Fault tolerant framework and techniques for component-based autonomous robot systems

Conference Paper · January 2011

DOI: 10.1145/1982185.1982307 · Source: DBLP

4 authors, including:

Heejune Ahn
Skolkovo Institute of Science and Technology
52 PUBLICATIONS **358** CITATIONS

SEE PROFILE

Sang Chul Ahn
Korea Institute of Science and Technology
120 PUBLICATIONS **926** CITATIONS

SEE PROFILE

Sung Yun Shin
South Dakota State University
85 PUBLICATIONS **157** CITATIONS

SEE PROFILE

Fault Tolerant Framework and Techniques for Component-Based Autonomous Robot Systems

Heejune Ahn
SeoulTech
Seoul, Republic of Korea
+82-2-970-6543
heejune@seoultech.ac.kr

Sang Chul Ahn
IMRC, KIST
Seoul, Republic of Korea
+82-2-958-5777
asc@imrc.kist.re.kr

Junyoung Heo
Hansung University
Seoul, Republic of Korea
+82-2-760-8039
jyheo@hansung.ac.kr

Sung Y. Shin
South Dakota State Univ.
Brookings, SD 57007 USA
+1-605-688-6235
sung.shin@sdstate.edu

ABSTRACT
Due to the benefits of its reusability and productivity, the component-based approach has become the primary technology in service robot software frameworks, such as MRDS (Microsoft Robotics Developer Studio), RTC (Robot Technology Component), ROS (Robot Operating System) and OPRoS (Open Platform for Robotic Services). However, all the existing frameworks are very limited in fault tolerance support, even though the fault tolerance function is crucial for the commercial success of service robots. In this paper, we present a rule-based fault tolerant framework together with widely-used, representative fault tolerance measures. With our observation that most faults in components and applications in service robot systems have common patterns, we equip the framework with the required fault tolerant functions. The system integrators construct fault tolerance applications from non-fault-aware components by declaring fault handling rules in configuration descriptors or/and adding simple helper components, considering the constraints of the components and the operating environment. Much more consistency in system reliability can be obtained with less effort of system developer. Various fault scenarios with a test robot system on the proposed OPRoS fault tolerant framework demonstrate the benefits and effectiveness of the proposed approach.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: Fault Tolerance

General Terms
Management, Measurement, Reliability, Experimentation.

Keywords
Service robot, fault-tolerance, framework, component-based design.

1. Introduction
Service robots have recently gained an increasing interest in research and industry. Carnegie Mellon University's Minerva, Honda's Asimo, Sony's AIBO, and iRobot's Roomba are just a few examples of evidence for the interest [1]. Service robots are complex computer and control systems consisting of a number of integrated hardware and software modules, such as sensors, actuators, controllers, signal processing, artificial intelligence, and human-interface algorithms. Furthermore the robot's modules cooperate to achieve specific tasks. Due to their need for tight integration with the physical world and their unique characteristics, service robots in general pose considerable impediments and make the development of robotic applications non-trivial.

To overcome these integration difficulties in robot system development, various proposals for robot middleware or frameworks have been proposed, such as MIRO, Orca, UPnP middleware, RT-middleware, and OROCOS. More recently the component-based design is the trend of robot software frameworks, due to the benefits of modularity, reusability, and productivity. Fig. 1 shows the typical system structure of component-based robot systems, illustrating the components as Lego blocks and the framework as a mainframe. In addition to providing specific application functions, the components follow a well-defined design pattern and interface so that the framework can control the lifecycle, execution, and data-passing of the components. The representative systems include OMG's RTC (Robot Technology Component) [2], Microsoft's MRDS (Microsoft Robotics Developer Studio) [3], ROS (Robot Operating System) [4], and OPRoS [5, 6, 7].

On the other hand, for the commercial success of intelligent service robots, fault tolerance technology for system reliability and human safety is crucial [8, 9]. This is because mobile service robots operate with moving mechanical parts in the human working space. Since fault tolerance operations are typically a system or application level function, it is extremely difficult, if not impossible, to design a fault tolerance procedure at the component development stage. In other words, component developers have limited knowledge of the integration and operating environment of the component. Therefore, the framework must support fault tolerance service as a framework service. However, to our knowledge, the component-based robot software frameworks mentioned above do not yet provide an appropriate level of fault tolerance support. This lack in the existing frameworks in fault tolerance is mainly due to the belief that fault tolerance is application-dependent and the framework is application-independent in nature.

In this paper, we show that a rule-based fault tolerant framework can automate fault tolerance instrumentation considerably well in component-based robot systems. We draw common patterns in faults in components and applications in service robot systems

SAC'11, March 21-25, 2011, TaiChung, Taiwan.

and equip the framework with desired fault tolerant functions. The system integrators merely have to declare fault handling rules in configuration descriptors and/or add simple monitor components

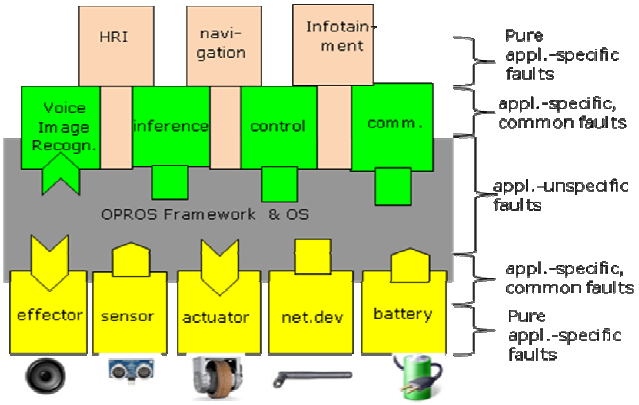


Fig. 1. Typical System Structure of Component-based Robot Systems

considering the constraints of components and the operating environment. Thus, the proposal does not require changes in existing application components and application design. To demonstrate the feasibility and benefits of the proposed approach, we implement a fault tolerant OPRoS framework and test robot systems and expose the system to various fault scenarios.

The remainder of the paper is organized as follows. Section 2 provides a brief description of OPRoS, as a typical example for component-based robot software frameworks. In Section 3, we propose the fault tolerant middleware architecture and integrated fault tolerant techniques. Section 4 describes the test robot system and test application scenarios. In Sections 5 and 6, with various fault examples, we demonstrate how the well-known fault tolerant tools are integrated in the proposed architecture. Section 6 concludes this paper.

2. OPRoS: A Component-based Robot Framework

A detailed description of OPRoS standards might be required for an implementation-level understanding of our proposed fault tolerant architecture and mechanisms, but readers can understand the overall operation using Fig. 2 and the following quick summary. For a detailed description, readers should refer to the specification [5], overview paper [6], and the OPRoS project website [7], where one can download the framework implementation source, sample components, utilities, and presentation materials.

The OPRoS robot system consists of an OPRoS framework engine and application components. The OPRoS components provide application specific services, e.g., sensors, actuators, and various algorithm components. The OPRoS framework runs as a process in an operating system and provides the execution, lifecycle management, configuration, and communication services to the application components. The framework manages the lifecycle of the components, such as loading/unloading the component library and creating/destroying an instance, the state of the components using lifecycle interface functions, initialize(), start(), stop(), destroy(), recover(), update(), and reset(). The framework also executes the components' functions by invoking the callback functions, that is, onExecute() and onEvent(), defined in the user components.

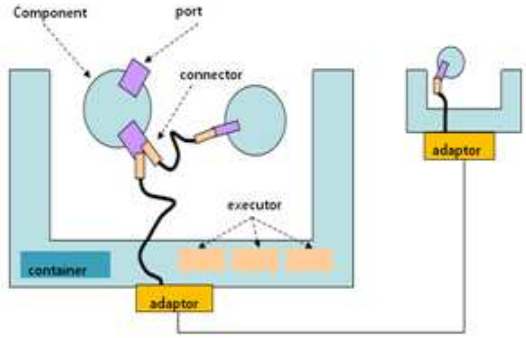


Fig. 2. The OPRoS Framework and Component Model

A user defined OPRoS component class inherits the basis class'Component' and overrides the callback functions. When its callback functions are called, a component can execute its jobs and communicate data with other components only through ports. OPRoS ports are classified as 'data', 'event', and 'service' according to the synchronization and argument styles. For some cases, multiple atomic components can be merged into a 'composite' class, which is again treated as a component.

For integrating the system, the component configuration information, such as the component name, port name, port type, and execution type, is provided in an XML file called 'component profile'. The components are grouped into application tasks in another XML file called the 'application profile'. The application profile provides the port connection information and initialization values for the components. The framework reads the application and component profiles to initialize, connect, and run the components. The connectors and adaptors for the communication middleware in Fig. 2 have little to do with the subject of this paper, so we will give no further description here.

3. Fault Tolerant Framework Design

3.1 Design Motivation

Our rule-based fault tolerant framework is inspired by the observation that most faults and fault tolerant techniques in service robots have common patterns. When the typical component-based robot system is introduced in Fig. 1, the three categories of faults are marked together. The application-unspecific faults include memory access errors, crashed, or deadlock. The application-specific faults are logical faults and misbehaviors. However, as you have probably experienced in a car-repair shop, most faults of components can be easily detected by examining the input-output ranges and responses, so we categorize the application specific faults once again into common and pure-application specific faults.

3.2 System Architecture

Our fault tolerant framework architecture can be described in two parts: the internal structure of the fault manager and its interaction with other modules in the robot system. Fig. 3 illustrates the fault manager internal structure with the fault processing flow. Note that the fault manager is itself a component, a so-called system

component. Therefore, the fault manager can receive/send fault events from/to user components through input/output event ports. In addition to these user level fault events, the fault manager can receive fault status information from other parts of the framework, such as ports, executors, and other managers. Once a fault event is received, the fault is diagnosed and handled according to the configuration files. Since the configuration files of OPRoS are XML files, the fault configuration is defined as XML element extensions. The currently implemented extension will be described in Section 5 with application examples.

Fig. 4 illustrates this fault event flow and interfaces with other parts. The 'bomb' markers symbolize fault detections, and the arrows are used to denote the fault event propagation flow. Most of the runtime exceptions are detected at executors, and the configuration information is obtained through the component manager.

3.3 Employed Fault Tolerant Techniques

Table 1 summarizes the fault tolerant techniques employed in our fault tolerant framework extension. These techniques encompass most well-known fault tolerance techniques both in the computer and in control engineering literature [10, 11]. Fault tolerance processing is divided into three stages: fault detection, fault diagnosis, and fault handling. The detailed application is described in Section 5 after introducing the test service robot system.

Table 1. Fault Tolerance Technologies Employed

Function	General Fault Tolerance Technique	Supported Method in Our Work
Fault Detection	OS exception	Structured Exception Handler in Win32 Signal-setjmp-longjmp in POSIX
	signal model	Range checker at port input/output
	Process model	Sliding window decision filter
	watch dog	Heat beat monitor thread
Fault Diagnosis	System State Estimation	Not supported yet
	Fault isolation	Severity and dependency tags and graph
Fault Handling	Reset	Life cycle callback
	Check pointing	Periodic backup thread
	Replacement	Alternative component tag
	Fault Stop	Based on severity and dependency tags and graph
	N-versioning	No frame support (better implemented using composite component model)

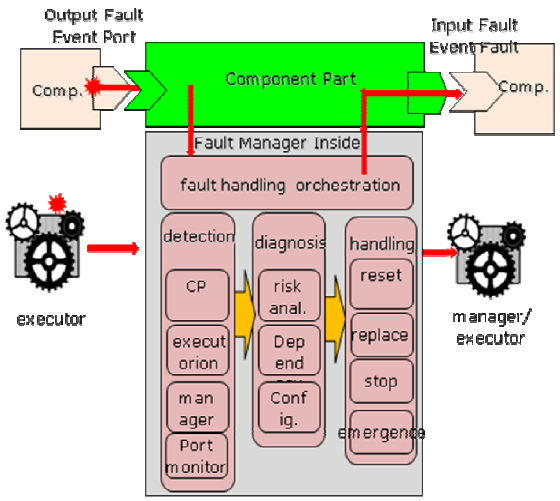


Fig. 3. Internal Structure of the Fault Manager and Fault Process Flow

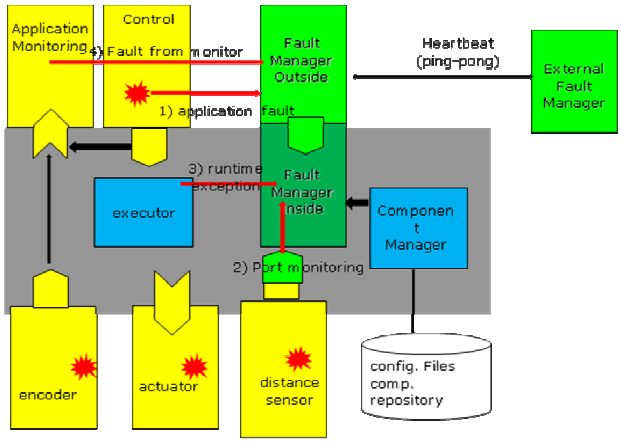


Fig. 4. Fault Propagation in the Proposed Framework

4. Test Service Robot System

Since a desktop Windows, Linux, and embedded Linux system are the most widely used systems, various tests with the fault tolerant framework have been performed on all three types of operating environments. In particular, to learn from a realistic robot operating environment and fault scenarios, we implemented a typical embedded robot system. Table 2 is the summary of the system specifications. The main board runs the OPRoS framework and controls the motion board. The AVR microprocessor on the motion board controls four DC motored wheels and checks the ultrasonic and encoder sensors. A control protocol from the main board to the motion board is defined in Table 3.

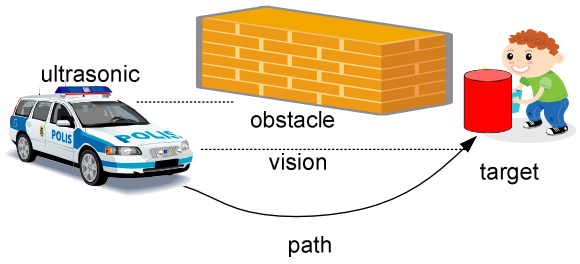
The main test application scenario is a navigation task essential for mobile service robots. The application consists of two independent sub-tasks, path planning and obstacle avoidance (Fig. 5). The color object detection algorithm is based on the algorithm used for the robot soccer world-cup [12]. The path-planning task consists of one vision sensor component, one object detector component, one path planning algorithm component, and one actuator control component. The path-planning component

receives the obstacle location from the obstacle-avoidance task and obtains the target location information its own way, then builds a moving path for the target. Finally, the motion decision made at the path-planning component is sent to the actuator component for the DC motor control.

For the obstacle avoidance subtask, periodically captured ultrasonic data are filtered at the obstacle avoidance component. The final obstacle information is sent to the path-planning component. The obstacle avoidance task uses two different obstacle avoidance algorithm components, one for primary and the other for secondary backup. When there are no faults, the robot reaches the target, avoiding a collision with the obstacles.

Table 2. Fault Tolerance Technologies Employed

Part	Subpart	Spec. and Function
Main Board	CPU	PAX272-520 MHz
	Memory	16 MB flash ROM, 64 MB DRAM
	OS	Linux 2.6.12
	Peripheral	CCD Camera 3 UARTs
Motion Board	Body	1.3 kg, 210x140x500 mm
	CPU	Atmega128, no OS
	Sensors	4 ultrasonic sensor (3 cm-3 m)
	Motors	4 DC motors PWM control
	Encoder	4 encoders (one for each motor)



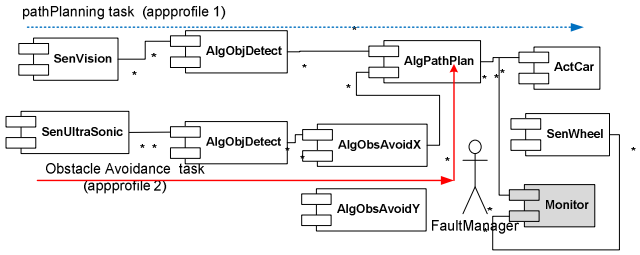


Fig. 5. Basic Operating Application Scenario: Target Following with Obstacle Avoidance

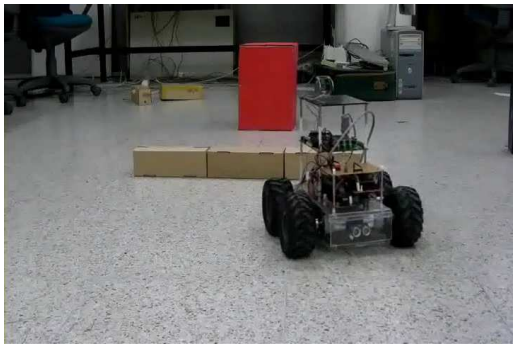


Fig. 6. Snapshot of Test Scenario

Table 3. Protocol between Main and Motion Board

Message	Field	Linux-PXA272
from main board to motion board	motion_type	0 (stop), 1 (forward), 2 (backward), 3 (turn-left), 4 (turn-right),
	parameter	rpm / rpm difference
from motion board to main board	encoders	left & right encoder value
	sensors	4 sensor values

5. Experiment Results: Functional

This section describes the proposed fault tolerant frameworks with simulated fault examples. On top of the components and application configuration in Section 4, fault tolerance tools and configurations are added, and then various possible faults are intentionally injected.

5.1 Fault Detection

Fault detection is usually the most difficult and important step for fault processing. We integrated the following fault detection mechanisms into the OPRoS framework. As will be described in this subsection, the executors, ports, and monitor components perform fault detections. On the fault detection, the fault events are informed to the fault manager through fault events or an internal fault manager interface function.

5.1.1 Runtime Software Exception Detection

Lee [13] reports an intensive experimental result on software errors and their propagation patterns. Most runtime software exceptions, such as a segment fault divided by zero, are caused by coding bugs. Furthermore, the most frequent sources of run-time errors are memory access errors, so-called pointer errors, such as de-referencing errors from invalid pointer variables, buffer bound overflow, and memory leaks. Most memory faults start with un-initialized pointer and index overflows.

The exception handling methods in modern operating systems can detect many runtime exceptions. Specifically, we implemented 'sigsetjmp()' and 'longjmp()' on a POSIX system [14], and SHE (structural exception handling), i.e., '__try &__except' on a Microsoft Windows system [15]. First, the exception handler is setup with a stack frame reserved before calling the component callback function. When a runtime exception occurs, the program control flow is jumped to the exception handler. Finally, the exception is reported to the fault manager. No XML extension is

defined for runtime exceptions at present, because classification and custom handling is not yet considered.

5.1.2 Signal Model-Based Fault Detection

In signal model-based fault detection, the component developers or integrators provide rules for checking the validity of the input and output values of ports. The range is described using the XML tag <validity>, which has sub elements of <min> and <max>. Then, the ports are setup by the fault manager at port connection time and monitor the data passed through the ports. Broken hardware and short/open circuits are known as the most frequent faults. Though logical errors cannot be detected easily without knowing the logic inside of components, it is possible to check the range of values and parameter type/number mismatch.

The signal model-based technique is used to detect errors the ultrasonic sensors in the test system. With the fact that the valid output range of the sensor is from 3 cm to 3 m (even though the object is at infinity), unplugging the sensor from socket (broken sensor emulation) results in output value '0'. With the configuration of 'min = 3' cm and 'max = 3000' cm, the port immediately detected the fault.

5.1.3 Process Model-Based Fault Detection

A process model based technique [15] uses models of the system components and plant (such as environments), and then compares the simulation results with the measured values. When the difference is larger than a certain threshold, it is considered to be a fault. Generally, the model can be very complex, dynamic, and even stochastic, so applying a process model requires detailed information about the component and sophisticated system identification and intelligent decision algorithm. We can also develop a fully comprehensive process model for the proposed architecture. Instead, a simplified but still powerful approach is integrated. A reference model component is added for one or a sequence of components. The same input data is provided to the target component and reference model component, and the output is directed to the fault manager. There, the filtered outputs with sliding windows period 'T' are compared. When the difference of filtered outputs becomes bigger than a threshold 'D', the fault event is sent to the fault manager.

5.1.4 Voting Method

The 'voting' mechanism or 'multiple versioning' is maybe the only mechanism for fault tolerance when we do not have either a system or a signal model for a component. The voting mechanism combines a fault detection and a fault handling procedure in one step. Three sonar sensors can be used for sensing distance from obstacles. A fault tolerant composite component is composed of a voting component and the sensor components. The voting component can select a median value from three inputs or applying a more intelligent filter to the inputs. When a fault is detected from a component, the voting component sends a fault event to the fault manager.

5.2 Fault Effect Analysis and Fault Isolation

All the detected faults discussed above are transferred to the fault manager. We should understand the properties, importance, and influence of a component and its faults. Based on the fault severity level <severity> ('ignore', 'reset', 'stop', 'emergency') in the configuration file, different fault handling is performed. The coverage of faults is also categorized into component, executor (thread or task), and system level using the <dependency> tag. This is how the proposed mechanism provides fault-isolation and containment checking.

A fault in one component needs to stop the application that it is in. Stop, i.e., the fault of an application again requires the stop of other applications that depend upon the fault application. The dependency graph model in [16] was adopted for a solution method. We define and express this dependency with <dependency> tags in system profile and application profiles. For simplicity, we assume that the components and application do not have circular dependency. Fig. 7 is an extension application configuration of our prototype robot system, adding HRI applications. There are three kinds of components with component duplication. Red components are faulty. Therefore, application A can run in spite of two faulty components in type B, but application B cannot run any more. Again application D, dependent upon the service of application A, can run, while application C, dependent upon application B, cannot run. In our prototype, the navigation components do not command movement when the obstacle detection application is faulty/stopped, which is caused by an ultrasonic sensor error. The scenarios are tested and verified with our prototype robot.

5.3 Fault Handling

Based on the fault diagnosis, fault handling is done in three different ways. Fault-Recovery (self-healing): whenever possible, the recovery of a fault component should be done fast enough for real-time operation. The component recovery mechanism has two types, component resurrection and component replacement, which are briefly detailed below. Fault-Stop: when a fault cannot be recovered, the relevant components in the system should also be checked, so that the fault containment region is minimized. When the corresponding application (or whole system) to faulty components (or application) can run without it, the application (or system) has to keep running. Fault-Safe: when a faulty component cannot be recovered and the corresponding application keeps running, the system may harm human beings or their environment, so the system must perform an emergency stop to prevent adverse effects to human beings.

5.3.1 Component Resurrection

Component Resurrection is a technique to recover and reuse the fault component. Most stateless components and often even some statefull components in robot applications can be resurrected by reset. Three callback functions, onError(), onReset(), onRecover(), are called when an error occurs and the platform tries to reset the faulty component and then the component is recovered, respectively. However, some statefull components do need state-recovery. A check pointing mechanism is implemented in the OPRoS framework. The component should call a framework service function, called 'critical (void *addr, int len)', where the first input argument 'addr' is the address for memory backup, and the second input argument 'len' means the size of the memory buffer. The periodic backup and recovery of faults are done by the framework automatically.

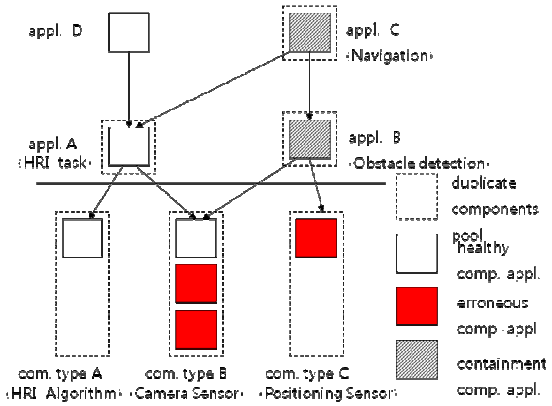


Fig. 7. Dependency Graph: Fault Isolation

5.3.2 Component Replacement

The so-called 'recovery block' component replacement technique is the most important fault tolerant tool. When the executor detects that the fault cannot be overcome by reset and it is a serious component for the task level, it is reported to the fault manager. The fault manager checks the application configuration file to determine whether a secondary component is prepared by the integrator. When it finds the alternative component, the fault manager loads its dynamic library and passes the component to the executor for replacement.

The reset, checkpoint, component replacement techniques are tested and verified using simulated obstacle detection component errors.

5.3.3 Emergency Stop

Because the robot is a mechanical system, the fault isolation and fault stop cannot guarantee fault safety. For example, the faults at a sensor can stop the navigation component, but the robot is moving according to the previous command. These risks are well pointed out in ISO safety standards [8]. We break our rule that the framework does not know the specific application of components and just follow the rule. The fault manager takes special care for the actuator components. When the monitor components of encoder and sensors report the fault to the fault manager, the manager sends a special fault event, the 'emergency stop', to the actuator. Then the actuator changes its operation mode from 'normal' to 'emergency stop'. The first action of the actuator on the emergency stop is to place the motors into a safe condition, i.e., 'stop in neutral'.

6. Experiment Results: Real Time Performance

In addition to the functional verification, we measured the fault detection and recovery time to check the real-time performance. In this paper, the fault detection time is defined as the elapsed time from fault manifestation to the instant of detection at the framework or monitor components, excluding the fault dormant time. A comprehensive study could not be performed due to the diversity of faults in robot systems. However, since process-model detection is considered to take a longer time than the runtime exception and signal boundary fault detection, we show the real-time performance of a motor driver fault. Fig. 6 shows the encoder rpm values when the driver power of the left wheels is switched off at 1000 ms. The encoder values are not dropped to 0, because of physical inertia and the motion of the right wheels. The sliding windows filter at the monitor component begins to generate output values less than the threshold of 0.5 reference values with more than 9 encoder samples, and it is declared as a fault at 1800 ms, around 200 ms later.

We also examined the fault recovery performance. Table 3 shows the recovery delay times with various system load conditions on desktop Windows, desktop Linux, and embedded Linux systems. The runtime and logical exception handling took a few milliseconds in a moderate system load condition. However, a secondary component replacement procedure often took over several hundred milliseconds when the load of tasks was over 80% CPU computing power or memory usage. It is not an unusual operating condition in an embedded robot system because of its resource limitations and heavily loaded vision processing. Our analysis revealed that the cause and performance patterns originated from the component loading time. The big difference between the Windows system and Linux is due to the large 'DLL' file due to the back (from the component to the based class) reference, which should be optimized. No significant difference between the 'dlopen()' option RTLD_NOW/LAZY was found. The difference between a desktop Linux and embedded target is due to the slow flash ROM.

Two methods are considered to overcome this problem. The first solution is to maintain the computation and memory load of the robot system under a certain level, for example, 80%. The second solution uses component pooling/preloading, which loads the secondary components before a fault occurs. We prefer the pre-loading approach. With this solution, we could manage the fault recovery time within 20 ms in any case we tested. The sizable recovery latency observed under the heavy load condition has been resolved using our pre-loading technique. Though the deadline measurement is not performed in this work, the robot continues its motion without any noticeable break. Therefore, it is considered that the recovery time also satisfies the real-time requirement.

7. Conclusion

The component-based approach is adopted in most recent robot middleware or frameworks to solve the software and system reusability problems. However, no or very limited support of fault tolerance is found in most of the proposed component-based robot software platforms.

In this paper, we presented a fault tolerant architecture that provides fault tolerance functions at system integration time with off-the-shelf non-fault-aware components. In addition to the fault event propagation mechanism, generalized model-based faults are handled automatically in a rule-based manner through the system configuration files. The prototype system demonstrated how well and easily the typical fault tolerance tools can be accommodated. Furthermore, the demonstrating robot test-bed with simulated faults validates its real-time support. Although the results of this paper are based on a specific middleware framework, OPRoS, the architecture can be applied to any other component-based robot systems.

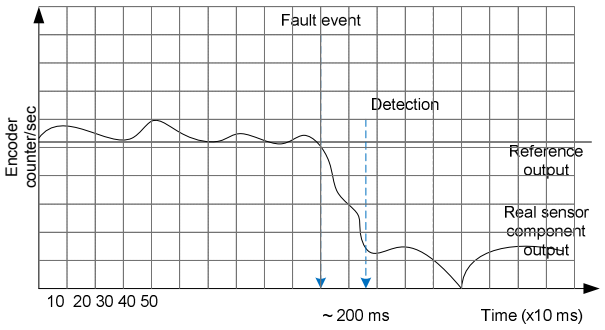


Fig. 8. Fault Propagation in the Proposed Framework

Table 3. Fault Recovery Time (numbers in the parenthesis are latencies after out component pre-loading technique is applied)

Load Level	winXP-Pentium4	Linux-Pentium4	Linux-PXA272
~10%	20 ms (<20 m)	20 ms (<1 ms)	20-50 ms (<1 m)
~40%	10-40 ms (<20 m)	2-12 ms (<1 ms)	20-120 ms (<20 m)
~60%	20-100 ms (<20 m)	10~30 ms (< 1 ms)	100-350 ms (<20 m)
~80%	200 ms (< 20m)	> 100 ms (< 20 ms)	> 500 ms (<20 m)

8. ACKNOWLEDGMENTS

This research was supported in part by the Ministry of Knowledge Economy (MKE), Korea, under the Strategic Technology Development Program (No-10030826)

9. REFERENCES

[1] Iborra, A., Caceres, D., Ortiz, F., Franco, J., Palma, P., and Alvarez, B. Design of service robots, *IEEE Robotics & Automation Magazine*, 16, 1 (Jan. 2009), 24 – 33.

[2] OMG, *Robotic Technology Component Specification Version 1.0*, April, 2008.

[3] Microsoft Robotics Developers Studio R2, http://msdn.microsoft.com/en-us/robotics.

[4] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng, "ROS: an open-source Robot Operating system," in *Proc. of the International Conference on Robotics and Automation* (ICRA), 2009.

[5] Korean Intelligent Robot Standard Forum Standards, *OPRoS Component Specification*, 2009.

[6] Song, B., Jung, S., Jang, C., and Kim, S. An Introduction to Robot Component Model for OPROS. In *Proceedings of SIMPAR* 2008, Italy, Nov. 2008, 592-603.

[7] OPRoS project official site, http://www.opros.or.kr/.

[8] ISO/TC 184/SC2/WG8 Service Robot Group DIS, *Robots and robotic devices — Safety requirements - Non-medical personal care robot*, 2010.

[9] Ahn, H., Lee, D.-S., and Ahn, S. C. A Hierarchical Fault Tolerant Architecture for Component-based Service Robots. In *Proceedings of 2010 IEEE Industrial Informatics INDIN '10)*, 2010, 487–492.

[10] Koren, I., and Krishna, C. M. *Fault Tolerant System*. Morgen Kaufman Publisher, San Francisco, CA, 2007.

[11] Isermann, R. Supervision, fault-detection and fault-diagnosis methods — An introduction. *Control Engineering Practice*, 5, 5, May 1997, 639-652.

[12] Bruce, J., Balch, T., and Veloso, M. Fast and inexpensive color image segmentation for interactive robots. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robot and Systems (IROS '00)*, 2000, 2061–2066.

[13] Lee, I., Iyer, R. K. Software Dependability in the Tandem GUARDIAN System. *IEEE Trans. on Software Engineering*, 21, 5, May, 1995, 455 – 467.

[14] Robbins, K. A. *UNIX System Programming*. Prentice Hall, 2004.

[15] Hart, J. M. *Win32 system Programming: Chapter 5 Structured Exception Handling*, Addison-Wesley Professional, 1997.

[16] Ahn, H., Oh, H.-J., and Hong, J..: Towards Reliable OSGi Framework and Applications. In *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, 1456-1461.