A Hierarchical Fault Tolerant Architecture for Component-based Service Robots

Heejune Ahn[1], Dong-Su Lee[1], Sang Chul Ahn[2]
[1]Dept. of Control and Instrumentation Engineering, Seoul National University of Technology, Seoul, Republic of Korea
[2]Imaging Media Research Center, Korea Institute of Science and Technology, Seoul, Republic of Korea
Email: heejune@snut.ac.kr,acemania83@hanmail.net, asc@imrc.kist.re.kr

Abstract- **Due to the benefits of reusability and productivity, component-based approach has become the primary technology in service robot system development. However, because component developer cannot foresee the integration and operating condition of the components, they cannot provide appropriate fault tolerance function, which is crucial for commercial success of service robots. The recently proposed robot software frames such as MSRDS (Microsoft Robotics Developer Studio), RTC (Robot Technology Component), and OPRoS (Open Platform for Robotic Services) are very limited in fault tolerance support. In this paper, we present a hierarchically-structured fault tolerant architecture for component-based robot systems. The framework integrates widely-used, representative fault tolerance measures for fault detection, isolation, and recovery. The system integrators can 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 components and the operating environment. To demonstrate the feasibility and benefits, a fault tolerant framework engine and test robot systems are implemented for OPRoS. The experiment results with various simulated fault scenarios validate the feasibility, effectiveness and real-time performance of the proposed approach.**

I. Introduction

Recently, interest in service robots has been increasing in research and commercial domains. A ‘service robot’ is a robot that provides services for the well-being of humans, society, and equipment outside industrial automation applications [1]. Carnegie Mellon University’s Minerva, Honda’s Asimo, Sony’s AIBO, and iRobot’s Roomba are just a few evidences of such high interest on service robots.

As it may be deduced from its definition, a service robot is not dedicated to a specific application but new services can be easily added to it. Because the component-based system development approach fits the requirement well, due to the modularity, reusability, and productivity, the recent robot software frameworks, notably Korea OPRoS [2, 3], OMG’s RTC (Robot Technology Component) [4], and Microsoft’s MSRDS (Microsoft Robotics Developer Studio) [5], are based on the component-based model.

For the commercial success of intelligent service robots, the fault tolerant technology for system reliability and human safety is crucial [6]. This is because mobile service robots operate with moving mechanical parts in the human working space. Since a fault is defined in relation to a system function and the functions are application specific, traditionally fault tolerance mechanisms have also been implemented at the application level. Furthermore in the component-based development, fault tolerance tools of component developers have been limited to the application level. However, our intensive survey on the fault tolerance for control and robot systems (Table 1) shows that the fault tolerant techniques therein share common design patterns, so that a framework may provide a systemic approach for fault tolerance function. In this paper, we demonstrate this argument by developing a fault tolerant OPRoS framework and applying it to example test scenarios.

In this paper, we present a fault tolerance framework for component-based robot system. The framework integrates widely-used, important fault tolerant measures for fault detection, isolation, and recovery. The system integrators can construct fault tolerance applications from non-fault-aware components, by adding simple helper components or/and declaring fault processing rules in configuration descriptors, considering the constraints of components and the operating environment. To demonstrate the feasibility and benefits, a fault tolerant framework engine and test robot systems are implemented for a component-based framework standard, OPRoS (Open Platform for Robotic Services). The experiment results with various simulated fault scenarios validate the effectiveness and real-time performance of the proposed approach.

TABLE I
WIDELY-USED FAULT DETECTION AND FAULT HANDLING TECHNIQUES IN CONTROL AND COMPUTING SYSTEM AREA (THE ‘>’ MEANS THE NEXT STEP WHEN THE FORMER METHOD CANNOT RECOVERY THE SYSTEM)

Fault Cause	Fault Detection	Fault Handling
SW/runtime error	OS exception	Reset>self-recovery>replacement> stop
SW/logical error	signal model	Reset>self-recovery>replacement> stop
SW/deadlock	watch dog	Reset >replacement> stop
HW/ additive error	signal model	replacement> stop
HW/multiplicative error	signal model	replacement> stop
Design error	Signal & process model	reconfigure> stop
Integration error	Signal & process model	reconfigure> stop
External (misuse, operating condition)	Signal & process model	reconfigure> stop

The remainder of the paper is organized as follows. Section 2 provides a brief description of OPRoS standards and presents the proposed fault tolerant framework with its architectural benefits. Section 3 describes the fault tolerant tools employed in the current OPRoS framework implementation. Section 4 shows our test-bed, simulated faults, and performance results. Section 5 concludes this paper with on-going works.

II. Framework Architecture

A. *OPRoS Framework*

A detailed description of OPRoS standards may be required for an implementation-level understanding of our proposed fault tolerant mechanisms, but the readers can understand the overall operation using Fig. 1 and the following quick summary. For a detailed description, refer to the specification [2], overview paper [6], and project site [3], where you can download the implementation source, sample components, and instruction manuals.

The framework, normally a process in the operating system, contains multiple components. Typical service robots include sensor, actuator and various algorithm components. The framework provides the execution, lifecycle management, configuration, and communication services. The framework manages the lifecycle of the components, such as loading/unloading the component library and creating/destroying an instance, the state of components using lifecycle interface functions, initialize(), start(), stop(), destroy(), recover(), update() and reset() and executes jobs by invoking the callback functions defined in the user components, that is, onExecute() and onEvent().A component can communicate only though ports, which are classified as 'data', 'event', and 'service' port types according to the synchronization and argument styles.

When its callback functions are called, a component can execute its jobs and communicate with other components only through ports. The component configuration information is provided in an XML file called 'component profile'. Also, the components are grouped into application tasks in another XML file called the 'application profile'. The connectors and adaptors for communication middleware have little to do with the subject of this paper, so we will give no further description here. Note that the briefly described OPRoS standards share the same architecture with OMG's RTC standards. Thus, most of the findings in this paper can be also applied to RTC based system.

B. *Proposed Fault Tolerant Framework Architecture*

The system architecture for fault management illustrated in Fig. 2 follows the hierarchical architecture. This is because detecting and confining errors to the lowest possible level of the system hierarchy maximizes the effectiveness of the recovery procedure and minimizes the impact of errors on system performance [7].

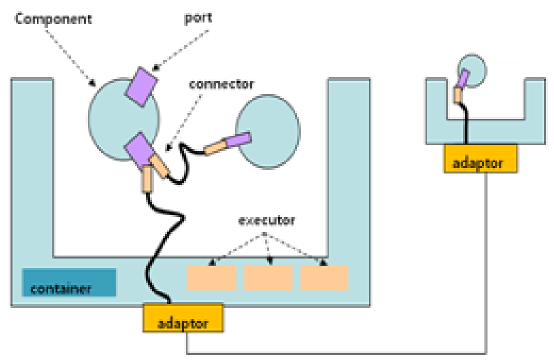


Fig. 1. The OPRoS Framework and Component Model [2].

The system integrators choose the appropriate fault handling tools by mainly declaring XML configuration descriptors considering the constraints of components and the robot's operating environment. Inter-task relation and handling is instructed by the fault manager. The ports, executors, and monitor components detect the low level faults by monitoring the input/output data and runtime exception handler, or comparing component outputs. The detected fault is notified the fault manager. The fault manager looks up the rule in the descriptor to diagnose the faults and decide the handling procedure. Some faults can be ignored or overcome by an internal handling. When the fault cannot be fixed, the influence is checked again using the descriptor file, and the defined action such as component replacement, application stop, or system all stop.

Fig. 3 compares the proposed fault tolerance architecture with traditional application level fault tolerance. A framework-based fault tolerance approach shifts the role of the fault tolerance mechanism from application components to the service platform. The fault manager in the framework prepares a set of fault tolerant measures of detection, isolation, and recovery. The system integrators choose the appropriate fault handling tools by mainly declaring XML configuration descriptors considering the constraints of components and the robot's operating environment.

Since the fault tolerance extension elements are described in detail in Section 3, we emphasize here that the framework-based architecture has many benefits over application level fault tolerance techniques. Firstly, all component developers do not need to be fault tolerant experts. In fact, they cannot be often ones. Repeated and different fault tolerance implementations can be avoided. The application component developer can focus only on its application function.
Secondly, the system integrators can check consistent reliability. Note that according to the reliability theory, the weakest component dominates the reliability of the overall system. The system integrators can check the level of reliability and enforce a certain level. Finally, the framework can perform system-level control beyond components. Components and applications in service robots are often dependent upon other components and applications. The framework can maximize the system reliability and usability using the dependence information.

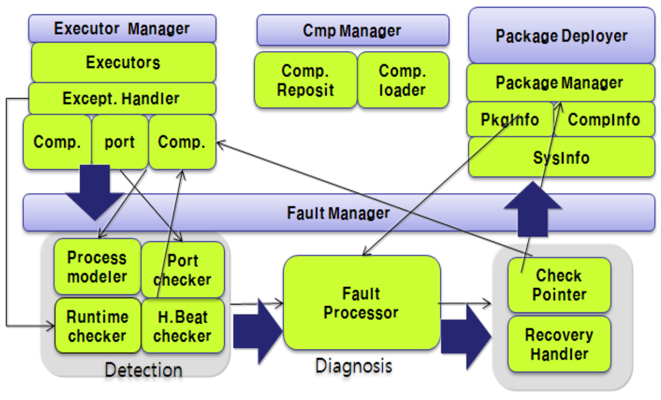


Fig. 2. Internal Components of OPRoS Framework Fault Manager and the Typical Fault Process Flow.

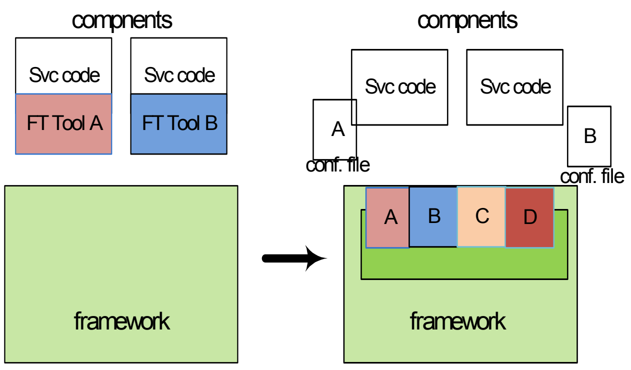


Fig. 3.Comparison between Application-based and Framework-based fault tolerance supports.

III. Employed Fault Tolerance Technologies

It should be noted that the main purpose of this paper is to find the appropriate fault tolerant tools for an OPRoS framework, not to invent a new fault tolerant tool. In general, the process for handling faults is performed in 3 steps: fault detection, diagnosis, and fault handling [8].

A. *Fault Detection*

Fault detection is the first and the most difficult step for fault processing. We integrated the following fault detection mechanisms into the OPRoS framework.

1. *Runtime Software Exception Detection*

Most runtime software exceptions such as segment fault, divided by zero are caused by coding bugs. The exception handling methods can be used for those runtime exceptions. Specifically, we implemented 'sigsetjmp&longjmp' on POSIX system [9], and SHE (structural exception handling), i.e., '__try &__except' on Microsoft Windows system [10]. 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 leak. Especially [11] reports that most memory faults start with un-initialized pointer and index overflows.

We are also investigating a component-based electrical fence based on Electric Fence [13]. The reason we cannot directly apply the electric fence algorithm is that allocation unit level protection demands too many resources, so inappropriate for real time system.

2. *Signal Model-Based Fault Detection*

Broken hardware and short/open circuits are practically the most common faults. Though logical errors cannot be detected easily without knowing the logics inside of components, it is possible to check the range of values and parameter type/number mismatch. In our approach, the component developers or integrators provide rules for checking the validity of the input and output values of ports. A valuable example is to provide a sensor input range from (min, max), where there is a non-zero value, and then we can check the dead component or open/short circuits.

3. *Process Model-Based Fault Detection*

A process model based technique [14] 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 requires the component and application designer's effort. Although this modeling difficulty dispirits our efforts for a framework-based fault tolerance support, we found two ways for the process modeling.

First, we have many components that are reasonably simple and practical to model. Most automatic control systems have a simple set-point with a desired target time. An example is a control signal to wheel motor driver (x, y, t) and the real rotation value (theta, t) from the gyroscope sensor. The sensor measurement is compared repeatedly with in (t–dt, t+dt), where 'dt' means the time tolerance. In our design, the model is declared again using the configuration XML.

For the case of complex system, the component developers or integrators can provide model function as a library. A task description script has been added to the OPRoS system recently [3], and we will use this script method soon.

B. *Fault Diagnosis*

When a fault is detected, a fault diagnosis is performed in 2 steps first in the executor and then the fault manager.

1. *Fault Cause Classification*

All the detected faults discussed above result in error return codes defined in the OPRoS specification. The executor elaborates OPRoS return types to classify the causes of faults such as caller, callee, type mismatch, resource shortage, and so on. Based on the fault severity level, i.e., 'ignore', 'reset', 'stop', in the configuration file, the different fault handling is performed. Also the coverage of faults is categorized into component, executor (thread or task), and system level. This is how the proposed mechanism provides fault-isolation and containment checking.

2. *Fault Effect analysis and Fault Isolation*

A fault in one component needs to stop the application that it is in. The stop, i.e., the fault of an application again requires the stop of other applications that depend upon the fault application. 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.

In the example case in Fig. 4, there are 3 kinds of components with component duplication, where 'white', 'red', and 'grey' denote healthy, faulty (in itself), and containment components or application, respectively. Therefore, application A can run in spite of two faulty components in type B, but application B cannot run any more. Again application D that depends upon the Service of application A, can run, even though application C dependent upon application B cannot run.

Our fault tolerance manager follows the dependency graph, and prevents fault propagation through the components and applications.

C. Fault Handling

Based on the fault diagnosis, fault handling is done in 3 different ways. Each step is illustrated in Fig. 5.

-Fault-Recovery (self-healing): whenever possible, the recovery of a fault component should be done fast enough for real-time operation. Component recovery mechanism has two type, component resurrection and component replacement, which are detailed below shortly.

-Fault-Operation: 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-Safety: 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.

1. *Component Resurrection*

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. Most stateless components and even some statefull components in robot applications can be resurrected by reset. 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.

2. *Component Replacement*

We chose 'recovery block' among the typical fault handling mechanisms. 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 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.

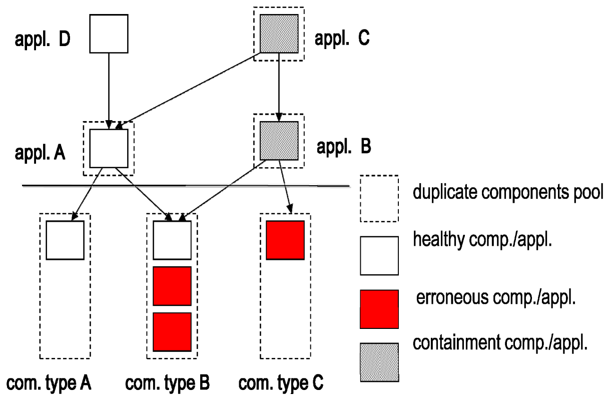


Fig. 4. Fault Propagation in Component-based Application Models

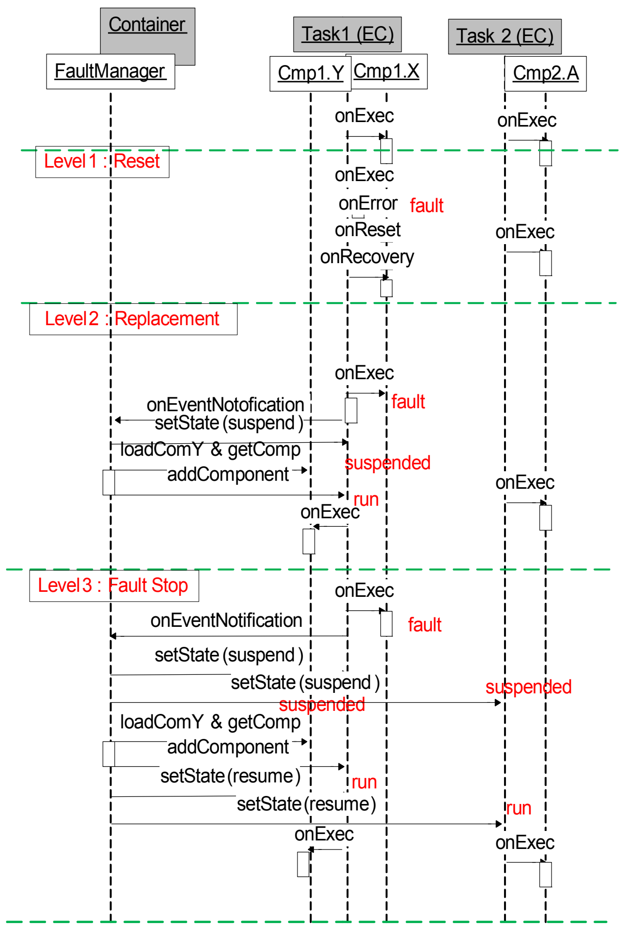


Fig. 5. Method Call Flows for 3 Different Fault Handling Cases

TABLE II.
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	Simple output comparator
	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)

IV. SYSTEM EVALUATION

A. *Functional Evaluation*

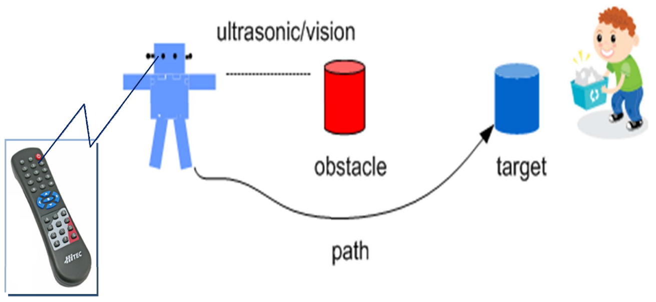
We implemented a fault-tolerant OPRoS runtime engine based on an OPRoS component specification draft. With our ORPoS engine, several experiments have been performed using a desktop Linux environment and an educational embedded Linux board with a Robonova body, HBE-Robonova-AI [15]. The main test application scenario is a navigation task essential for mobile service robots. The application consists of two independent sub-tasks of path planning and obstacle avoidance (Fig. 6).

The obstacle avoidance task uses one vision sensor, one color object detector, and two different obstacle avoidance algorithm components, one for primary and the other for secondary backup. The color object detection algorithm is based on the applied for robot soccer world-cup and published in [16]. The periodically captured image data at the vision sensor component is processed for ‘red’ color detection at the color object detection component, and then filtered at the obstacle avoidance component for illumination variation due to the light change and robot walk. The final obstacle information is sent to the path-planning component.

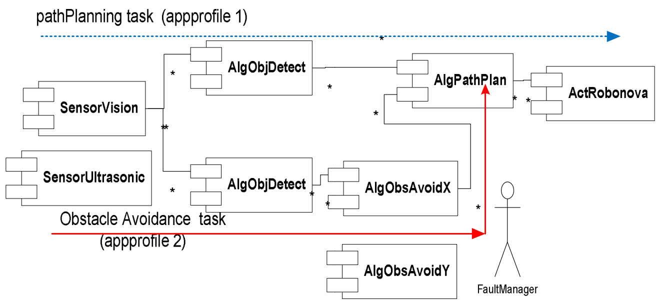
The path-planning task consists of one vision sensor component, one object detector component, one path planning algorithm component, and one actuator control component. The image data experiences the same flows as in the obstacle-avoidance task. The path-planning component receives the obstacle location from the obstacle-avoidance task, and obtains the target location information its own way, then builds a safe path for the target. The motion decision made at the path-planning component is sent to the actuator component for the DC motor control.

When no fault occurs, the robot reaches the target avoiding a collision with the red obstacles. Though the generation of a real fault at the sensor or actuator is desirable, it was not easy for our test robot system. Instead, we injected faults such as segment fault errors by setting a wrong pointer value using the IR remote controller input.

When a fault in the first obstacle-avoidance component occurs, the fault is handled at the fault manager, either resetting or replacing the faulty components. It is considered safe to stop the path-planning task when the obstacle avoidance task cannot perform correctly. So when the first component gets faulty and no secondary components are prepared in the ‘ObstacleAvoidAppProfile.xml’ configuration file, the robot stops walking and generates a ‘help’ beep sound.



(a)Scenario Illustration: Robot Navigation.



(b) Component and Application Task Structure for Robot Navigation with Obstacle Avoidance.

(c) Snapshot of Test
Fig. 6. Test Application Scenario

B. *Real-time Performance Evaluation*

In addition to the functional verification, we measured the detection and recovery time to check the real-time performance. Table III shows the latency variation with various system load conditions. The runtime and logical exception handling takes a few milliseconds in a moderate system load condition. However, a secondary component replacement procedure often takes over several hundred milliseconds when the load of tasks is 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 component to 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.

We invented two methods to overcome this problem. The first solution maintains the computation and memory load of 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.

V. CONCLUSION

This paper had developed a hierarchically structured, framework-based fault tolerant architecture for component-based robot systems in contrasts with the typical application-level approach. This is because the component-based development in robot middleware or framework can solve the software and system reusability problems in robot application software development, but the system service such as real-time scheduling, security, and reliability functions must be taken care of by framework at the component integration stage. This paper focused on the fault tolerance and reliability function.

We presented a concrete implementation based on the coming robot component standards, OPRoS, and showed how well and easily the typical and popular fault tolerance tools can be accommodated. Furthermore, the framework-based approach has many benefits of a system-wide reliability guarantee and ease in customizing off-the-shelf non-fault-aware components. The demonstrating robot test-bed with simulated faults validates the real-time support as well as the effectiveness of the proposed hierarchical framework architecture.

However, we have realized that the component level information and processing would be more effective if not inevitable, to provide the fault tolerance service to more realistic service applications, especially for fault detection. We are now developing a component extension mechanism that enhances fault detection function to the original service components.

TABLE III.
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%	8 ~ 20 ms (<1m)	1 ~ 3 ms (<1ms)	10 ~ 50 ms (<1m)
~40%	10 ~ 40ms (<1m)	2~ 12ms (<1ms)	10 ~ 120 ms (<1m)
~60%	20 ~ 100ms (<1m)	10~30 ms (<1ms)	100 ~ 350 ms (<3m)
~80%	> 200ms (<1m)	> 100 ms (<1ms)	> 500 ms (<5m)

ACKNOWLEDGMENT

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

REFERENCES

[1] A. Iborra, D. Caceres, F. Ortiz, J. Franco, P. Palma, and B. Alvarez, "Design of service robots," *IEEE Robotics & Automation Magazine*, vol. 16, no. 1, pp. 24 – 33, 2009.

[2] Korean Intelligent Robot Standard Forum, "OPRoS Component Specification," Standards, 2009.

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

[4] OMG, Robotic Technology Component Specification Version 1.0, April, 2008

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

[6] B. Song, S. Jung, C. Jang, and S. Kim "An Introduction to Robot Component Model for OPROS," in *Proc.of SIMPAR 2008*, Italy, Nov. 2008.

[7] C. Ferrell, "Failure Recognition and Fault Tolerance of an Autonomous Robot," *Adaptive Behavior*, vol. 2, pp 375-398, 1994.

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

[9] K. A. Robbins, UNIX System Programming, Reading, Prentice Hall, 2004.

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

[11] I. Lee, R. K. Iyer, "Software Dependability in the Tandem GUARDIAN System," *IEEE Trans.on Software Engineering,* vol. 21, no. 5, pp. 455 – 467, May, 1995.

[12] G. R. Lueke, J. Coyle, J. Joekstra, M. Kraeva, Y. Li, O. Taborslaia, Y. Wang, "A Survey of Systems for Detecting Serial Run-Time Errors," Concurrency and Computation: Practice and Experience, vol. 18, no.15, pp. 1885-1907, 2006.

[13] B. Perens, "Electrical Fence", http://perens.com/Freesoftware/Electricfence.

[14] R. Isermann, "Supervision, fault-detection and fault-diagnosis methods — An introduction," *Control Engineering Practice*, vol.5, no. 5, pp. 639-652, May 1997.

[15] Hanback Electronics Inc., HBE-Robonova-AI: an Embedded Robot with Robonova Body, http://www.hanback.co.kr/products/.

[16] J. Bruce, T. Balch, and M. Veloso, "Fast and inexpensive color image segmentation for interactive robots," in *Proc. of the 2000 IEEE/RSJ International Conference on Intelligent Robot and Systems (IROS '00)*, vol. 3, pp. 2061–2066. 2000.