Online Robustness Testing of Distributed Embedded Systems: an Industrial Approach

Khaled Alnawasreh*†, Patrizio Pelliccione*, Zhenxiao Hao*, Mårten Rånge†, and Antonia Bertolino‡
*Chalmers University of Technology | University of Gothenburg
Department of Computer Science and Engineering, Gothenburg, Sweden
†Ericsson AB, Gothenburg, Sweden
‡ISTI - CNR, Via G. Moruzzi 1, 56124 Pisa, Italy
khaled.nawasreh@gmail.com, patrizio.pelliccione@gu.se, haozhenxiao2010@gmail.com,
marten.range@ericsson.com, antonia.bertolino@isti.cnr.it

***Abstract*—Having robust systems that behave properly even in presence of faults is becoming increasingly important. This is the case of the system we investigate in this paper, which is an embedded distributed system consisting of components that communicate with each other via messages exchange in the RBS (Radio Based Station) at Ericsson AB in Gothenburg, Sweden. Specifically, this paper describes a novel fault injection approach for testing the robustness of distributed embedded systems with very limited computation power. The new approach is inspired by Netflix's ChaosMonkey, a fault injection approach that has been developed for testing distributed systems hosted in the cloud. However, ChaosMonkey cannot be used in the context of RBS since the latter consists of small-embedded components with specific requirements of performance, programming language, and communication paradigm. This paper reports about the approach called Postmonkey we developed, illustrates the results of applying it to RBS, and discusses the potential of utilizing fault injection to test complex, embedded, and distributed systems. The approach and tool are now adopted by Ericsson.**

***Keywords*-online testing; fault injection; distributed embedded systems;**

I. Introduction

As software for distributed systems becomes more complex, ensuring that a system meets its prescribed specification is a growing challenge for software developers [1]. Testing of large scale distributed systems may be very hard due to the fact that many different types of faults can occur at any time and it is very hard to imagine all possible scenarios that could occur due to interactions and collaborations among the distributed and potentially independent systems. Traditional testing techniques are not sufficient to predict the effects of faults on real-world applications running on large distributed systems [2]. In general, manual tests tend to test the existence of specified (known) failure modes. However, unknown failure modes should be also taken into account.

This is true in particular in testing system robustness, i.e. the degree to which a system can continue to function correctly even in the presence of invalid inputs or stressful environmental conditions [3]. As a matter of fact, verifying the robustness of distributed system is not trivial. Moreover, when faults occur in running services, faults can exponentially propagate [4]. For this reason, testing should become also a run-time activity [5]. Online testing aims at evaluating systems in their real execution environment [6] and at validating that the system meets its requirements continuously.

In this paper, we present an approach, called Postmonkey[1], that has been conceived for online robustness testing of distributed embedded systems. Postmonkey has been developed in collaboration with a team at Ericsson AB in Gothenburg, Sweden. As said, when dealing with large scale distributed systems it is very hard to anticipate the possible scenarios that can occur as caused by interactions and collaborations among the distributed and potentially independent systems. This motivates the interest for non-deterministic approaches that permit to go beyond specified and, consequently, known failure modes.

The Postmonkey approach provides a non-deterministic testing method for checking the robustness of distributed embedded systems with very limited computation power. Postmonkey injects two different fault types, namely sending invalid messages and delaying the messages. We believe that randomized testing is a cost-effective way to complement traditional testing, remove potential developer bias and identify unknown failure modes.

To develop the approach, we followed an agile development process, and consequently Postmonkey was evaluated continuously also through log files storing wanted and unwanted system behaviors, like system interrupts, message delaying, timeouts and so on. Continuous evaluation of the approach helped us in developing the work as needed, decreasing the uncertainty level, and increasing the work quality. The source code of each iteration was pushed to a remote repository which practitioners had access to. During regular weekly meetings with practitioners, we got feedback on the fault injection approach and tool. Furthermore, during the last iteration, we presented the fault injection approach and tool to the development team for a final validation.

As a conclusion of this study, we show that fault injection techniques can be used for improving the robustness of embedded distributed systems. Furthermore, using our approach we detected unexpected faults within distributed embedded

[1]Postmonkey is a portmanteau fusing the word Postman and Monkey. The rational is that the approach is inspired by ChaosMonkey and applies it at the level of message exchange (content and delay) - this is why Postman.

systems of Ericsson. Although the quality of the tested system has been proven to be quite high, there have been some surprises, which were not caught by traditional testing: (i) fuzzing of signals has detected a weakness in the system handling dynamically sized objects that might cause a crash and that could be then rectified, and (ii) randomized delays detected misconfigured timeout handling. It is important to highlight that the manual test cases were defined based on the same wrong assumptions made for the production of the code, thus the error was missed.

Postmonkey also provided a clear indication of dependency bottlenecks, i.e. where faults might most probably occur. Consequently, we could suggest improvements of the software architecture and some of these improvements were actually implemented, based on the observed results. The approach is now adopted by the division of Ericsson with which we collaborated for realizing this work. At present stage, Postmonkey has not yet been deployed in customer production networks, as further validation is ongoing. However, it is now used in networks that are run internally at Ericsson to capture faults at last stage before deployment to production.

Summarizing, the main contribution of the paper is Postmonkey, a non-deterministic testing approach for checking the robustness of distributed embedded systems with very limited computation power. The approach and lessons learned are general and can be adopted and replicated in other companies. The successful collaboration model between academia and industry is based on iterative development/validation cycles and close collaboration with practitioners thus enabling continuous feedback and integration with real environments.

This paper is structured as follows: Section II identifies the context of the paper and presents the industrial environment in which we performed the research. Section III compares the approach to related works. Section IV presents the research methodology. Section V introduces Postmonkey and Section VI describes the implementation of the approach. Section VII reports the validation and results obtained by applying the approach to a real industrial environment. Section VIII discusses lessons learned. Finally, Section IX concludes the paper with final remarks and future research directions.

II. Context

The usage of the Internet is continuously growing in the industry due to the fact that more and more devices are connected, as shown in Figure 1. Specifically, Machine-to-Machine (M2M) connection is expected to grow strongly, driven by new use cases, e.g., in cars, machines and utility metering. Distributed systems need to be increasingly resilient to different types of faults and combinations of them as part of the system's normal operating procedure. Combination of faults may have a major impact on the performance of the system, sometimes even leading to systems being temporarily out of service, which has dramatic effects on embedded distributed systems, because such systems are increasingly fragile and safety critical [7]. Indeed, distributed systems tend to have more unexpected faults than other types of systems when used in reality [2].

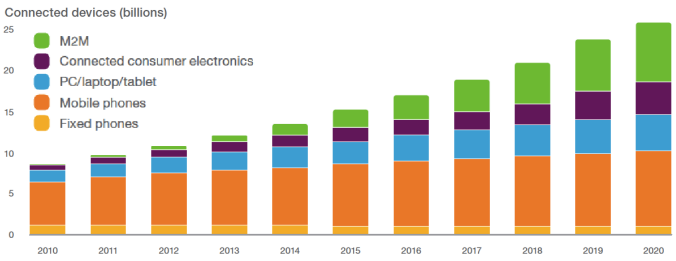

Fig. 1. Ericsson Mobility Report June 2015 [8]

RBS (Radio Base Station) is an integral part of the mobile data solutions delivered by Ericsson. RBS consists of embedded components that are connected in order to support high network availability to a wide area network. A mobile network is a distributed system that consists of many RBSs. Along with the increase of network signaling volumes and the operators demand of more complex configurations for supporting more usage scenarios, the need for improving resilience is also increasing. The network should be flexible to respond to the challenge of the dramatic increase of the network traffic and signaling created by the operators and subscribers [9].

The Performance Management (PM) framework is one key component of RBS used to count phone calls and network usages of subscribers. The RBS components, including the PM framework, are tested using traditional approaches like unit testing and functional testing. Unit testing provides the instruments for testing the functionality of the units taken individually, however, it will not catch integration or system-level errors. Functional testing combines different test cases and can test the code at system level, however, such a combination is usually limited. Functional and unit testing approaches are mostly used to address failure modes that are expected. Potential unknown failure modes are very hard to be detected by traditional testing.

In order to catch also unknown failure modes, testing should be randomized and fault injection [10] can be a valuable technique to be exploited for this purpose. While several approaches exist for fault injection, such as Chaos Monkey [11], they are not adequate for testing distributed embedded systems with very limited computational power such as the PM framework. In fact, the use of Chaos Monkey is costly and consumes extra resources such as network bandwidth, storage space, and processing power [21]. This motivated the design and implementation of Postmonkey, which is specifically tailored for systems with limited computational power.

III. Related Work

A. Fault injection

There are two types of fault injection: hardware and software fault injection [12]. In hardware fault injection faults are injected at the physical level by controlling the parameters of the environment variables. In this case, fault types can be like

disturbing the power supply, voltage sags, heavy ion radiation and electromagnetic interference etc. [12]. In software fault injection a possible strategy to inject faults is to consider different fault types such as register and memory faults, system overload, and missing, delayed and faulty messages. Faults can be injected within the application, between the target and operating system or between the critical components in the system. Software fault injection techniques can be classified into two groups based on when and where the faults are injected: compile-time and run-time faults. Faults can be triggered by different mechanisms, for instance, time-out, exception, code modification, and sending of different random messages [12].

The work in [2] presents a new model called Failure Scenario as a Service (FSaaS). The main purpose of this model is to test the resilience of cloud applications. This can be done by monitoring the result of generating different failure scenarios on the cloud. This work investigates the impact of the failure on the jobs running in Hadoop for analyzing the MapReduce jobs. Hadoop is an open source software framework to process large distributed data set on huge computer clusters [13]. MapReduce is an efficient model for processing and generating large data set that is running on the distributed system on computer clusters [14]. A fault injection tool called AnarchyApe [15] was used to inject faults into Hadoop clusters. In order to evaluate the effects of individual faults and combinations of faults, some sample fault scenarios and different types of workloads were performed on different types of Hadoop jobs. The authors discovered that the resulting behavior of the distributed system depends much on the fault types, combination of faults, job types, and the time when the faults are injected. The study presents a new model that can deal with large cloud applications to be tested efficiently. The model helped them to identify the weak spots of the system and trying to fix them as well as monitor the resources for efficient utilization. The limitation of the model is that it can be only used by cloud service providers and clients who rely on Hadoop. However, their goal is to develop a system that can be applicable to general cloud scenarios [2]. In this paper, we aim at developing a fault injection approach and a tool that can be applied to small embedded distributed systems.

B. Online Testing

Because of the inherent complexity of modern systems, testing cannot be confined during development and offline phases but needs to be extended also to online phases. Several online testing approaches have been proposed in many areas, like service choreographies [16], [6] [17], [18], [19], [20].

1) Chaos Monkey Testing: Chaos Monkey Testing (CMT) [11] has been developed when Netflix moved their data center to Amazon Web Services (AWS). The main reason for developing CMT was to assess how potential failures in AWS would affect their ability to produce continuous services. CMT works by sending fault commands to the components that are hosted in the cloud. On receiving the commands, the instance of the cloud itself will fail. By introducing faults CMT enables Netflix to discover the bottlenecks and weaknesses in their systems. Moreover, this helps them to strengthen the weak areas that are critical in their system. CMT helps to detect different failure scenarios and to find unexpected failures that cannot be detected using traditional methods.

CMT software design is flexible and can be used for other cloud service providers. However, it is just applicable for testing the availability and robustness of the services running in the cloud. Chaos Monkey Testing is applicable in systems that have the ability to perform even in the presence of faults. Otherwise, the risk is that the overall system will crash within a short time. Additionally, CMT suffers from the limitation in terms of computational power. In the case of a large number of test cases to be executed, it might even dominate the power consumption. Using CMT in large-scale systems will be costly and will consume extra resources such as network bandwidth, storage space, and processing power [21]. When using this approach, an engineer should consider hardware reliability, memory managements, and the runtime environment.

2) Let-it-Crash Philosophy: The work in [22] investigates the use of the Let-it-Crash (LiC) paradigm to assess the applicability of safety-related software, check how error handling performs, as well as identify potential improvements for future work. The main challenge of the LiC paradigm is to identify different software fault types in order to improve error handling in safety critical systems.

LiC only triggers the monitoring process, then fast replacement of the terminated software part can be performed. LiC can be efficiently utilized in safety-related software development that has a low number of complex tasks. The programming language should be Erlang and it does not cover large-scale embedded distributed system. Furthermore, this approach has a problem when it comes to safety-critical systems with hard time constraints, since Erlang does not support them [22].

IV. Research methodology

Our collaboration model between academia and industry is based on the design and creation methodology, which was developed to address theoretical questions about the characteristics of learning in context. It is mainly used for creating new knowledge and artifacts that are required to solve a particular phenomenon or problem [23].

The Postmonkey approach and tool have been conceived and implemented following an agile development process and in strong synergy with practitioners within Ericsson. As anticipated in the introduction, the approach and tool have been developed in iterations (precisely four) and each iteration was composed of four phases, namely:

- *Awareness of problem*: we had weekly meetings with practitioners at Ericsson and analyzed available documentation about the PM framework for identifying the bottleneck of the system and determining the injections. Additionally, Ericsson's internal fault reports and other

available documents helped in gathering more information about the weakness of the system and identifying where problems can occur.

- *Suggestion and solution*: based on the Ericsson fault report and on existing fault tolerance solutions of the PM framework, potential fault types were discussed with practitioners. We decided to address two fault types: the first one was sending random messages and the second one was delaying messages. The rationale of sending random messages is to test the filterability of the PM framework against random invalid messages. Message delays were introduced between the critical components of the PM framework where faults can occur.
- *Implementation*: the implementation phase contains two steps: tool development and tool integration. It is detailed in Section VI.
- *Evaluation*: since we followed an agile development process we had a continuous evaluation of the approach and tool through the following means: (i) weekly meeting with practitioners for getting feedback; (ii) continuous validation of defined requirements for the expected conformance level on how the approach should work; (iii) observation of the results of the execution of the fault injection tool stored in log files; (iv) comparison of the observed results with the expected ones (how the system should perform in its normal state); (v) if faults were detected, evaluation of the faults by tracing backward to their causes.

V. The Postmonkey approach

Figure 2 presents the basic components of Postmonkey. It consists of the target system (PM_fwk - which stands for PM framework), fault injector, monitor (log files), Inter Process Communication (IPC) library as well as the controller that includes the automatic testing environment and the configuration files. As better explained in Section VI configuration files are exploited to make the fault injector customizable, extensible, and reusable in different contexts.

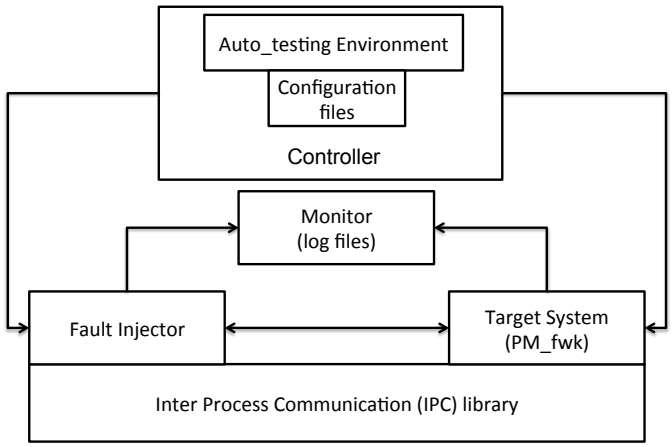

Fig. 2. The Postmonkey approach

After running the Postmonkey tool, all data are monitored from the log files using stack trace. These data are collected and analyzed for the evaluation of the approach and of the tool. The user can run and control the tool through the controller, where the automatic testing environments and the configuration files are located. Through the automatic testing environment, different testing techniques are listed. The Postmonkey tool supports random mechanisms where the number of faulty messages, message delay, and the time interval between sending the messages are randomized. The user can manually adjust the test cases through the configuration files.

The Postmonkey approach provides different fault types at different locations and times. As said, we have developed two fault types: (i) sending random messages and (ii) delaying the messages. Fault types such as sending invalid messages and delaying the messages were done by linking some dynamic libraries of IPC during run time. IPC library is a separate component, which consists of several system functionalities such as sending messages between different system component.

In order to perform interesting and valuable experiments, together with practitioners, we identified suitable system targets, i.e. some critical components of the PM framework that have strong dependencies. Faults were injected on system targets that have strong coupling. Due to the fact that the time factor can have an effect on running the experiments and to make sure that the results we got are reliable, we have repeated the experiments an additional two times.

1) Sending random messages: The communication within the system follows some specific protocols. Sending random messages is mainly developed to test the fault tolerance ability of the protocols. The messages are categorized into three types: request, confirmation, and rejection. Based on the designed fault tolerance feature and the possible message types in reality, we only perform sending of random request messages. We focus on three aspects of the invalid messages, namely the origination, the contents and the amount of sending.

The system is designed to tolerate invalid request messages, one main filtering standard of invalid messages is the origination of the messages. We focused on three aspects of the message origination: the namespace name, the mailbox name, and combination of namespace and mailbox names. We first created multiple mailboxes that have the same namespace name, i.e., the random messages are generated from different threads but from the same process. Then we created mailboxes with same mailbox names but different namespace names; this means that the random messages are generated from both different threads and different processes. Lastly, we created mailboxes with both different mailbox names and namespace names. The three cases of mailbox and namespace names are shown in Table I.

	mailbox	namespace
case 1	S	D
case 2	D	S
case 3	D	D

S stands for same
D stands for different

TABLE I
COMBINATIONS OF MAILBOX AND NAMESPACE NAMES

The message contents inside the system varied dramatically: some contents describe the actual payload, some describe the protocol version, some specify the size of the messages. The fields of the messages were also different from each other. Based on the above characteristics, we constructed the random messages from two different aspects: random fields, and random contents.

The number of random messages would affect the filtering ability of the protocols. For instance, by sending one random message, the system is able to filter it, but by sending 1000 random messages, the system might only be able to filter 500 of them. With a huge amount of random messages, the CPU load of the systems would be high. Please refer to Section VII for details about the performed experiments.

2) Delaying messages: The second fault injection scenario was delaying the messages between the system components. The delay mechanism used a fixed time interval, which we identified as default value. Moreover, it used a randomized-time interval between different components of the system in order to check if the system can handle different timeout intervals. Multiple types of messages are exchanged within the PM framework, e.g., request message, response message, rejection, confirmation message and messages describing the data. Some types of messages followed some timeout mechanism strictly. For instance, when the connection request is sent, the node waits for confirmation or rejection messages for a period of time; if neither a confirmation nor a rejection message is received, the current request of the node just goes in timeout.

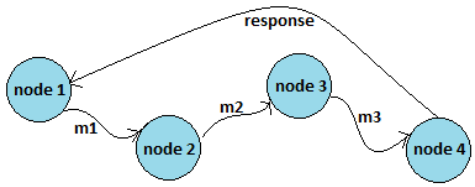

Fig. 3. Message delaying chain

Some messages sent within the PM framework have relationships with each other. Those messages usually work as a chain, as shown in Figure 3. In this case, `node1` wants to send a request to `node4` for some data, however, `node4` is reached by sending a message to `node2` that in turns will send a message to `node3`, which finally will deliver the request to `node4`. Once `node1` sends a message to `node2` it starts also a clock for the time-out mechanism, while `node1` only waits for a response from `node4`. Delays can happen on any of the messages sent between `node1` and `node4`. In reality, such a chain can be really long and the delay of the messages sent between each node can also be very unpredictable.

Delaying messages in the PM framework also tests the network performance. Currently, the network performance test of the PM framework is conducted by adjusting the network bandwidth. This approach has two main drawbacks. First, the test has some hardware requirements, which makes the test more complicated. Second, adjusting the network bandwidth makes the test uncontrolled, because the delaying of the message caused by a low network bandwidth is very hard to track. The current network performance test is like a black-box testing since what is really going on in the system during the test is not completely clear. Our test provides a white box testing for the PM framework network performance. During the message delaying, each message delay is recorded in the log file, and thus the PM framework developers at Ericsson can easily track the message delaying. A novel network bandwidth benchmark has been created based on the experiences of the message delaying.

VI. Implementation

The implementation of Postmonkey consists of two steps: tool development and tool integration. The first step mainly concerns the development of the fault injection tool using an Ericsson internal shared library called Inter Process Communication (see Section VI-A); the results of this step are some compiled C++ object files. The second step mainly concerns the work of integrating the developed tool into the automatic testing environment of the PM framework, where Shell Scripts and Ruby were used (see Section VI-B). Then, this section will describe the implementation of the procedure for the two fault types we considered in the study: sending random messages (see Section VI-C) and delaying messages (see Section VI-D).

A. Inter Process Communication

Inter-Process Communication (IPC) is used as a communication paradigm in some components of the PM framework. The collection of information from the system is done through the collection of the counters and events between its components. IPC is used to send messages between mailboxes, processes, and processors. A processor is an execution unit that is handled by one instance of an operating system, it can have multiple cores and several simultaneously executing contexts. A process has its own memory map and can have several running threads. A thread is a single execution context that can coexist with other threads within a process. We worked on a simulated environment, where the processor was the CPU of the computer, a process was a simulation of a processor on the embedded components, and a thread was a simulation of a process in the real operating system. Figure 4 shows the relationship between the real operating system and the simulated environment. In our implementation, only processes and threads were involved; they were identified by the process namespace and the thread name.

IPC is a widely used library of RBS. The PM framework is built on top of IPC and uses the APIs of IPC for the communication within the system. By exploiting the IPC APIs we performed our implementation without touching the code of the PM framework and were able to inject random faults into the PM framework from outside the framework. In this way, the fault injection is not requiring any extra computation to the framework.

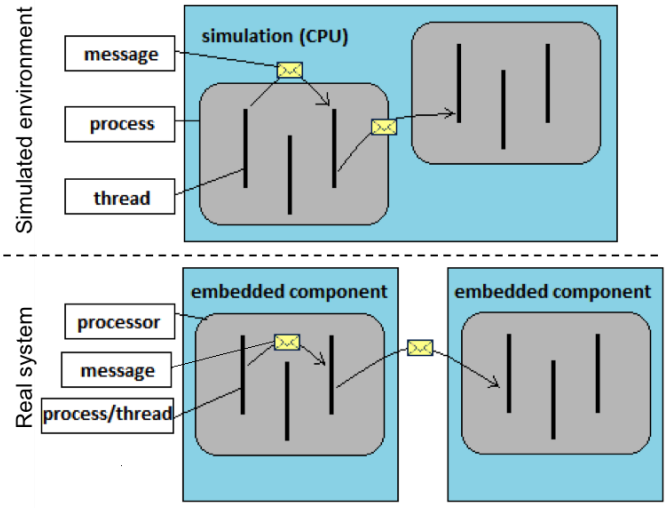

Fig. 4. Relationship between real system and simulated environment

B. Automatic testing environment

We developed a set of automated scenarios to test the functions of the PM framework. The automatic testing environment is used to test the PM framework with such scenarios. The automated test scenarios will be used by the fault injection tool to verify that the PM works even in the presence of faults.

The Postmonkey tool was integrated into the automatic testing environment using Shell Scripts, which inject random faults into the PM framework while executing the scenarios. The automatic testing environment also controls the operations of the fault injection tool like code generation, compiling and linking. The integration of Postmonkey with the automatic testing environment facilitates the adoption in industry and enhances the usability of the tool.

The results of the execution of the fault injection are presented at the end of the testing execution and details are stored in log files. Figure 5 shows an example of testing execution: it shows that some errors have been found and, consequently, some log files have been created.

```
##########################################################################
   > ASim: Timed out while waiting for ROP files!
##########################################################################
   > Debug information:
      > LTTng Trace:
         > Errors:     Some
         > Abnormals: Some
      > Core dumped:  No
##########################################################################
   > Logs:
      > simulator call log:
   /repo/ehhozzo/pm_fwk/iwas1/pmfBl/testAll/asim/asim_cmds.log
      > validation results log:
   /repo/ehhozzo/pm_fwk/iwas1/pmfBl/testAll/asim/asim_results.log
      > random message log:
   /repo/ehhozzo/pm_fwk/iwas1/pmfBl/testAll/asim/asim_random_msg.log
      > LTTng trace log:
   /repo/ehhozzo/pm_fwk/iwas1/pmfBl/testAll/asim/lttngTrace.log
##########################################################################
```

Fig. 5. Running results of automatic testing environment

C. Sending random messages

In IPC, different messages were constructed with C++ structs; after initialization, the message objects were encapsulated in a message union and sent together. The fields of the messages are specified in some XML files; an excerpt is shown in Listing 1. We first ignored those files and constructed the messages with random fields, this means that the messages might miss some necessary fields and might contain some extra unnecessary fields. After that, we constructed the messages following the XML files but with randomly filled contents, for instance, making the field called messages size to be a random number instead of the real message struct.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <signal name="CountersRcfm" osename="COUNTERS_RCFM
      " size="12">
3         <element name="sigNo" size="4" stereotype=
      "signal_number" />
4 </signal>
```

Listing 1. XML that specifies message fields

Constructing the message structs was based on the skeleton of the XML files, different categories of messages had different message fields and required different C++ header files. In order to implement the randomization mechanism, all the request message structs and their corresponding required header files need to be considered. To make the functionality of sending random messages customizable, the messages are specified in a configuration file. A code generator written in Ruby is used to create the message structs. By reading the configuration file and the XML file that specifies the message fields, different message structs code are generated and compiled to different binaries.

The next step after the construction of random messages is to create the senders. The senders are actually some mailboxes located in the same namespace, which means that each mailbox has a separate thread and they are all located in the same process. In order to send the messages to some targets, the mailbox address of the targets should be known. In the PM framework, the mailbox addresses can be located by using combinations of the namespace names and the mailbox names. In our implementation, we put all the namespace names and mailbox names into vectors, we calculate all possible combinations and we put them into a map. So if there are n namespace names and m mailbox names, then we have $m * n$ combinations in total. Once a TID[2] is successfully located, it is saved in another vector that will be used as the target of the sending. The messages are sent in two different ways, synchronously and asynchronously. Sending the messages synchronously is implemented by joining the mailbox to the main thread.

Sending random messages is integrated into the automatic testing environment, which is written in Shell script. Every error triggered by the random messages is logged and the logging is implemented with a watch dog. The watch dog keeps track of which message are sent and how many of them

[2]TID stands for Thread ID, i.e. an identifier used to identify a thread. It is also used to identify the mailbox associated with that thread.

were sent. The watch dog also keeps pinging all the previously located TIDs and if any of them crashes at the time when the messages are sent, the watch dog would also log such a TID. The log file provides a summary of all the relevant events happened while injecting random messages to the PM framework; this facilitates the detection of errors that are not found with unit testing.

D. Delaying messages

In the PM framework, messages are sent using IPC by linking some dynamic libraries during runtime. In C and C++, it is possible to wrap the dynamic library functions. Such a wrapping is implemented by creating shared libraries that override the functions of the original dynamic libraries. Such a dynamic library file can be linked by setting an environment variable called `LD_PRELOAD`. We implemented the delaying messages by wrapping the IPC dynamic libraries that are responsible for sending messages. As shown in Listing 2, the name of original message sending function was called `__itc_send`, the main parameters of this function were the message contents named "*msg*", the receiver named "*to*" and the sender named "*from*". The wrap function had exactly the same function name and parameters with the original function.

```
1  int __itc_send(
2    union itc_msg **msg,
3    itc_mbox_id_t to,
4    itc_mbox_id_t from,
5    const char *file,
6    int line)
7  {
8    int (*__real_itc_send)(
9      union itc_msg **,
10     itc_mbox_id_t,
11     itc_mbox_id_t,
12     const char *,
13     int) = dlsym(RTLD_NEXT,"__itc_send");
14   sleep(1);
15   __real_itc_send(msg, to, from, file, line);
16 }
```

Listing 2. Code that wraps the sending function of IPC

As it can be seen in Line 14, before calling `__real_itc_send`, we let the current thread sleep for 1 second; this means that the `__itc_send` function was always executed 1 second after being called, which implemented the delaying of message sending.

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <config >
3   <env var="DEBUG" value="true"/>
4   <env var="LD_PRELOAD" value="/PATH/TO/wrap.so"/>
5 </config>
```

Listing 3. XML for setting the environment variable

After being compiled, the shared library can be loaded by setting the environment variable `LD_PRELOAD` to the path of this shared library. The environment variables are set in an XML file as shown in Listings 3 and 4. At Line 4 of Listing 3 the environment variable called `LD_PRELOAD` is set to the path of a shared library file called "wrap.so". Listing 4 is an XML configuration file of a node in the PM framework, Line 5 contains the path to the environment variable configuration file, the environment variable is set by adding this line into the node configuration file and restarting the node.

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2 <appdata target="appm">
3   <loadmodule tag="dynamic" name="
      PMTESTAGGREGATOR" id="CXC1737528">
4     <file type="i686" relpath="SOME_PATH" />
5     <file type="config" relpath="PATH_TO/
      env_config.xml" />
6   </loadmodule>
7 </appdata>
```

Listing 4. XML configuration file of a node in the PM framework

VII. Validation and Results

This section introduces the results of performing fault injection with the Postmonkey tool. First, we show the steps of performing the fault injection, the variable parts, and the expected outcomes. Then, we show the real outcome, the performance of the PM framework and the detected bugs.

As explained in the introduction, we developed the approach through an iterative approach. In order to evaluate if the fault injection approach reached the conformance level against its requirements, the approach was evaluated at the end of each iteration. At the end of the final iteration, we made a final validation. The fault injection approach has detected unexpected faults that were not found by previously used testing techniques. As matter of fact, we could also see that this approach is able to identify the bottleneck of existing embedded distributed systems. In the following, we report the detailed results of the validation. Specific information is however omitted according to a non-disclosure agreement with the company.

A. Sending random messages

The validation related to injecting random messages aims to give an answer to this question:

- *To what extent is the system able to filter invalid messages?*

To answer the above question, we sent random messages in two ways, synchronously and asynchronously. Filtering of invalid messages is conducted by checking the properties of the messages. Such checking consumes time and processing power. So the filtering ability can also be affected by the frequency of the random messages. To test the filtering ability of the PM framework in terms of random message frequencies, the random messages are sent to the PM framework with different frequencies. This was implemented by giving different time intervals between each message sending.

The number of invalid messages that need to be generated in order to let the system behave differently from what expected varies depending on the time interval between the messages as well as the time when the messages are triggered. Sending a lot of invalid messages without having any time interval between messages leads the system to behave differently than expected. This happens due to the fact that the computation power of the embedded components is limited, the number of the messages are large and there is not enough time between message

sending. Thus, the system will not be able to recognize if a message is valid or not.

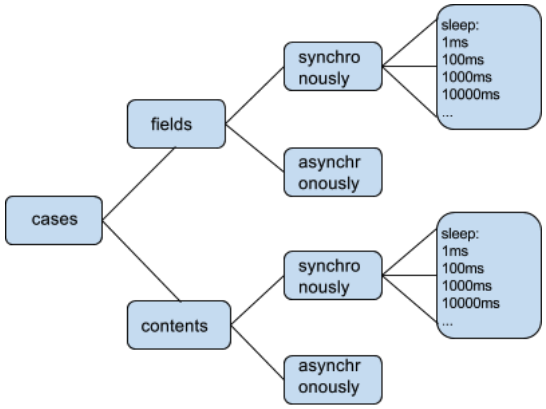

Fig. 6. Testing chains

Figure 6 shows the test chains of the PM framework. In order to perform the test randomly, we created different C++ classes to represent different variables in the chain. There are four variable classes, which are Cases, DataType (fields or contents), Concurrency (synchronously or asynchronously), and Intervals. When the test starts, variable objects of the test chains are initialized and connected randomly. One sample test chain from the left to right can be read like this: test case1, with same fields but different contents, send the random messages synchronously, the time interval between each random message is 100ms.

By performing the random message sending to the PM framework, we are able to test the PM framework according to two different aspects. The first aspect is the message type, which tests the PM framework at the functional level. The second aspect is the message amount, which tests the PM framework at the performance level. The module test of the PM framework has covered almost all the invalid message sending, but the test is not random and the messages are sent one at a time. Our test sends random messages randomly and messages are also combined randomly. This helps the PM framework developers at Ericsson to find corner cases that were not covered before.

Sending random messages also plays a performance test role. By adjusting the number of random messages and the time interval between the random messages sent, a performance threshold of the PM framework was found. Based on the demand in reality, this test helped the PM framework developers to find secure hardware requirements, which provide optimal and economic solutions for Ericsson.

Figure 7 shows a curve of sending random messages to one node in the PM framework. The response time of messages sent to this node should be less than 200 milliseconds. If no response is received after 200 milliseconds, the sender will time out. Figure 8 shows a snapshot of sending random messages synchronously with different time intervals between each sending, which will not trigger timeout. Figure 7 shows that the node can maximally receive 1200 random messages

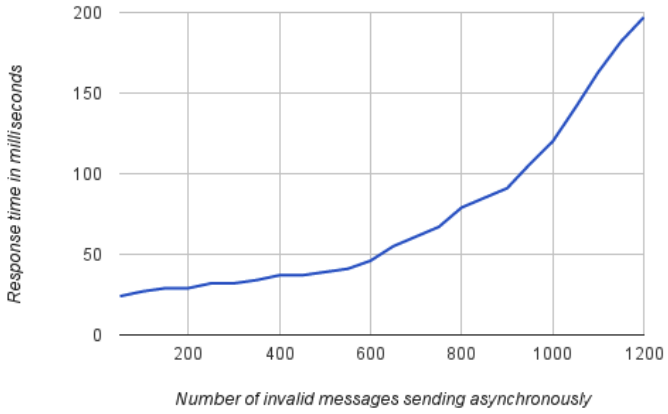

Fig. 7. Sending invalid messages asynchronously

at the same time. Figure 8 shows that in order to send 6000 messages, each message sending should have at least an interval of 80 milliseconds. These snapshots have been saved as benchmarks for improving the performance of the nodes.

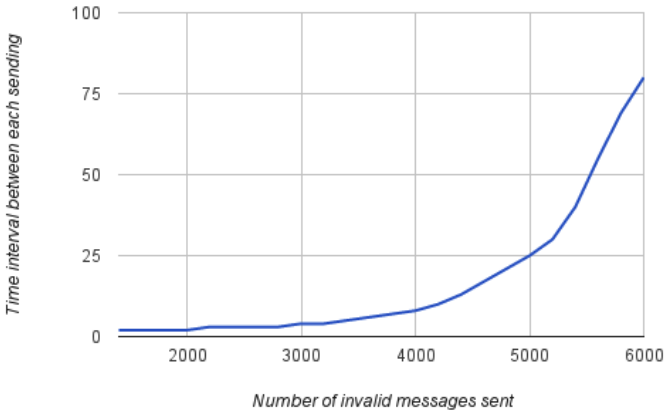

Fig. 8. Sending invalid messages synchronously

Thanks to these tests, we identified some errors that were not identified by traditional testing. For instance, fuzzing of signals, i.e. perturbing the messages/inputs randomly, has detected a weakness in handling dynamically sized objects, which was causing a crash.

B. Delaying messages

In this case, the validation aims to give an answer to the following question:

- *Is delaying messages an effective strategy to discover faults?*

In order to provide an answer to this question, we operated as follows. Figure 9 shows the time of delaying in milliseconds and the corresponding network bandwidth in Mbit/s. The nodes use wireless communication and Ericsson uses the international standard of Wireless 802.11n [24], whose normal bandwidth is 600 Mbit/s. This benchmark is used to adjust the network bandwidth when the system is performing

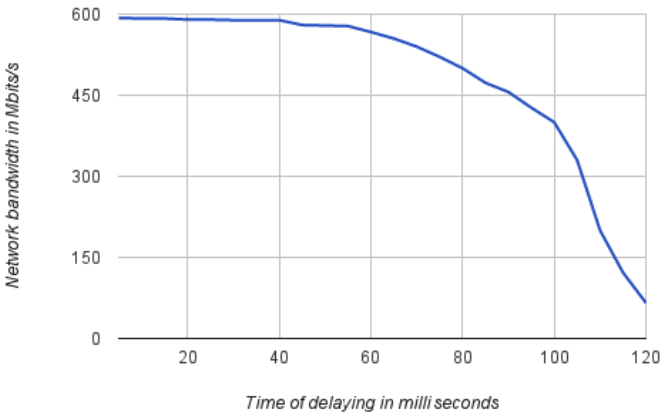

Fig. 9. Message delaying and corresponding network bandwidth

abnormally; the delaying mechanism is exploited to identify which process is responding slowly.

Randomized delays detect misconfigured timeout handling. This was not identified by manual testing since the testers made the same wrong assumptions of the developers, and consequently, the defect was missed. This testifies the potential existence of human bias in manual testing.

VIII. Discussion

In this section, we discuss the lessons learned (Section VIII-A), the quality of findings and what benefits could be gained from the study (Section VIII-B), and the challenges of adopting the fault injection approach (Section VIII-C).

A. Lessons learned

Randomized testing: Randomized testing can be successfully used to complement traditional testing since it removes potential developer bias and helps identifying unknown failure modes. This study indicated that fault injection can be feasible and applicable for testing the robustness and the dependability of the embedded distributed system.

Randomization challenges: First of all faults need to be more of the character of "nuisances" that the system should be able to handle. We are using here the term nuisance to identify delays, lost messages, invalid data, restarted instances, and so on. We had good results by sending random messages and through randomized delays. In fact, as described in Sections VII-A and VII-B, this permitted to identify some errors that were not identified by traditional testing. Then, we had the need of automatic use case validation to ensure the detection of errors introduced by the nuisances. Since nuisances are random, it is not simple to reproduce errors. Possible solutions are to use known seeds or to record and playback of events. Moreover, good traces are needed for post-mortem analysis of a crash.

Context-specific approach: The fault injection strategy depends on the context, although recurring patterns might be probably identified. In particular, when considering embedded systems, it is necessary to interact with domain experts to understand which kind of faults should be injected, and also personalize the implementation of specific faults.

Adoption goes beyond technicalities: Adoption in practice goes beyond technicalities, thus involving also organization, processes, and specificities of companies. Therefore, it is important to have a strong and continuous collaboration between academia and industry. Iterative development methodologies are a good strategy to enable continuous validation and collection of feedbacks from practitioners.

B. Quality of findings

Even though the quality of system has been proven to be quite high, unexpected faults that were not caught by traditional testing have been detected, such as:

- fuzzing of signals has detected a weakness in the system handling dynamically sized objects that caused a crash;
- randomized delays detected misconfigured timeout handling. It is important to highlight that the manual test case had the same mistake as the production code so this error was missed.

The faults were identified and diagnosed by observing the stack trace on the log files. Diagnosing the faults helped us also in identifying the bottleneck of the distributed system. This has been shown after injecting the message delay fault type on the weak points of the system. The identification of the weak points enables improvements in the system architecture. For instance, Postmonkey helped to improve the software architecture of the system. In a distributed system, there are always dependencies between the components and some dependencies are more critical than others. After a deep study of the distributed system architecture at Ericsson, we identified some critical components, whose failures can be potentially due to the strong dependencies. For such dependencies, we used message delaying as failure type for testing the timeout mechanism. Specifically, we delayed the messages between the critical components as well as tracked the system behavior through the log files. Under those circumstances, we have discovered that the system will not act as it should when delaying the messages at a specific time. In the long run, software architecture can be improved by having lower dependencies on those critical components. By having lower coupling, components will be easier to replace, the chance that a change in one component causes a problem in the other components will be reduced, which enhances maintainability and reusability, and the chance that a fault in one component causes failures in other components will also be reduced, which enhances robustness [25].

Moreover, detecting and diagnosing the faults also helped in discovering more faults to inject.

C. Challenges in adopting the fault injection approach

In this study, we have met some challenges and barriers to adoption of the fault injection approach.

Unrealistic results: Considering that we proposed a non-deterministic testing approach the result of running such technique can be misleading. Doing a complete randomization of the test cases might produce useless results since the generated test case can be unrealistic. In order to mitigate this limitation,

we made our test cases less randomized such as having time interval, specify the messages type, number of messages and delaying time. For instance, when we implemented the first fault type, which was sending random messages, the system acts always differently. That was due to the fact that the injection is performed on embedded distributed systems with limited computation power. Sending a lot of random messages without any time between them was misleading. We overcome this problem by inserting delays between each message sending. As a conclusion, having the time between each sending gave us a reasonable and realistic indication.

Negative side effects: When sending a lot of random messages without any time interval between them, the system will probably crash since the computation power is limited on the embedded components. As stated in [16] online testing could produce negative side effects out of the tester's control. It is then important to take into account also policies that help to prevent or ameliorate such adverse effects. Also, as stated in [16], online testing could have a negative impact on non-functional characteristics; this can be mitigated by trading off testing accuracy with performance.

The strategy we adopted to avoid undesired crashes of the overall system because of injected fault is as follows: (i) Delaying signals - we make sure the delays fall within what the system should handle; (ii) Fuzzing signals - we limit fuzzing to faking Confirm/Rejects which the system should discard.

IX. Conclusion

This paper proposes a fault injection approach to testing the robustness of distributed and embedded system with limited computation power. We performed the study in collaboration with Ericsson AB in Gothenburg, Sweden. This approach came as a complementary to traditional testing.

As described in the paper, the approach detected some unexpected faults. We observed also that fault injection can be adopted to test embedded distributed system and dependency bottlenecks can be identified. This approach provided the development team at Ericsson with a better validation technique that permits to reduce the number of unpredicted faults that appeared during the operations on the customer's sides.

As future work, we plan to replicate the study to other organizations within Ericsson and to other companies to better validate the generality of the approach. We plan also to investigate steps that will permit to deploy the approach on customer production networks.

Finally, additional fault types can be injected such as register and memory faults, killing process, CPU overloads, slow down network, fork bomb at a particular node, and drop network packets for a duration of period at a certain rate, etc. It would also be an area of interest to test a combination of faults. This can be done by injecting a combination of faults in a different order and at a different time.

References

[1] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, "Testing of fault-tolerant and real-time distributed systems via protocol fault injection," in *Proc. of Annual Symp. on Fault Tolerant Computing*. IEEE, 1996.

[2] F. Faghri, S. Bazarbayev, M. Overholt, R. Farivar, R. H. Campbell, and W. H. Sanders, "Failure Scenario As a Service (FSaaS) for Hadoop Clusters," in *Proc. of SDMCMM '12*. ACM, 2012.

[3] ISO/IEC/IEEE 24765:2010 Systems and software engineering - Vocabulary, "Available at: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-1:v1:en."

[4] A. Metzger, E. Schmieders, O. Sammodi, and K. Pohl, "Verification and testing at run-time for online quality prediction," in *Workshop on S-Cube*, June 2012, pp. 49–50.

[5] E. Fredericks, A. Ramirez, and B. Cheng, "Towards run-time testing of dynamic adaptive systems," in *In Proc of SEAMS2013*, 2013.

[6] A. Bertolino, G. D. Angelis, S. Kellomaki, and A. Polini, "Enhancing service federation trustworthiness through online testing," *Computer*, vol. 45, no. 1, pp. 66–72, Jan 2012.

[7] Y. K. Malaiya, "Antirandom testing: getting the most out of black-box testing," in *Proc of Int. Symposium on Software Reliability Eng.*, 1995.

[8] Ericsson, "Mobility Report," http://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf, 2015.

[9] ——, "High Availability is more than five nines," http://www.ericsson.com/real-performance/wp-content/uploads/sites/3/2014/07/high-avaialbility.pdf, 2014.

[10] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, "Statistical fault injection," in *Int. Conf. on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008.

[11] Chaos Monkey Testing (CMT), "http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html."

[12] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, and F. Törner, "Improving fault injection in automotive model based development using fault bypass modeling." in *GI-Jahrestagung*, 2013, pp. 2577–2591.

[13] "Apache Hadoop," http://en.wikipedia.org/wiki/Apache_Hadoop.

[14] "MapReduce," http://en.wikipedia.org/wiki/MapReduce.

[15] "Anarchyape," https://github.com/david78k/anarchyape/.

[16] M. Ali, A. Bertolino, F. De Angelis, G. De Angelis, D. Fani, and A. Polini, "An extensible framework for online testing of choreographed services," *Computer*, vol. 47, no. 2, pp. 23–29, Feb. 2014.

[17] C. B. Seaman, "Software maintenance: Concepts and practice," *J. Softw. Maint. Evol.*, vol. 20, no. 6, 2008.

[18] R. V. Binder, "Object-oriented software testing," *Commun. ACM*, vol. 37, no. 9, Sep. 1994.

[19] Q. Wang, F. Chen, H. Mei, and F. Yang, "An application server to support online evolution," in *Proc of Int. Conf. on Software Maintenance*, 2002.

[20] M. Canini, V. Jovanović, D. Venzano, D. Novaković, and D. Kostić, "Online testing of federated and heterogeneous distributed systems," in *Proceedings of SIGCOMM '11*. ACM, 2011.

[21] H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins, "Failure as a service (faas): A cloud service for large-scale, online failure drills," *University of California, Berkeley*, vol. 3, 2011.

[22] C. Woskowski, M. Trzeciecki, and F. Schwedes, "Assessing the applicability of the let-it-crash paradigm for safety-related software development," *Technical Report*, 2013.

[23] A. Collins, D. Joseph, and K. Bielaczyc, "Design research: Theoretical and methodological issues," *Journal of the Learning Sciences*, vol. 13, no. 1, pp. 15–42, 2004.

[24] "Wireless 802.11n," https://en.wikipedia.org/wiki/IEEE_802.11n-2009.

[25] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 3rd ed. Upper Saddle River, N.J: Addison-Wesley, 2012.