Design and Implementation of a Fault Tolerant Computing Platform for Service Robots

Heejune Ahn¹ and Sang Chul Ahn²
1Seoul National Univ. of Technology
Seoul, Republic of Korea
[e-mail: heejune@snut.ac.kr]
2KIST (Korea Institute of Science and Technology)
Seoul, Republic of Korea
[e-mail: asc@kist.re.kr]

Abstract
For commercial success of intelligent service robots, the fault tolerant technology for system reliability and human safety is crucial. Fault tolerance methods have been implemented almost in application level. However, based on the common design patterns in fault tolerance studies, we argue that a platform-based approach provides many benefits in providing reliability for system development. The fault manager in platform provides a set of fault tolerant measures of detection, isolation, and recovery. Therefore, the application component developer only focuses on components’ service function, and the system integrators choose the appropriate fault handling tools by declaring XML configuration descriptors, considering the constraints of components and operating environment. To demonstrate the benefits, we build a platform based fault tolerant (PBFT) engine for OPRoS (Open Platform for Robotic Service) standards. We also provide the specific techniques useful for service robot system.

Keywords: Fault tolerance, platform based approach, design pattern, robot software platform, OPRoS, component-based development

1. Introduction
A ‘service robot’ is a robot that provides usefullness for well-being of humans, society, and equipment outside industrial automation application. Recently, interests on service robots are increasing in the research and comercial domain. Due to the benefits of modularity, reusability, and productivity, many robot software platforms, especially based on component model, are proposed; OPRoS [1], RTC (Robot Technology Component), and MSRDS (Microsoft Robotics Developer Studio).

For commercial success of intelligent service robots, the fault tolerant technology for system reliability and human safety is crucial [2], because mobile service robots operate with moving mechanical parts in the human working space. Since most faults are application specific, fault tolerance technology have been also considered and implemented in 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 pattern, so that a platform can provide systemic approach for fault tolerance function.
In this paper, we demonstrate this argument using fault tolerant OPRoS platform and example scenarios. Section 2 provides a brief description of OPRoS platform. Section 3 describes the benefits of the PBFT approach. Section 4 describes the fault tolerant tools included. Section 5 shows an experimental use-case with performance results. Section 5 concludes this paper.

### Table 1. The fault causes and proposed fault-tolerant mechanisms.

<table>
<thead>
<tr>
<th>Origin Type</th>
<th>Fault detection</th>
<th>Fault handing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>OS signal exception</td>
<td>Setjmp&amp;longjmp _try&amp;_exception</td>
</tr>
<tr>
<td>Logical</td>
<td>validity and range check (&lt;\text{tag}) in comp profile</td>
<td>State initialize</td>
</tr>
<tr>
<td>state error</td>
<td>Heat beat check</td>
<td>State initialize</td>
</tr>
<tr>
<td>common</td>
<td>(&lt;\text{retry}_\text{max}&gt;&lt;\text{secondary}&gt;&lt;\text{tag}&gt; &lt;\text{value}&gt; &lt;\text{app}&gt;</td>
<td>2\text{ndary component Replacement Fault isolation}</td>
</tr>
<tr>
<td>platform external</td>
<td>Time-out Exception</td>
<td>resurrection, watchdog</td>
</tr>
<tr>
<td>HW (sensor/actuator)</td>
<td>Driver Health Check</td>
<td>Port value check</td>
</tr>
<tr>
<td>User</td>
<td>E-Stop Button</td>
<td>Global Fault manager</td>
</tr>
<tr>
<td>common</td>
<td>(&lt;\text{severity}&gt; &lt;\text{tag}&gt; &lt;\text{value}&gt; &lt;\text{app}&gt;</td>
<td>Global Fault manager</td>
</tr>
</tbody>
</table>

2. The OPRoS Platform Standards

A detailed description of OPRoS platform is required for 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.

The container, a process in operating system, contains multiple application tasks called ‘application profile’. Each application task is mapped to a thread in implementation called ‘executor’. An executor runs one or more components. A component has one or more communication ports, which are one of ‘data’, ‘event’, and ‘service’ port types.

The container manages the lifecycle of components, such as loading/unloading the component library and creating/destroying an instance. The reason why the container should manage the instance is because the component can be single tone (a shared instance) as well as multiple (an instance for an executor).

The executor controls the state of component 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(), onEvent(). When its callback functions are called, a component can execute its jobs and communicate with other components only through ports. The connectors and adaptors for communication middle ware have little with subject of this paper, so no further description here.

3. PBFT Architecture

A platform-based fault tolerance (PBFT) approach shifts the role of fault tolerance mechanism from application components to the service platform (Fig. 2). The fault manager in the platform prepares a set of fault tolerant measures of detection, isolation, and recovery. The application component developer only focuses on its application function, and the system integrators choose the appropriate fault handling tools by declaring XML configuration descriptors considering the constraints of components and robot’s operating environment.

The PBFT architecture has many benefits over application level fault tolerance techniques.
- All component developers do not need to be fault tolerant experts. In fact, they cannot be ones.
- Repeated and different fault tolerance implementation can be avoided.
- The system integrators can provide consistent reliability.
- The system integrators can check the level of reliability and enforce a certain level.

**Fig. 2.** Comparison between application-based and platform-based fault tolerance supports.

Furthermore, the system architecture for fault management follows the hierarchical architecture. It is because we believe that 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 the error on system performance [4]. The ports and executor detects the low level faults by monitoring the input and output data and runtime exception handler. Some insignificant faults can be ignored or overcome by an internal handling. Inter-task relation and handling is instructed by the fault manager.

### 4. Fault Tolerance Tools in PBFT

#### 4.1 Fault Detection

It should be noted that the main purpose of this paper is to find the appropriate fault tolerant tools for OPRoS platform, not to invent a new fault tolerant tool. In general, fault handling process is performed in 3 steps: fault detection, diagnosis, and fault handling [3]. We start discussion with the fault detection mechanism. Fault detection is the first and the most difficult step for fault processing. We developed the following fault detection mechanisms.

A. Run-time 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 ‘__try & __except’ on Microsoft Windows system [5] and ‘sigsetjmp & longjmp’ on POSIX system [6]. 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. A component-based electrical fence is developed based on Electric Fence [7, 8], not to have too much resource demands for allocation unit level protection demands [8].

B. Signal Model Based Fault Detection

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

C. Process Model Based Fault Detection

Process model based technique uses models of the system components and plant (such as environments), and then compare the simulation results with the measured values. When the difference of them is larger than a certain threshold, it is considered as a fault. In generally the model can be very complex and stochastic, but we have many components that have reasonably simple and practical to model. An example is for control signal to wheel motor driver and the real rotation value from the gyroscope sensor. 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.

Ahn et al.: Classification of Traffic Flows into QoS Classes by Unsupervised Learning and KNN Clustering

4.2 Fault Diagnosis and Isolation

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

A. Fault Cause Classification

All detected faults discussed above result in an error return codes defined in 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.

B. Fault Handling Decision

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 the fault-isolation and containment checking.

4.3 Fault Handling

A. Fault Handling Strategy

Based on the fault diagnosis, fault handling is done in 3 levels. The each step is illustrated in Fig. 3 and the configuration extension in Table 2.

- **Fault-Recovery** (self-healing): whenever possible, the recovery of fault component should be done fast enough for real time operation.
- **Fault-Operation**: when a fault cannot be recovered, the relevant components in the system should also be checked, so that fault containment region is minimized.
- **Fault-Safety**: when a fault cannot be recovered and keeping working the system may harm to human beings or environment, the system should perform emergency stop not to cause critical effects to human beings.

B. Component Resurrection

Three callback functions, onError(), onReset(), onRecover() are called when an error occurs and the platform tries to reset the faulty component then the component is recovered, respectively.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Elements (DTD schema used for space limit)</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault isolation handling</td>
<td>dependency := (name, reference, severity) name := (#PCDATA) reference := (URL) severity := (ignore</td>
<td>stop</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fault := (retry_max, secondary+, severity) retry_max := (#PCDATA) reference := (URL) severity := (ignore</td>
<td>local</td>
</tr>
<tr>
<td>Fault detection</td>
<td>validation := (rang_min, range_max) rang_min := (#PCDATA) range_max := (#PCDATA)</td>
<td>component profile</td>
</tr>
</tbody>
</table>

C. Component Replacement

We chose ‘recovery block’ among typical fault handling mechanisms. When the executor detects the fault cannot be overcome by reset and
it is serious component for the task level, it is reported to the fault manager. The manager checks the application configuration file whether a secondary component are 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.

5. Evaluation Experience

5.1 Test Scenario

We implemented a fault-tolerant OPRoS runtime engine [9, 10] based on OPRoS component specification draft.

With our ORPoS engine, several experiments have been performed using desktop Linux environment and an educational embedded Linux board with a Robonova body, HBE-Robonova-AI [11]. A test application scenario including two independent tasks of path planning and obstacle avoidance in Fig. 3 is prepared.

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 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 to the target avoiding the 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 inject faults such as segment fault errors by setting 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 ‘node.xml’ configuration file, the robot stops walking and generates a ‘help’ beep sound.

5.2 Experiment Results

In addition to the functional verification, we measured the detection and recovery time for check the real-time performance. Table 3 shows the latency variation with various system load condition. The runtime and logical exception handling takes a few milliseconds in moderate system load condition. However, secondary component replacement procedure often takes over several hundred milliseconds when load of tasks is over 80% CPU computing power or memory usage. It is not unusual operating condition in an embedded robot system because of its resource limits and heavy loaded vision processing.
5.3 Latency Analysis and Solutions

The analysis revealed the cause and performance patterns is originated from 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 ‘dlopen()’ option RTLD_NOW/LAZY is found. The difference between desktop Linux and embedded target is due to the slow flash rom.

We invented two methods for this problem. First solution limits the computation and memory load under a certain level, i.e., 80%. Second solution is 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 msec in any case we tested. The sizable recovery latency observed under the heavy load condition has been resolved using our pre-loading technique. As a result, the recovery time also satisfies the real-time requirement.

Table 3. OPRoS XML elements for Fault Tolerance

<table>
<thead>
<tr>
<th>Load level</th>
<th>WinXP/P4</th>
<th>Linux/P4</th>
<th>Linux/arm (flash ROM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~10%</td>
<td>8 ~ 20 ms (&lt;1ms)</td>
<td>1 ~ 3 ms (&lt;1ms)</td>
<td>10 ~ 50 ms (&lt;1ms)</td>
</tr>
<tr>
<td>~40%</td>
<td>10 ~ 40 ms (&lt;1ms)</td>
<td>2 ~ 12 ms (&lt;1ms)</td>
<td>10 ~ 120 ms (&lt;1ms)</td>
</tr>
<tr>
<td>~60%</td>
<td>20 ~ 100 ms (&lt;1ms)</td>
<td>10 ~ 30 ms (&lt;1ms)</td>
<td>100 ~ 350 ms (&lt;3m)</td>
</tr>
<tr>
<td>~80%</td>
<td>&gt; 200 ms (&lt;1m)</td>
<td>&gt; 100 ms (&lt;1ms)</td>
<td>&gt; 500 ms (&lt;5m)</td>
</tr>
</tbody>
</table>

6. Conclusion

This paper proposed a platform based fault tolerant (PBFT) architecture for robot service development in contrast to typical application level approach. We argued that the PBFT approach has many benefits including system-wide reliability guarantee and easiness to customize off-the-shelf non-fault-aware components. The verification test with our extended OPROS platform showed that the proposed PBFT provides real-time fault handling and minimizes the fault containment, and finally fault safe feature.

References