates the source of the fault(`feedback`). A `thanks` is sent back to Helper to let it know the help is no more needed.

The resulting movement is calculated according to the fault detected. When Faulty tries to move but no progress is detected, if Helper reports a current position, pose, or depth different than the initial one, it considers the encoders have fault, otherwise either the actuators have faults or the wheels are stuck. When Faulty detected a movement that is in discordance with the expected movement, the composition of Faulty's pose and angle is compared to the projected values, higher values indicate fault in right actuator, lower values indicate fault in left actuator, otherwise is considered a false positive.

2) Helper's Plans: As we saw in Section IV-C, when the Faulty robot fails to isolate the source of a fault, it asks for help, broadcasting a help request in the network. This situation results in the Helper robot receiving a request to help Faulty, and the two robots trying to diagnose the problem interacting between them, using the plans presented in the behavior trees shown in Figure 3 and Figure 4. The next steps refer to Helper's behavior tree, shown in Figure 4.

When the Helper robot receives a request for help from Faulty (`helpMe`), it first tries to detect the Faulty robot by slowly rotating on its axis until both colored cylinders from the Faulty robot are close to the center of its field of

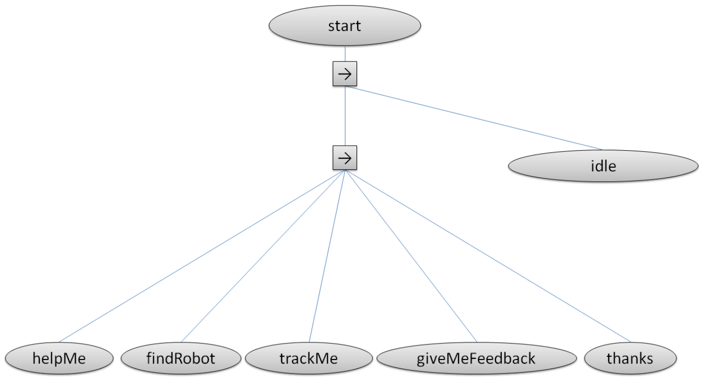


Fig. 4: Helper's behavior tree

vision (`findRobot`). Once Faulty was detected and Helper is positioned, the latter informs Faulty that it has found it. Upon receiving confirmation that it has been detected, Faulty in turn asks Helper to keep tracking it while it moves (`trackMe`). Helper then proceeds to track Faulty's movement by telling it to move, and observing Faulty's behavior. When Faulty has completed its movement, it requests Helper's perceptions about its position, pose, and depth (`giveMeFeedback`). Once the diagnosis has been completed, Faulty sends a message to Helper to let it know that the cooperative diagnostic procedure is finished, so Helper can reset the information about Faulty and return to its tasks (`thanks`).

V. Results

This section presents the results for the fault scenarios described in the case study in Section IV in terms of latency to complete the diagnostic steps, which are: the fault detection step, the robot detection step, and the fault diagnosis step. The *fault detection step* is not a cooperative step but, as all the other steps, it involves the integration between Jason and ROS, either to move the robot or to generate events of faults to the decision layer. This step begins when Faulty starts pursuing its goal and finishes when Faulty detects a fault and asks for help. In the *robot detection step* initiates the cooperation between the robots. Once the Faulty has detected a fault and it is not able to diagnose alone, since there is ambiguity according to the nature of the fault, it asks for help, and Helper starts looking for Faulty through the perception coming from its computer vision synthesizer node. The robot detection step finishes when Faulty is detected or a full turn is completed without detecting Faulty. In the *diagnosis step*, Faulty tries to perform the actions needed to disambiguate the diagnosis and, after that, it requests some feedback from Helper, that reports position and pose according to its beliefs, based on its own perceptions. Several runs were performed for each one of the proposed fault scenarios. The tests were recorded during theirs executions and the elapsed time for each step was extracted based on the logs.

The robots are positioned side by side and separated by a distance of 1.5 meters in order to give Faulty enough space to move according to the injected fault, and to create in Helper the need for movement to find Faulty. The faults are injected before executing the plan of the agent associated with the Faulty robot.

On the fault scenario where both actuators have faults, the injected fault was detected, Faulty robot was located by Helper and the cooperative diagnosis has successfully been done in all runs. Table I shows the value for each step of all runs. The mean of the total time is 22.42 seconds and the standard deviation is 0.78 seconds, which corresponds to less than 4% of the mean time.

	Fault detection step	Robot detection step	Diagnosis step	Total
Run 1	4.3 s	14.7 s	4.4 s	23.4 s
Run 2	4.0 s	14.2 s	4.7 s	22.9 s
Run 3	4.5 s	13.8 s	4.2 s	22.5 s
Run 4	4.1 s	12.4 s	4.9 s	21.4 s
Run 5	4.5 s	13.1 s	4.4 s	22.0 s

TABLE I: Latency on a diagnosis of faults in both actuators.

On the fault where Faulty tries to move but, differently from the previous fault scenario, the actuators worked properly, and the encoders do not detect any movement in the wheels, the injected fault was successfully detected by the Faulty robot in all runs, and Faulty robot was detected successfully by Helper robot in 85% of the runs. Table II shows the value for each step of all runs. From the successful runs, the mean of the total time is 25.55 seconds and the standard deviation is 10.97 seconds, which corresponds to approximately 43% of the mean time.

	Fault detection step	Robot detection step	Diagnosis step	Total
Run 1	7.1 s	7.4 s	7.0 s	21.5 s
Run 2	6.7 s	7.0 s	6.1 s	19.8 s
Run 3	6.6 s	8.7 s	5.7 s	21.0 s
Run 4	7.3 s	8.5 s	5.8 s	21.6 s
Run 5	7.3 s	62.3 s	+INF	+INF
Run 6	7.7 s	35.1 s	5.1 s	47.9 s
Run 7	6.7 s	8.8 s	5.9 s	21.5 s

TABLE II: Latency on a diagnosis of encoders faults.

In Table II, the high standard deviation is due to the fact that Helper, in run number 6, after detecting robot Faulty, required repositioning to keep Faulty close to the center of its visual field, preventing from visual loss. During the process of repositioning, Faulty was not perceived in the center of image and, as a consequence, Helper kept turning to centralize Faulty. Thus, despite Faulty's being at the center of the visual field, Helper continued believing that Faulty was at its right side and kept moving until Faulty was perceived in a wrong place. When Helper then, realized that Faulty was on its left side, it needed to reposition itself again. All the process of dealing with interaction faults during the robot detection step increased the time needed for Helper to get ready to help Faulty.

In run number 5, during the time which Faulty was within the Helper's field of view, Faulty was not perceived or Helper had an incomplete perception of Faulty, i.e., the colored cylinders which identify Faulty were not detected in the same image frame. Thus, Helper was not able to help Faulty.

On the fault scenario where Faulty detects the fault in one of the actuators, two experiments were done. In the first experiment a fault was injected in the left actuator, and after, in the right actuator. Beside being the same fault model, Faulty places itself in different locations, that could create different difficulties to the fault diagnosis process.

When the fault was injected in the left actuator, Faulty was located by Helper and the cooperative diagnosis was successfully done in all runs. Table III shows the value for each step of all runs. The mean of the total time is 30.25 seconds and the standard deviation is 1.52 seconds, which corresponds to 5% of the mean time. Besides the fault being detected in all runs, in run number 3 the diagnosis was based on a perception of a previous state of Faulty due to occlusion of the orange cylinder. In run number 4, the diagnosis also occurred based on a perception of a previous state of Faulty, due to the impossibility of depth extraction from the point cloud and, for consequence, pose estimation when Faulty reached the target position.

A similar situation happened in both cases when the diagnosis happened based on perceptions which did not represented the current state of Faulty. Once Faulty was positioned on the right side of Helper and Faulty had a fault in the left actuator, Faulty performed a soft turn to the left side, positioning itself in front of Helper. Faulty's pose, close to 90° in relation to Helper, caused the occlusion of the orange cylinder, preventing Helper from retrieving its depth.

	Fault detection step	Robot detection step	Diagnosis step	Total
Run 1	18.5 s	0.9 s	10.3 s	29.7 s
Run 2	19.2 s	1.6 s	11.3 s	32.1 s
Run 3	18.1 s	2.4 s	10.5 s	31.0 s
Run 4	17.8 s	1.4 s	9.4 s	28.6 s

TABLE III: Latency on a diagnosis of faults in the left actuator

When the fault was injected in the right actuator, despite similarities related to the behavior of the robots and fault characteristics, instead of positioning itself in front of Helper, Faulty performs a soft turn to the right side, moving away from Helper.

The injected fault was detected, Faulty robot was located by Helper, and the cooperative diagnosis was successfully done in all runs. Table IV shows the value for each step of all runs. The mean of the total time is 40.73 seconds and the standard deviation is 1.60 seconds, corresponding to less than 4% of mean time. Besides the fault being detected in all runs, in run number 3 the diagnosis was based on a perception of the previous state of Faulty, due to the non-detection of the orange cylinder caused by illumination issues.

	Fault detection step	Robot detection step	Diagnosis step	Total
Run 1	17.6 s	10.6 s	11.3 s	39.5 s
Run 2	18.9 s	9.2 s	14.2 s	42.3 s
Run 3	16.9 s	11.9 s	13.1 s	41.9 s
Run 4	15.6 s	10.8 s	12.8 s	39.2 s

TABLE IV: Latency on a diagnosis of faults in the right actuator

Tables III and IV show high latencies to the fault detection step once Faulty moves slowly due to the actuator fault. Low latencies in robot detection step (Table III) are a consequence of Faulty's movement to wrong direction, positioning itself in front of Helper when the fault is in the left actuator. When the robot detection steps began, Helper had already perceived Faulty. Differently from the previous scenario, the robot detection step has its latency time increased when the fault was injected in the right actuator (Table IV) since Helper performed a turn to locate Faulty.

VI. Conclusions

This paper presented a method of cooperative high-level fault diagnostic using an agent programming language. A case study is presented on two physical robots executing a cooperative fault diagnostic plan used to disambiguate the root cause of a fault.

Results presented in Section V show a success rate of 85% for the scenario of fault in the encoders and 100% for the remaining fault scenarios. Once the issues encountered during the tests are related to the computer vision techniques applied to locate and track Faulty and not the method by itself, we consider viable the development of better plans to diagnose faults as well as new plans to perform a compensation of the workload, to delegate tasks or even reconsider plans and goals. From the diagnosis step, it would be possible to increase robustness and fault tolerance in the system by triggering a recovery method or even using the presented method as part of a control loop.

As future work, we aim to improve the computer vision techniques to minimize interaction faults and to further evaluate the combined framework in various other fault situations and to assess the use of other multi-agent coordination techniques for applications with larger numbers of robots.

References

[1] A. Elkady and T. Sobh, "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography," *Journal of Robotics*, vol. 2012, pp. 1–15, 2012.

[2] P. Iñigo Blasco, F. Diaz-del Rio, M. C. Romero-Ternero, D. Cagigas-Muñiz, and S. Vicente-Diaz, "Robotics software frameworks for multi-agent robotic systems development," *Robotics and Autonomous Systems*, vol. 60, no. 6, pp. 803–821, Jun. 2012.

[3] J. Carlson, S. Member, and R. R. Murphy, "How UGVs Physically Fail in the Field," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 423–437, 2005.

[4] H. Bruyninckx, "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2001, pp. 2523–2528.

[5] E. Einhorn and T. Langner, "MIRA-middleware for robotic applications," in *Intelligent Robots and Systems (IROS)*, no. Iros, 2012, pp. 2591–2598.

[6] 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*, 2009.

[7] R. Bordini, J. Hübner, and M. Wooldridge, *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[8] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *Proceedings of the 7th MAAMAW*, ser. LNCS, W. V. de Velde and J. W. Perram, Eds. Springer-Verlag, 1996, vol. 1038, pp. 42–55.

[9] M. E. Bratman, D. J. Israel, and M. E. Pollack, "Plans and resource-bounded practical reasoning," *Computational Intelligence*, vol. 4, no. 4, pp. 349–355, 1988.

[10] D. Crestani and K. Godary-Dejean, "Fault Tolerance in Control Architectures for Mobile Robots: Fantasy or Reality?" in *National Conference on Control Architectures of Robots*, 2012.

[11] Y. Zhang and J. Jiang, "Bibliographical review on reconfigurable fault-tolerant control systems," *Annual Reviews in Control*, vol. 32, no. 2, pp. 229–252, Dec. 2008.

[12] N. Melchior and W. Smart, "A framework for robust mobile robot systems," in *Proc. SPIE 5609, Mobile Robots XVII*, 2004.

[13] D. Kirchner, S. Niemczyk, and K. Geihs, "RoSHA: A Multi-Robot Self-Healing Architecture⋆," in *RoboCup International Symposium RoboCup-2013*, 2013.

[14] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.