2014 IEEE/RSJ International Conference on
Intelligent Robots and Systems (IROS 2014)
September 14-18, 2014, Chicago, IL, USA

ReFrESH: A Self-Adaptation Framework to Support Fault Tolerance in Field Mobile Robots

Yanzhe Cui, Richard M. Voyles, Joshua T. Lane, Mohammad H. Mahoor

Abstract— **Mobile robots are being employed far more often in extreme environments, such as urban search and rescue, with greater levels of autonomy; yet recent studies on field robotics show that numerous failure modes affect the reliability of the robot in meeting mission objectives. Therefore, fault tolerance is increasingly important for field robots operating in unpredictable environments to ensure safety and effectiveness of the system. This paper demonstrates a self-adaptation framework, *ReFrESH*, that contains mechanisms for fault detection and fault mitigation. The goal of ReFrESH is to provide diagnosable and maintainable infrastructure support, built into a real-time operating system, to manage task performance in the presence of unexpected uncertainties. ReFrESH augments the port-based object framework by attaching evaluation and estimation mechanisms to each functional component so that the robot can easily detect and locate faults. In conjunction, a task level decision mechanism interacts with the fault detection elements in order to generate and choose an optimal approach to mitigating faults. Moreover, to increase flexibility of the fault tolerance, ReFrESH provides self-adaptation support for both software and hardware functionality. To our knowledge, this is the first framework to support both software and hardware self-adaptation. A demonstrative application of ReFrESH illustrates its applicability through a target tracking task deployed on a mobile robot system.**

I. Introduction

Robotic systems are playing increasingly important roles in hostile and unpredictable environments often encountered in space, military, and Urban Search and Rescue (USAR) operations [1]. Robots are appealing for such environments because they can be deployed both ahead of and in place of human beings and perform tasks that humans cannot. However, studies on mobile robots used in the field have shown a noticeable lack of reliability in real world conditions [2]. The worst case being that a robot becomes completely nonfunctional. Therefore, robots should be able to detect, isolate and even alleviate faults that result from both component failures and system configuration inadequacies.

Traditionally, to prevent or eliminate faults, system designers have adopted formal design and programming methods in the design phase as well as performed verification and validation (V&V) in the testing phase [3]. The alternative of fault tolerant is to identify faults dynamically and repair and validate them on the fly as a complement of V&V. Two such runtime fault tolerant techniques in the post-deployment phase have been proposed to increase the reliability of these robots: the first is to provide redundancy-based fault tolerance, achieved either by adding mechanical and sensory redundancy to a single robot or by deploying several identical robots at one time [4]; the second proposed solution is to support cooperation within a team of mobile robots of different functionalities so that task responsibilities may be redistributed in the event of an abnormal situation [5]. The work presented in this paper focuses on the latter method, applied throughout the framework at multiple levels in the hierarchy. Thus, the aim here is to develop a self-adaptive framework to support fault tolerance in cooperative field mobile robotic systems. Through dynamic self-adaptation, robots are capable of restoring themselves to normal functionality by distributing information and functionality across robot boundaries. Extending this concept further, the self-adaptation framework can detect faults on the fly and defective configurations can be altered to satisfy task requirements. We anticipate this framework to be used in conjunction with V&V to make systems highly tolerant of faults.

In order to generalize the framework to more easily adapt to a wide variety of robot structures, we developed a four-layer self-adaptation framework, termed *ReFrESH* (*Re*configuration *Fr*amework for distributed *E*mbedded systems for *S*oftware and *H*ardware). While ReFrESH appears to be a conventional layered architecture, it differs in that it provides diagnosable and maintainable infrastructure support, built into a real-time operating system (RTOS), to manage *functional requirements* and *non-functional requirements* across each robot boundary in the presence of uncertainties in the physical environment. In other words, ReFrESH explores a reflective view of self-adaptive systems where the non-functional services, software/hardware functional components executing in a kernel, runtime component management and dynamic self-adaptation within a robotic team are designed and implemented within the same paradigm: the Port-Based Objects (PBOs) [6], its real-time operating system (PBO/RT), and the Embedded Virtual Machine (EVM) [7].

An additional feature of ReFrESH is that it allows for easier integration of both software and hardware functionalities in field programmable gate arrays (FPGAs). To our best knowledge, ReFrESH is the first fault tolerant framework for robot applications that supports both software and hardware adaptation. We shall point out that a significant practical problem of distributing components across multiple robots through a network is communication connectivity, latency and jitter is currently being researched by ourselves [8] and by our collaborators at UPenn [7] and is beyond the scope

Yanzhe Cui is with the College of Engineering, Purdue University, West Lafayette, IN, 47907 USA cui56@purdue.edu

Richard M. Voyles and Joshua T. Lane are with the College of Technology, Purdue University, West Lafayette, IN, 47907 USA rvoyles,lane54@purdue.edu

Mohammad H. Mahoor is with the College of Engineering, University of Denver, Denver, CO, 80208 USA mmahoor@du.edu

of this paper. Also, the specific details of software and hardware self-adaptation methods are not included in this paper. Interested readers can refer to our previous publications [9] [10]. In this paper, we mainly present ReFrESH from the framework point of view.

II. Related Work

A great deal of research in robotics has focused on the development of architectures. Each architecture that has been developed for robotic systems tends to focus on providing the fault tolerance capability to the deployed robot or distributed robot team.

ALLIANCE [11] is a software architecture that facilitates cooperative control of teams of mobile robots for fault tolerance. It is a fully distributed, behavior-based architecture that allows each robot to select its appropriate action on the fly. [12] presents a self-adaptive robotic architecture that supports self-assembling components using a formal statement of high-level system goals. It is one in which components automatically configure their interaction in a way that is compatible with an overall architectural specification and achieves the goals of the system. [13] demonstrates an architecture-based self-adaptive system that focuses on supporting runtime change of adaptation policies that are decoupled from the architectures they relate to. Nevertheless, these architectures are not without their weakness: [11] does not support robotic self-adaptation; both [12] and [13] do not support distributed robotic teams; and all three approaches cannot migrate verified code modules that are not pre-compiled into an existing executable.

The architecture described in this work trades formal specifications of system behaviors for a higher degree of flexibility and supports runtime self-adaptation through migrating and loading components among all heterogeneous robots in a team. Therefore, to support this migration capability across robot boundaries, a virtual machine method is embedded into our architecture.

There exist some outstanding virtual machines (VMs) for migrating software. Scylla [14] is a traditional VM adapted for embedded systems whose interaction is assumed to be between an end-user and a single isolated node in a network. It is not designed as an interface among the nodes themselves. One physical machine exposes interfaces to a single logical machine. Mate [15] is a bytecode interpreter to run on multiple nodes. It is a more customizable solution about VM but it cannot adapt to component configuration dynamically. EVM [7] is being developed by our collaboration team, it consists the systems software around the embedded real-time operating system (RTOS) in each node, which facilitates the mechanisms to parametrically and programmatically control the operation of the node at runtime. It allows the control logic to dynamically assign tasks to controllers at runtime through task and network slot migration, partitioning and reallocation. But EVM can only migrate bytecodes among nodes. The architecture we propose makes up these limitations of the aforementioned VMs. It supports pre-compiled functional components (object code snippets) distributed among robots based on system and task specification.

III. Design Scenario

The mobile robots we used in this work were developed in our lab [16]. Small size brings limitations in actuators, sensors, computational power and battery life (for example, the Hokuyo LIDAR is too bulky for them). These limitations based on size force us to find new ways of accomplishing certain tasks: instead of having one robot, multiple robots should work together. This allows the requirements for actuators, sensors, and computational power to be split up between them. Therefore, to present the ReFrESH design concept explicitly, an example scenario of searching for survivors with multi-robot teams is presented.

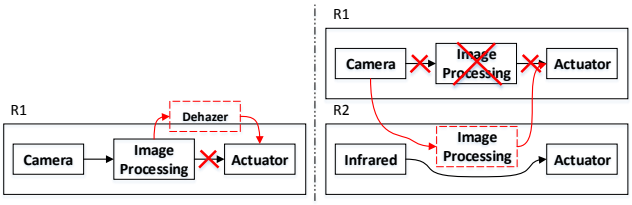

Fig. 1: Two considerations of ReFrESH design in this paper: left, single robot self-adaptation by loading a non-compiled functional component; right, multi-robot cooperation by migrating a functional component between them.

Two situations are considered in the design of ReFrESH: 1) As shown on the left of Fig. 1, as $R1$ uses its camera to detect a target, smoke or debris begins to degrade visual quality and the current pre-compiled image processing algorithm cannot handle this unexpected situation. Unfortunately, $R2$ does not equip a camera sensor. So to guarantee the search task is achieved successfully, using ReFrESH, $R1$ will undergo self-adaptation by loading a new incoming advanced image processing algorithm, such as a "Dehazer"; 2) As shown on the right of Fig. 1, as $R1$ uses its camera to detect a target, a failing battery threatens to kill the pre-compiled image processing algorithm. Though $R2$ has sufficient power to perform the computation, it has not pre-compiled this algorithm. To guarantee the search task is achieved successfully, using ReFrESH on the two robots, $R1$ will migrate the image processing component to $R2$, and $R1$ and $R2$ can construct a single cooperative system.

IV. ReFrESH Architecture

In this section, we describe the details of each layer and the relationship among the four layers of the ReFrESH framework as shown in Fig. 2. In order to present ReFrESH concisely and clearly, from this section on, we use the following terminology: "*Component*" in place of "*Functional Component or Functionality*" and "*Reconfiguration*" in place of "*Self-Adaptation*". Thus, a task consists of a configuration of **components** and an underlying state machine and the fault toleration process provides detection of abnormal

components or configurations and alleviation of faults by altering the task configuration through **reconfiguration**.

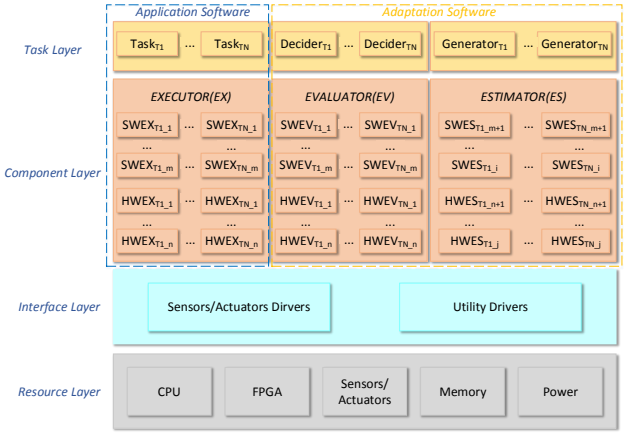

Fig. 2: ReFrESH 4-layer framework.

A. Resource Layer - Non-Functional Services

The port-based object (PBO) is a software abstraction for designing and implementing dynamically reconfigurable real-time software. This forms the basis of a programming model that uses domain-specific elemental units to provide specific, yet flexible, guidelines for integrating software components [6]. PBO specifies a mechanism for attaching meta-data to component assemblies through resource ports; such as the communication information among sensors/actuators and the resource information about other subsystems. These meta-data elements influence the task behavior by triggering the execution of non-functional services. For example, through a resource service to retrieve the capability of a node, it is possible to show and define the performance constraints of this node. Here, we choose the set *CPU, FPGA, MEMORY, POWER, SENSORS/ACTUATORS* as the characteristics of a node: the chosen "CPU" limits the maximum operation cycle and computation capability; the particular "FPGA" used decides the area limitation(how many hardware components could be loaded); "MEMORY" restricts the data stream size and update rate; "POWER" decides the life of a node; and equipped "SENSORS/ACTUATORS" limit the task execution capabilities. All essential services performed by this level are the provision of *non-functional requirements*.

ReFrESH proposes two innovative approaches to integrating such non-functional requirements into an application: 1) by implementing them as regular PBO components and 2) by providing an attachment mechanism for connecting these non-functional requirements to each running component. By using PBO components to satisfy non-functional requirements, we provide an integrated solution where there is only one paradigm for the implementation of a fault tolerant robotic system.

B. Interface Layer - Data Transfer Services

This layer provides the driver and interface means of providing reliable data(packet) transfer services to the layer above, the – *Component Layer*. Specifically, it provides a point-to-point link of sensors/actuators and their corresponding operational components and builds a transfer channel for requesting non-functional utility data.

C. Component Layer - Runtime Component Services

The most essential design target of ReFrESH is that it better supports fault tolerance. Thus, one of the cores of the ReFrESH framework is a component layer which supports elemental services for component management. This layer is required to support robotic systems that have an inherent need for runtime reconfiguration. ReFrESH allows for the customization of the execution policy associated with a series of running components, (*EXECUTOR (EX)*). For this framework, the PBO paradigm was augmented for hardware component implementation as well. Thus, in ReFrESH, a task could be composed of both software and hardware components that have a uniform inter-communication mechanism.

To provide composable real-time components, ReFrESH encapsulates elemental building blocks into augmented versions of real-time PBOs which are used to monitor and assess each *EX*. Additionally, to make a node capable of evaluating and optimizing its own task performance from the component perspective, ReFrESH creatively augments the component viewpoint by introducing an "Evaluator" (*EV*) and an "Estimator" (*ES*). *EV* is a standard subroutine, the same as *EX*, meaning that it runs continually as long as the component it is attached to exists in a configuration. By retrieving the non-functional requirements, *EV* assesses the performance of the running component. Conversely, *ES* is instantiated only if the current configuration has not satisfied the performance requirement and is used to assess the potentially satisfying, non-running alternative components. The incorporation of *EX,EV, and ES* is termed a *Super-PBO* to show the distinction from the original PBO concept.

D. Task Layer - Configuration Generating Services

The task layer takes charge of determining the necessity for fault alleviation, re-assembling components based on the requirements of a task, and generating an optimal configuration policy at runtime. Unlike a conventional task layer which only includes a state machine to control each component to satisfy the requirement of a task, such as by turning a component on/off; ReFrESH also extends the scope of the *Task* from the "execution" perspective to the "fault toleration" perspective by adding the "Decider" and "Generator" in order to support the capability of runtime reconfiguration.

To accomplish this, the "Decider" first requests all component related performance information from the *EV* to judge whether there is any reconfiguration requirement. If there is a need for reconfiguration, which signifies that one of the components or the configuration as a whole are not suitable for executing the current task, the "Generator" will be instantiated. The "Generator" executes in two phases: 1) dependency analysis, which shows the construction of a

configuration and decides if a component could be replaced or amended by another homogeneous functional component; and 2) functional assembly, which shows the process of combining required components to generate all potential configurations. Since ReFrESH is based on the PBO kernel and PBO/RT provides flexible interfaces to control each component, it is convenient to analyze the dependency and link the components in the configurations together. For these generated configuration candidates, the corresponding *ES* will be instantiated to estimate the performance of each component in each candidate for the current task which helps the "Decider" to produce an optimal configuration. The implementation details of each of the aforementioned basic services in this level will be illustrated in section V.

V. ReFrESH Fault Tolerance as A New Discipline

In Fig. 3 we show a framework schematic of the fault tolerant mechanisms contained in ReFrESH. *PBO/RT Interface* is modeled after the Chimera Port-Based Object interface for subsystems servers (SBS). It provides a series of control for components, such as spawn task (sbsSpawn), turn on/off component (sbsControl) and set/get parameters of components (sbsSet/sbsGet). The *Component Manager* monitors the input and output signals of functional components, as well as the system resource consumption, which it then compares to a high-level abstraction of the system. Specifically, the high-level abstracted view could trace the *non-functional requirement* as the cause of the fault; such as CPU usage, system power consumption, or memory usage, through the interface layer to the bottom level resource layer. The *Component Assembly Generator* will generate all possible configuration candidates based on the information obtained from the *Component Manager*. Finally, according to the *functional requirement* of the current task and the *non-functional requirement* of the system nodes, the *Configuration Decider* is able to choose the optimal configuration from the set of configuration candidates. This management unit is described in more detail in the following sections.

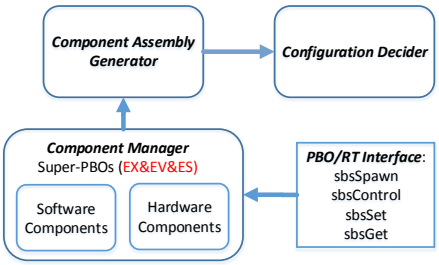

Fig. 3: The dataflow of ReFrESH supporting fault tolerance.

A. Fault Detection - Component Manager

1) Detecting Faults with Running Component Monitor - Evaluator: The *EV* monitors the performance of a specific component in the system to provide for the detection of faults. The *EV* generates discrete readings by sampling the signal according to a frequency set in PBO/RT; usually the same frequency as the *EX* it is attached to but this is user defined. A component is evaluated based on how well it implements its action in line with the demand of the overarching task as well as how effectively it uses system resources. In this paper, considering we want to support both software and hardware functional components, we chose a set of *non-functional requirements* in defining the non-functional performance of a component to be *FPGA Area, Memory, CPU and Power*. However, the user could add any parameter to this set. It is then up to the user to decide on which functional aspect of the component it will be evaluated, such as the error in a tracking algorithm on $R1$ in Section III, and the weight that it will hold against the non-functional performance. In particular systems a certain component may not be vital to the execution of a task and only provides some secondary functionality. In that case, it is more important to the health of the system that the component consumes minimal resources, therefore the functional performance is given less weight.

Alg. V.1 presents pseudo code that demonstrates the attachment of the *EV* to the *EX* with the use of PBOs in PBO/RT. To ensure real-time monitoring of the active component, we embed the evaluator initialization process in the component turn on function. This way, when the task activates the component, the component's evaluator will be activated as well.

Algorithm V.1: PBO Augmentation I - EV(i)

comment: Initialize and Run Component EX_i

for $i \leftarrow 0$ **to** N

do $\begin{cases} EX\_ID_i \leftarrow \\ sbsSpawn(EX\_init_i, Freq_i, fRealTime_i); \\ \\ sbsControl(EX\_ID_i, SBS\_ON); \end{cases}$

comment: Attach EV to EX in EX Turn On Function

for $i \leftarrow 0$ **to** N

do $sbsControl(EX\_ID_i, SBS\_EV)$;

comment: Evaluate Running Component

for $i \leftarrow 0$ **to** N

do $U_{comp(i)} \leftarrow p_f \cdot w_f + p_n \cdot w_n$;

2) Finding Fault Evidence: The utility of each component i, $U(Comp_i)$, changes dynamically based on system resource consumption and execution of their action and is calculated by the evaluator at runtime. We know how much of the resources a component in the configuration is expected to use (denote by $consumption_{ex,i}$), as this is set by the user during component creation. We also know the amount of resources the system node is capable of supplying (denote by $capability_{node}$). The spare resources available in the node with the current configuration are calculated by:

$$surplus_{node} = capability_{node} - \sum consumption_{ex,i}. \quad (1)$$

At runtime, we can request the resource information from the resource layer to measure how much of the resources the

component is actually using, $consumption_{actual}$. To determine the amount of surplus resources the component requires to continue functioning, subtract the expected consumption from the actual consumption.

$$consumption_{req} = consumption_{act} - consumption_{ex} \tag{2}$$

Then we can calculate how efficiently the component is utilizing the resource, $r(i)$, by:

$$u_{r(i)} = \frac{surplus_{node} - consumption_{req}}{surplus_{node}} \tag{3}$$

If the component requires no surplus of the resource it is performing as expected and so the utility of that resource for that component will be one. Intuitively, it is more important for the components to more efficiently use a resource with little surplus than a resource with a large surplus. Therefore, we assign weights to each resource by

$$w_{r(i)} = \frac{(1 - surplus_{node,r(i)})}{\sum(1 - surplus_{node,r(i)})} \tag{4}$$

where $surplus_{node,r(i)}$ is the surplus of resource $r(i)$ on the specified node. With these weight values, the non-functional performance, p_n, of each component based on resource consumption is calculated as

$$p_n = \sum(u_{r(i)} \cdot w_{r(i)}). \tag{5}$$

As was stated above, the utility of a component depends on both the functional and non-functional performance.

$$U_{comp(i)} = p_f \cdot w_f + p_n \cdot w_n \tag{6}$$

Here p_f is the functional performance of the component, w_f is the weight the user has given to the functional performance, and $w_n = 1 - w_f$. Now we have the utility of each individual component in the configuration. To determine the utility of the configuration as a whole, we simply take the lowest utility among the components,

$$U_{conf} = min(U_{comp(i)}). \tag{7}$$

B. Fault Mitigation I - Component Assembly Generator

1) Generating Configuration Candidates: The process for component assembly is based on component dependency analysis and functional assembly. A configuration is composed by instantiating a series of components and connecting their appropriate ports together. Component dependency defines the relationship between these components to show how their ports may be connected. The PBO concept provides a flexible component infrastructure based on port automata that makes component dependency analysis easy to perform. Each PBO has input, output, and resource ports and processes data from input to output as a result of a specific event. Communication between components is restricted to the input and output ports since the port names can be used to perform bindings. Therefore, with component dependency analysis we know which components in the system can be connected and in what manner. With this information it is straightforward to execute functional assembly. We have demonstrated the details for comprehensive component analysis and component runtime construction in [9], so we will not illustrate them again here.

2) Pruning Configuration Candidates: At this point, the *Component Assembly Generator* has created a list of every possible configuration based solely on dependency analysis. To reduce computation complexity in the final decision, a pruning procedure will shorten the candidate list based on the *non-functional requirements* of the configuration. This process is performed by considering the physical properties of the system nodes:

$$\begin{aligned} &Comp^{R}_{require} \leqslant Node^{R}_{remain} \\ &\qquad R \in \{Area, Memory, CPU, Power\} \\ &Comp^{Sensors/Actuators}_{require} \bigcap Node^{Sensors/Actuators}_{equip} = 1 \end{aligned} \tag{8}$$

C. Fault Mitigation II - Configuration Decider

With a minimal list of configuration candidates, the *Configuration Decider* activates the *ES* of the potential components to estimate the performance of each candidate in the execution of the current task under the current conditions. Alg. V.2 presents pseudo code that demonstrates the activation of the *ES* for use by the *Configuration Decider*. The *ES* performs the same action as the *EX* it is attached to, but the *ES* cycles only once and then evaluates its own performance. Therefore, the *ES* is essentially a non-running component monitor. Based on the estimated utility of each prospective component, one way to select the final configuration is to assemble the individual components each with the highest utility for a specific action as the best choice. But making this decision with only information local to the component may lead to a configuration that is globally sub-optimal. Another traditional method is to sum up all component utility values in each configuration and choose the configuration with the highest utility as the optimal choice. However, this method does not consider the characterization of the task and requirements of the user preferences.

Algorithm V.2: PBO AUGMENTATION II - ES(i)

comment: Call the ES from Configuration Decider
comment: i: configuration #; j: component #
for $i \leftarrow 0$ **to** N
  do for $j \leftarrow 0$ **to** M
  do $sbsControl(EX\_ID_{i,j}, SBS\_ES)$;

comment: Estimate Non-Running Component
for $i \leftarrow 0$ **to** N
  do for $j \leftarrow 0$ **to** M
  do $\begin{cases} EX\_ID\_cycle_{i,j}(); \\ U_{comp(i)} \leftarrow p_f \cdot w_f + p_n \cdot w_n; \end{cases}$

Additionally, weights of actions could be assigned to each action-related component, and the overall utility of each configuration could be obtained according to Eq. (9). Here

k belongs to each action type, such as "visual servoing" and "move" actions; i shows the components that can implement k; and $config_j$ is the jth configuration candidate. The optimal configuration is then the configuration with the highest value computed by Eq. (9).

$$U(config_j) = \sum_{k \in actions} U(Comp_i) \cdot W_k \qquad (9)$$

VI. Case Study on Multi-Robot Target Tracking

To demonstrate the feasibility and flexibility of ReFrESH, we demonstrate two situation we assume in Section III. In both situations, the task is to track a target with a camera. The first situation consists of a single robot executing this task while the second employs a team of two robots, call them $R1$ and $R2$. For both situations, the initial configuration is deployed solely on $R1$ and consists of a visual sensor component, a dehazing component implemented in hardware, a tracking algorithm component, and an actuator component. This initial configuration is shown in Fig. 4.

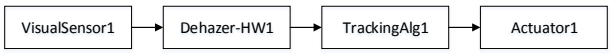

Fig. 4: Sample Application Initial Configuration

A. Single Node Runtime Fault Detection & Mitigation

In this situation,$R1$ executes a target tracking task. The robot is equipped with the four components contained in the initial configuration as well as a secondary dehazing component that is implemented in software. Each implementation of the dehazing component has an advantage over the other. The hardware implementation is able to process at a faster speed since the FPGA is not limited by the CPU, but the software implementation consumes far less power. The node initially loads the hardware dehazing component since the battery is fully charged and the system wants to process data as fast as possible for smooth operation. At the start of the task execution the field of view of the visual sensor is clear so the dehazing component is not needed and is therefore not active. Suppose after some time smoke blows in front of the visual sensor, obstructing the view of the target. Now the dehazing component needs to activate but by this time the battery has depleted so much that there is not enough power to activate the hardware implementation. Detecting this fault, the *Configuration Manager* sends a signal to the *Configuration Decider* to begin the reconfiguration process. Because the node acts alone, there are only two configuration candidates capable of executing this task; that using the hardware dehazing component and that using the software dehazing component. Because the software component consumes less power, it is intuitive that the decider will choose this component for the new optimal configuration. The new configuration, shown in Fig. 5, is initialized and the fault detected at runtime is mitigated.

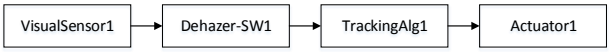

Fig. 5: Sample Application Single Node Reconfiguration

Fig. 6 depicts the performance of the system throughout this process based on the error in the tracking algorithm. We can see that the smoke blows in at around the 4.4 second mark where the error exceeds the threshold. At this point the robot reconfigures the task to use the software dehazing component and the error very quickly drops back below the threshold. The process for self-reconfiguration in this situation is very short since there are only two configuration candidates for the decider to consider and all of the components are contained on a single node so there is no need for lengthy data communication.

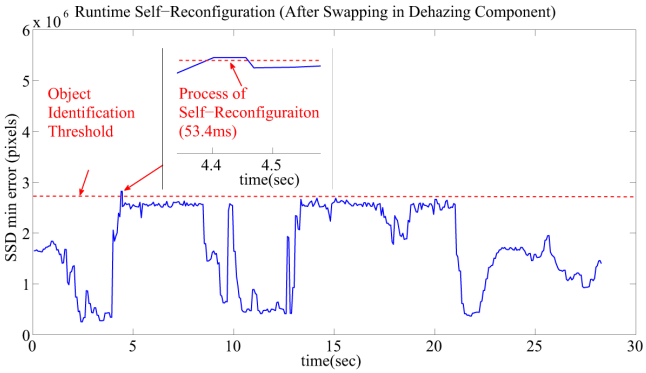

Fig. 6: Single Node Reconfiguration Performance Based on Tracking Error

B. Multiple Node Runtime Fault Detection & Mitigation

The use of multiple nodes in the proposed self-adaptation framework allows for more configuration candidates thus ensuring greater performance and reliability. For this situation, we suppose that the first node no longer contains a secondary software implementation of the dehazing component, but instead it can cooperate with a second node which does contain one. The second node is also equipped with all of the components utilized in the initial configuration as well. Once again as smoke blows in and the visual sensor is obstructed, the hardware dehazing component needs to activate. But once again, by this point there is not enough power on node one to activate it and the fault is reported to the *Configuration Decider* so that reconfiguration can begin. In this case there are many more candidates to consider since the two nodes can cooperate and share information, but because power consumption is still the main concern, the *Configuration Decider* will select the optimal configuration as the one that substitutes the software dehazing component on node two for the hardware dehazing component on node one. This configuration is shown in Fig. 7.

Fig. 8 depicts the performance of the multiple node system throughout this process based on the error in the tracking

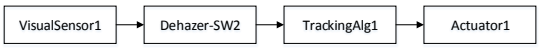

Fig. 7: Sample Application Multiple Node Reconfiguration

algorithm. Again the smoke blows in at around the 4.4 second mark where the error exceeds the threshold and node one begins the reconfiguration process. This time, because there are many more configuration candidates to consider and because the two nodes must communicate the estimated performance values of their components, the reconfiguration process takes much longer. During reconfiguration we can see that the error in the tracking algorithm remained high since the node was unable to activate a dehazing component. However, once the system reconfigured and redistributed task responsibilities, the error dropped below the threshold and the task was executed effectively once again.

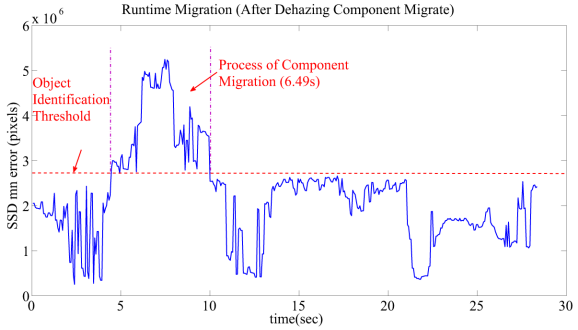

Fig. 8: Multiple Node Reconfiguration Performance Based on Tracking Error

This is a basic application where the ReFrESH framework proves effective in the detection and mitigation of a fault. This application can be extended to the complex visual servoing scenario depicted in Section III with the addition of several components, but we show a simplified case here to demonstrate the fault tolerance process by using reconfiguration.

VII. Conclusion

This paper demonstrates a self-adaptive framework, *ReFrESH*, which supports fault tolerance for systems of field mobile robots. *ReFrESH* contains mechanisms for fault detection and fault mitigation, built into a real-time operating system, to manage task performance in the presence of unexpected uncertainties in the field. Moreover, ReFrESH provides self-adaptation support for both software and hardware functionality. By augmenting the port-based object through the attachment of a component *Evaluator* and *Estimator*, ReFrESH enables a robot to autonomously detect and locate faults. Furthermore, to mitigate the effects of a fault at runtime, ReFrESH contains a series of mechanisms to generate a new optimal task configuration across the robot boundaries under the structure of an embedded virtual machine. Through an application involving runtime fault detection and fault mitigation by coordinated adaptation among two autonomous mobile robots, we have demonstrated the utility of ReFrESH for fault tolerance in robotic systems.

Acknowledgment

This work was supported by National Science Foundation grants CNS-0923518, CNS-1138674 and IIS-1111568 with additional support from the NSF Safety, Security and Rescue Research Center.

References

[1] K. Nagatani, S. Kiribayashi, Y. Okada, K. Otake, K. Yoshida, S. Tadokoro, T. Nishimura, T. Yoshida, E. Koyanagi, M. Fukushima, and S. Kawatsuma, "Emergency response to the nuclear accident at the fukushima daiichi nuclear power plants using mobile rescue robots." *J. Field Robotics*, vol. 30, no. 1, pp. 44–63, 2013.

[2] J. Carlson, R. Murphy, and A. Nelson, "Follow-up analysis of mobile robot failures," in *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, vol. 5, April 2004, pp. 4987–4994 Vol.5.

[3] J. Schumann, T. Mbaya, O. Mengshoel, K. Pipatsrisawat, A. Srivastava, A. Choi, and A. Darwiche, "Software health management with bayesian networks," *Innovations in Systems and Software Engineering*, vol. 9, no. 4, pp. 271–292, 2013.

[4] R. a. Carrasco, F. Núñez, and A. Cipriano, "Fault detection and isolation in cooperative mobile robots using multilayer architecture and dynamic observers," *Robotica*, vol. 29, no. 4, pp. 555–562, Jul. 2011.

[5] F. Tang and L. Parker, "Coalescent multi-robot teaming through asymtre: a formal analysis," in *Advanced Robotics, 2005. ICAR '05. Proceedings., 12th International Conference on*, 2005, pp. 817–824.

[6] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *Software Engineering, IEEE Transactions on*, vol. 23, no. 12, pp. 759 –776, dec 1997.

[7] M. Pajic, A. Chernoguzov, and R. Mangharam, "Robust architectures for embedded wireless network control and actuation," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 4, pp. 82:1–82:24, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2362336.2362349

[8] M. Ayad, J. Zhang, R. Voyles, and M. Mahoor, "Mobile robot connectivity maintenance based on rf mapping," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, Nov 2013, pp. 3398–3405.

[9] Y. Cui, R. Voyles, M. He, G. Jiang, and M. Mahoor, "A self-adaptation framework for resource constrained miniature search and rescue robots," in *Safety, Security, and Rescue Robotics (SSRR), 2012 IEEE International Symposium on*, Nov 2012, pp. 1–6.

[10] M. He, Y. Cui, M. Mahoor, and R. Voyles, "A heterogeneous modules interconnection architecture for fpga-based partial dynamic reconfiguration," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, july 2012, pp. 1 –7.

[11] L. Parker, "Alliance: an architecture for fault tolerant multirobot cooperation," *Robotics and Automation, IEEE Transactions on*, vol. 14, no. 2, pp. 220–240, Apr 1998.

[12] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 259–268.

[13] J. C. Georgas and R. N. Taylor, "Policy-based self-adaptive architectures: A feasibility study in the robotics domain," in *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, ser. SEAMS '08. New York, NY, USA: ACM, 2008, pp. 105–112. [Online]. Available: http://doi.acm.org/10.1145/1370018.1370038

[14] P. Stanley-Marbell and L. Iftode, "Scylla: a smart virtual machine for mobile embedded systems," in *Mobile Computing Systems and Applications, 2000 Third IEEE Workshop on.*, 2000, pp. 41–50.

[15] P. Levis and D. Culler, "Mate: A tiny virtual machine for sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 85–95, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/635508.605407

[16] R. M. Voyles, , and A. C. Larson, "Terminatorbot: A novel robot with dual-use mechanism for locomotion and manipulation," *IEEE/ASME Transactions on Mechatronics*, vol. 10, pp. 17–25, 2005.