Robust by "Let it Crash"

Christoph Woskowski, Mikolaj Trzeciecki, Florian Schwedes

Robust by „Let it Crash“

Christoph Woskowski, Mikolaj Trzeciecki, Florian Schwedes
Zühlke Engineering GmbH
Landshuter Allee 12
80637 Munich, Germany
{christoph.woskowski,mikolaj.trzeciecki,
florian.schwedes}@zuehlke.com

Keywords— safety-related; fault-tolerance; supervisor hierarchies; let-it-crash; Erlang

A. Introduction

Critical software systems are bound to perform extensive error detection and exception handling. The corresponding source code is typically implemented in a defensive programming style. Typical strategies to ensure robustness include elaborate exception handling and error-code returning routines. Most often, error handling code fragments are often not separable from the source code realizing the core functionality, and they are prone to errors themselves. For extending exception handling in order to further improve fault-tolerance, even more source code is necessary. However some leftover vulnerability always remains, especially in complex, multithtreading, and distributed systems. Producing more code ultimately results in more complexity while reducing readability and maintainability. This in turn inevitably leads to programming errors.

The programming language Erlang breaks a new ground for handling fault-tolerance problems. Very light-weight processes in separate memory areas enable straightforward concurrency with communication solely based on message passing. Processes are able to monitor and – in case of a process termination – restart each other very swiftly. The exception handling method of choice for a worker process is to terminate itself (“let it crash” – LiC), if it is unable to handle the situation locally. Dedicated supervisor hierarchies ensure appropriate error responses by starting a different process or by restarting a new instance of the terminated one.

This work presented in this abstract investigates, whether the let-it-crash paradigm for fault-tolerant systems may also be applicable to safety-related software projects. The scenario chosen for this demonstration approximates (and simplifies) a project within the medical device control software domain.

B. ModelProject

Although often a necessity, long term hospitalization is expensive and can even pose a health threat to hospitalized people. For reducing these costs and risks, a number of patients are treated at home. In such a case, an appropriate and reliable monitoring system must be used. In our (fictional) project, such a monitoring system is developed which uses so called “functional clothing”. This clothing is a kind of garment incorporating wireless sensors, which allows the patient to move freely around without being restricted, even while their vital signs keep being monitored. The signals from the sensors arrive wirelessly at a base station located in the same house or room as the patient. This device employs a constant connection with all active sensors, is able to power them on and off and switches to an alternative measurement location if need arises (failure, implausible data). The base station establishes a connection with the hospital and transfers the data for evaluation.

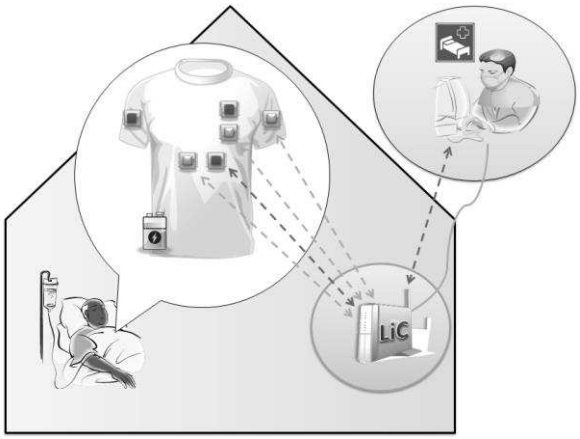

Fig. 1. Proof of concept scenario

The subject matter of the LiC proof-of-concept is the software development for the base station. The project focusses a high reliability of measurement data acquisition and transfer of the patient’s vital signs to the hospital. A maximum number of currently active sensors is set to limit power usage. At the same time a minimum coverage of the vital signs has to be guaranteed: for every point in time at least two out of three critical values (heart rate, breathing rate and blood pressure) have to be available.

The safe state of the house station is a complete shutdown, since the hospital system gets alarmed about the missing data.

C. Implementation and testing

Our prototypical implementation in Erlang makes use of the supervisor hierarchies and allows for deployment of worker processes and supervisors as well as the evaluation of separating business logic from error handling. The

development concentrates on the software of the base station and just simulates the external sensors on the one side and the hospital system on the other. The diagram [Fig. 2] depicts the example setup, showing the runtime view of the processes and dependencies.

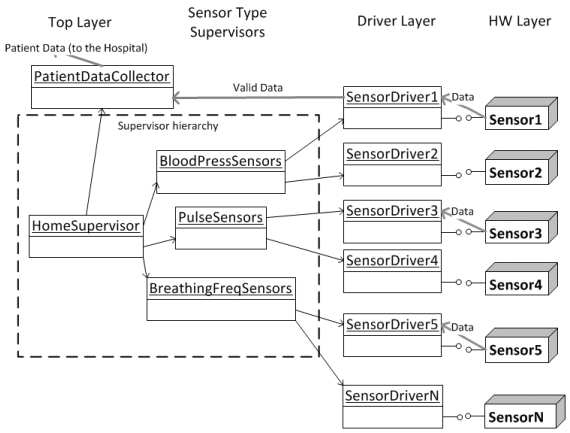

Fig. 2. Runtime view of processes and dependencies

The generic supervisor hierarchy is solely responsible for creating the worker processes (sensor drivers and data collector) and for handling errors by restarting or replacing terminated processes.

The sensor drivers and the data collector on the other hand contain the core functionality (business logic) and no error handling at all. In case of missing sensor values, for example, the sensor driver just terminates and gets replaced. The same happens if there is data available but outside of valid limits.

In connection with regulatory requirements concerning medical devices (e.g., IEC 60812), we test the prototype depicted above for the following failure situations:

- Failure to perform the desired function
- Performing a function that was not desired
- Performing a function at a wrong time
- Incorrect timing or order of executions
- Recognition and handling of critical conditions by the system

A simple and effective variant of testing fault-tolerance is based upon a so called "Chaos Monkey" - a process injected into the system under test with the sole task of randomly terminating other system processes. In traditional systems with a small number of complex tasks, this typically leads to complete failure within a very short period of time.

In our system following the LiC philosophy this only triggers the process monitoring and thus a fast replacement of the terminated software part. This has been tested in a simulated uninterrupted Base Station run of multiple days. In spite of the chaos monkey killing random components, our system is able to maintain basic functionality.

Further, we tested the concurrency behavior of the system by adding the necessity of the sensors to calibrate themselves. The calibration functionality opposes the normal sensor activity, as the abovementioned limitations to the maximum and minimum count of the active sensors remain in place. In our prototype, a sensor performing calibration at undesired moment gets "crashed" by a dedicated supervisor, following the LiC approach consequently.

D. Conclusion

Considering the LiC application hypotheses proposed above, the following can be stated about the patient monitoring scenario implemented in Erlang:

1. ***Ensure the execution of critical functionalities***. Ill-performing tasks are stopped and restarted, no matter the cause. For instance, a malfunctioning sensor driver gets terminated and replaced by another one.

2. ***Prevent the unintended execution of a function***. When a functional monitor detects a worker executing an unintended function, this worker gets terminated and replaced, thereby preventing the execution. For instance, a sensor calibration is aborted when there is another calibration request of higher priority.

3. ***Define and monitor the conditions for carrying out a critical function***. Workers and functional monitors can control task execution and results given distinct validation checks. This excludes any measures to correct the situation besides restarting affected processes. A sensor driver validates the data received from its sensor before forwarding it to the collector. If a violation is detected, the driver terminates itself so the supervisor can start another driver which in turn can connect to another physical sensor. The driver does however not attempt to correct the invalid values in any way.

4. ***Ensure carrying out critical functions at a specific time and in specific order***. Conflicts within task sequences can be resolved by terminating blocking processes which violate the order or a time constraint, as illustrated by the sensor calibration functionality. Thus lifelocks in calibration concurrency can be prevented – allowing only one sensor to calibrate at a time – and calibration of any sensor type is guaranteed within a given time-interval.

5. ***Unexpected failures have no influence or result in a safe state.*** Malfunctioning processes are immediately replaced by new ones, thus ensuring their functionality is not lost. Fatal function loss immediately results in system shutdown. For instance, the patient controlling system is robust with regard to sporadic process crashes as well as to the complete loss of one sensor data type.

E. Future work

The missing hard real-time abilities of Erlang pose a problem when it comes to time-critical safety applications. There are strategies to solve this issue, e.g. using external low-level libraries written in C/C++. These solutions have to be analyzed and developed further. For the applicability of LiC for safety critical systems, the underlying Erlang language features have to be evaluated against safety standards like IEC 61508-3. Research is also necessary on whether it is possible to apply LiC without Erlang. Analyzing the language features and corresponding counterparts in other languages or frameworks will provide the necessary information.