A Fault-Tolerant Architecture for Component-based Service Robot Software Platforms

Heejune Ahn¹, Byoung Wook Choi¹, Sang Chul Ahn²
¹Seoul National University of Technology, ²Korea Institute of Science and Technology
{heejune, bwchoi}@snut.ac.kr, asc@imrc.kist.re.kr

Abstract – This paper presents a fault tolerant architecture for OPRoS platform, a standard component-based robot software platform. The proposed architecture follows OPRoS system model hierarchy; the fault tolerant measures of detection, isolation, and recovery are mapped onto ports, components, tasks, and containers of OPRoS system. This hierarchical approach minimizes the fault handling time and fault containment area. Also, the fault handling is customized using XML descriptors by the component developers and system integrators who well understand the constraints of components and robot's operating environment. Test results using robot navigation service verify the flexibility and the real-time performance of the proposed fault tolerant architecture.

Keywords – fault tolerant, fault safe, robot software platform, OPRoS, component-based development.

1. Introduction

Due to the benefits of modularity, reusability, and productivity, recently many service robot software platforms are proposed based on component-based architecture: OPRoS [1], RTC (Robot Technology Component), and MSRDS (Microsoft Robotics Developer Studio).

The requirement level of safety and fault-tolerance is even higher than industrial robots [2], because mobile service robots operate with moving mechanical parts in the human working space. Furthermore, system integration with unverified components cannot guarantee the conformance between components and system level reliability. In spite of these importance and new challenges, the fault tolerant and safety mechanism in the component-based service robot platforms are not yet comprehensively studied.

This paper presents a fault tolerant architecture to detect platform-based faults and provides fault tolerant mechanisms to guarantee system and human safety. The paper is organized as follows. Section 2 provides a brief description of OPRoS platform and the requirement of fault-handling and safety. Section 3 describes our proposed approach for building fault tolerant platform. Section 4 shows an experimental use-case with performance results. Section 5 concludes this paper with a discussion about future study items.

2. Fault Tolerance Requirement for OPRoS

2.1 The OPRoS Platform Specification

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 container¹, 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 middleware have little with subject of this paper, so no further description here.

Fig. 1. The OPRoS platform and component model [1].

2.2 Fault Tolerance Requirement

The OPRoS platform has two key features. It is a platform for mobile robot and based on a

¹ Precisely writing, a platform includes containers and executors. However, the terms platform, container, and runtime are used interchangeably in this paper.
component-based model. The following requirement analysis is performed based on this recognition.

- Real time handling: a robot is real-time system working in the human working area. Therefore, the fault detection and handling should be performed reasonably quickly.
- Fault isolation: component-based approach allows the possible usage of incompletely verified SW/HW components.
- System integration: it’s also component-based system’s feature. Usually, system integration is not easy due to the interface mismatch and diverse/unexpected operating environment.
- Human Safety: mobile robots operate in circumstances where human beings perform activities. At least ‘fault safe’ should be taken when in-time recovery cannot be taken.

3. Proposed Fault Tolerant Architecture

3.1 Architecture Overview

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. We performed an intensive survey on the fault tolerance for control and robot systems in ‘google scholar’ and major on-line journal sites. Table 1 summarizes our survey result and analysis result.

![Fig. 2. The proposed fault tolerant architecture for OPRoS platform.](image)

The proposed fault tolerance approach follows hierarchical architecture, because it is the structure of OPRoS system itself, and 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.

3.2 Fault Detection

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.

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

<table>
<thead>
<tr>
<th>Origin</th>
<th>Fault type</th>
<th>Fault detection</th>
<th>Fault handing</th>
</tr>
</thead>
</table>
| Runtime      | OS signal exception | Setjmp&longjmp | _try&_exception_
| Logical      | validity and range check | <range> tag in comp profile | State initialize                  |
| Component    | state error | Heat beat check | State initialize                  |
| common       | <retry_max><secondary> in app profile | 2ndary component Replacement Fault isolation |
| Others       | platform external | Time-out Exception | resurrection, watchdog            |
| HW (sensor/ actuator) | Driver Health Check | Port value check |                                  |
| User         | E-Stop Button | Global Fault manager |                                  |
| common       | <severity> tag in app profile | Global Fault manager |                                  |

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 error from invalid valued pointers, buffer bound overflow, and memory leak. Many debugging tools for memory access check are presented such as Purify, Valgrind, and Electric Fence [7]. Each tools use different mechanism, source code change, customized ‘malloc’ with memory protection, object code changes and so on.

A component-based electrical fence is developed. The electrical fence needs neither component source code nor CPU information, but allocation unit level protection demands too much resource [8]. Building electrical fence for a component, not each allocation reduces the page allocation burden sharply.

B. Logical Fault Detection

We define logical fault as a fault that does not cause runtime exception, but the results of operation are useless
or contrary to the designer’s intention. A few examples are out-of-range of values and parameter type or number mismatch (it is not runtime fault in the OPRoS system).

Logical errors cannot be detected without knowing the logics inside of components. However, to examine the inside of component is not possible and contrary to the component based development paradigm. Therefore, our approach is to make the component developers or integrators provide rules for the validity check of the input and output values of ports. Then, the platform reads the configuration files, sets the validation rules to ports. Finally the port checks the validity rule by examining the input and output data passing through it.

C. Other Faults Detection

Broken or Degenerated hardware and communication loss are the main source of external faults. For hardware faults such as sensors and actuators, we assumed that a discrete detection is prepared².

3.3 Fault Diagnosis

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.

3.4 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.

C. Component Replacement

We chose ‘recovery block’ among typical fault handling mechanisms. We consider ‘N-version programming’ as a fusion technique in application level. 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.

<p>| Table 2. OPRoS XML elements for Fault Tolerance |</p>
<table>
<thead>
<tr>
<th>Purpose</th>
<th>Elements</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault isolation</td>
<td>dependency := (name, reference, severity)</td>
<td>node profile</td>
</tr>
<tr>
<td>handling</td>
<td>name := (#PCDATA)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reference := (URL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>severity := (ignore</td>
<td>stop</td>
</tr>
<tr>
<td>Fault recover y</td>
<td>Fault := (retry_max, secondary+, severity)</td>
<td>application profile</td>
</tr>
<tr>
<td></td>
<td>retry_max := (#PCDATA)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reference := (URL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>severity := (ignore</td>
<td>local</td>
</tr>
<tr>
<td>Fault detection</td>
<td>validation := (rang_min, range_max)</td>
<td>component node profile</td>
</tr>
<tr>
<td></td>
<td>rang_min := (#PCDATA)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>range_max := ((#PCDATA)</td>
<td></td>
</tr>
</tbody>
</table>

4. Test, Evaluation, and Optimization

4.1 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 in 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 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 values 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.

### 4.2 First Experiment Results

With 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 conditions. The runtime and logical exception handling takes a few milliseconds in moderate system load conditions. However, secondary component replacement procedures often take over several hundred milliseconds when load of tasks is over 80% CPU.

![Fig. 3. Our fault detections and handling](image)
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.

4.3 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.

Table 3. the Fault Handling Response Time

<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 ~ 40ms (&lt;1ms)</td>
<td>2 ~ 12ms (&lt;1ms)</td>
<td>10 ~ 120 ms (&lt;1ms)</td>
</tr>
<tr>
<td>~60%</td>
<td>20 ~ 100ms (&lt;1ms)</td>
<td>10 ~ 30ms (&lt;1ms)</td>
<td>100 ~ 350 ms (&lt;3ms)</td>
</tr>
<tr>
<td>~80%</td>
<td>&gt; 200ms (&lt;1ms)</td>
<td>&gt; 100 ms (&lt;1ms)</td>
<td>&gt; 500 ms (&lt;5m)</td>
</tr>
</tbody>
</table>

5. Conclusion

This paper presented a fault tolerant architecture for OPRoS platform. Its hierarchical approach enables the real-time fault handling and minimizes the fault containment, and finally fault safe feature. 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. The configurable approach presented is found to be very realistic method at today’s technology level. However, our research focused on mainly software component with the strict assumption that the hardware fault can be detected. Our next study will extend our architecture to cope with typical robot hardware faults.

Acknowledgement

The research was supported by the Korean Ministry of Knowledge Economy (MKE), the Strategic Technology Development Program, No.10030826

References