

UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica



tesi di laurea specialistica

Robustness testing di middleware DDS-compliant

Anno Accademico 2010/2011

relatore

Ch.mo prof. Domenico Cotroneo

correlatore

Ing. Gabriella Carrozza

candidato

Sergio Celentano

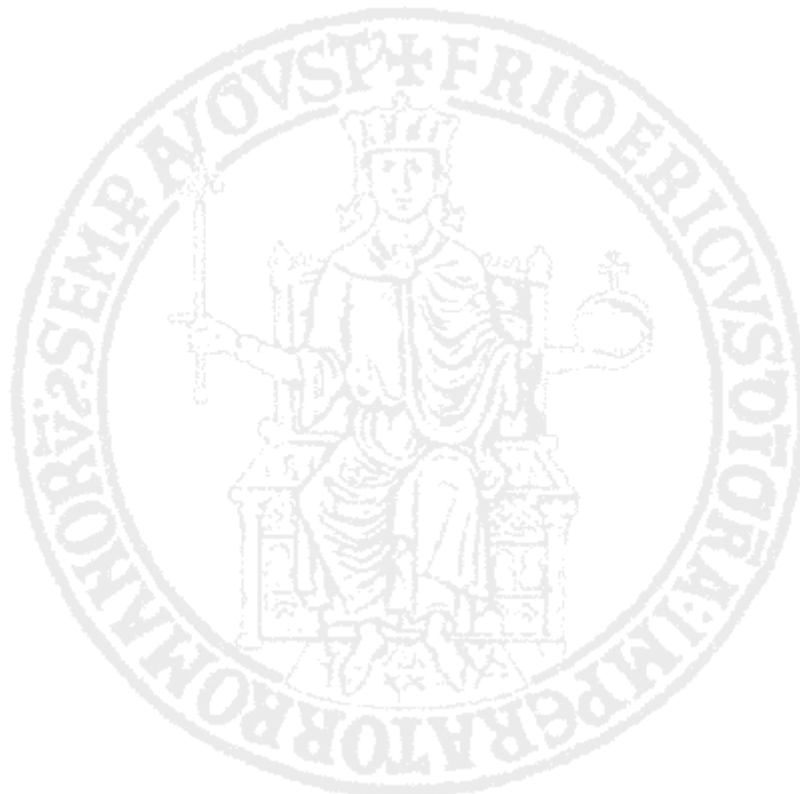
matr. 885/211

*Alla mia famiglia
Ai miei amici vicini e lontani*

Index

Introduction	5
Chapter 1. Dependability basics	8
1.1 Dependability attributes	9
1.1.1 Threat characterization	10
1.2 Robustness	11
1.2.1 Robustness fault sources	13
1.3 Existing approaches to robustness testing	15
1.3.1 Black-box testing	16
1.3.2 White-box testing	16
1.3.3 Template-based type-specific test	17
1.3.4 Mutation-based sequential test	17
1.4 Metrics for robustness assessment	18
Chapter 2. The proposed approach: Java fault injection tool (JFIT)	20
2.1 Overview on fault injection tools	20
2.1.1 The Ballista tool	20
2.1.2 SFIT	21
2.1.3 Jaca	23
2.1.4 Limitations of existing tools	24
2.2 Our approach to robustness testing of COTS system	25
2.3 Fault model	27
2.4 The proposed approach: the Java Fault Injection Tool (JFIT)	29
2.4.1 Features in general	29
2.4.2 Architecture	31
2.4.3 OLAP Database	34
2.4.4 Dynamic analysis: the experiment run in detail	36
2.5 Results example	38
2.6 Implementation details	42
2.6.1 Javassist library	43
2.6.2 The interception mechanism	43
2.6.3 Implementation issues	46
2.7 Limitations	47

Chapter 3. Middleware for Data Distribution	48
3.1 Publish - Subscribe paradigm	48
3.2 DDS technology (a brief description)	49
3.2.1 OMG DDS Conceptual Model	53
3.2.2 OMG DDS Profiles	55
3.3 PrismTech OpenSplice	56
Chapter 4. The experimental campaign	59
4.1 Preparation phase	59
4.2 Analysis of DDS API specification	60
4.2.1 API coverage	63
4.2.2 Data collection issues	66
4.3 Real world scenarios	67
4.3.1 SWIM-BOX	67
4.3.2 PrismTech Touchstone	69
4.4 Test-bed description	70
4.5 Tests execution	72
4.6 Results presentation	75
4.7 Results analysis	78
Conclusions and future works	80
Acknowledgments	82
Bibliography	83



Introduction

Hardware and software technologies are dramatically increasing the complexity of modern computer systems. Their fast progresses allows the development of very complex systems, in terms of both the mission they aim to fulfill and their dimension. Such advanced technological means makes possible to employ large scale and distributed infrastructure in a variety of application fields, including critical scenarios where the presence of strict dependability requirements requires to predict systems behaviour as much as possible in order to avoid unexpected failures. On the software side, current technologies allow to develop very complex programs according to modular architectures, exploiting the possibility of combining Off-The-Shelf (OTS) software items, i.e., modules that are already available on the market and *ready to use*. On the one hand, these allow industries to be competitive by reducing the development costs and the time to market. On the other, their usage introduces a twofold challenge: i) they are typically designed for general purposes and without dependability in mind; ii) they are likely to behave differently when used in real world scenarios rather than in testing environments due to the integration with (even dozens of) other modules from different vendors.

A clear example of advanced technological means enabling systems integration, interoperability, and increasing complexity comes from the platforms implementing the Data Distribution Service (DDS). DDS is a Publish-Subscribe standard for real-time, dependable and high performance data exchange able to obtain a loosely coupled form of

interaction by producing an immediate benefit when the system dynamically increases its dimension. Furthermore, since DDS adopts a data-centric model, the routing of the messages depends on the data they are delivering instead of their source and destination. For these reasons many mission-critical systems, such as aerospace/defense, banking/finance, healthcare/insurance, e-commerce and telecommunication applications, have been using DDS as the underlying data delivery and distribution mean.

While cost and development time is a common consideration for general purpose systems, the robustness of the software -the degree to which a component works correctly in the presence of invalid inputs or stressful environmental conditions- is almost always a major concern for mission and safety critical applications as the examples listed above. It is important that these applications are resistant to failures caused by abnormal inputs. Although this is a ground truth, not so much work has been done so far to assess DDS robustness degree in the context of real world mission and safety critical applications.

Among the purposes of this thesis work there's the definition of a methodology to assess the robustness of a DDS compliant middleware, and the development of a software tool through which this evaluation can be automatically performed. The tool has been implemented for a given implementation of DDS, namely OpenSplice by PrismTech, while the defined methodology is general enough to be applied to any kinds of DDS compliant platform.

All the thesis activity has been leaded at SESM company facilities in Giugliano in Campania, near Napoli. SESM is involved in several activities at the moment that concern the assessment of availability and reliability (e.g. *safety*) of software and middleware for Air Navigation Service (ANS) systems.

The first chapter of this thesis gives an overview of the software dependability theory focusing then on the main topic of this work, the robustness.

Chapter 2 illustrates the approach followed to automatically assess the robustness of OTS

systems both from a theoretical and technical point of view. A big effort has been put in the design process, so that the testing tool could address as much as possible all the requirements that had already stated. Details about the implementation process will also be given there.

Chapter 3 describes the system under examination more in details, starting from the DDS technology and focusing in the last part on the middleware identified as our study case, PrismTech's OpenSplice.

Chapter 4 is finally devoted to describe the experimental campaigns as well as to discuss the achieved results.



Chapter 1

Dependability basics

Dependability is a complex attribute whose definition changed several times in the last decade. We can state that dependability is the *ability to avoid service failures that are more frequent and more severe than is acceptable* [1].

The increasing complexity of systems has caused dependability to become a major concern, encompassing several aspects, from safety to security. Here the focus is on software dependability where faults can be related to design faults or human mistakes.

The dependability is a composed quality attribute, that encompasses the following sub-attributes:

- Availability: readiness for correct service;
- Reliability: continuity of correct service;
- Safety: absence of catastrophic consequences on the user(s) and the environment;
- Confidentiality: absence of improper system alterations;
- Maintainability: ability to undergo modifications and repairs.

In this chapter we first focus on some general aspects about software dependability, then we move to the main subject of this thesis work, the robustness. Finally we present a summary of the available techniques to perform the robustness testing.

1.1 Dependability attributes

The dependability attributes can be formalized mathematically, and basic measures have been introduced in charge of quantifying them.

The reliability, $R(t)$, was the only dependability measure of interest to early designers of dependable computer systems. It is the conditional probability of delivering a correct service in the interval $[0, t]$, given that the service was correct at the reference time 0:

$$R(0, t) = P(\text{no failures in } [0, t] | \text{correct service in } 0)$$

Let us call $F(t)$ the unreliability function, i.e., the cumulative distribution function of the failure time. The reliability function can thus be written as:

$$R(t) = 1 - F(t)$$

Since reliability is a function of the mission duration T , mean time to failure (MTTF) is often used as a single numeric indicator of system reliability. In particular, the time to failure (TTF) of a system is defined as the interval of time between a system recovery and the consecutive failure.

As for availability, they say a system to be available at a the time t if it is able to provide a correct service at that instant of time. The availability can thus be thought as the expected value $E(A(t))$ of the following $A(t)$ function:

$$A(t) = \begin{cases} 1 & \text{if proper service at } t \\ 0 & \text{otherwise} \end{cases}$$

In other terms, the availability is the fraction of time that the system is operational. The measuring of the availability became important with the advent of time-sharing systems. These systems brought with it an issue for the continuity of computer service and thus

minimizing the “down time” became a prime concern. Availability is a function not only of how rarely a system fails but also of how soon it can be repaired upon failure. Clearly, a synthetic availability indicator can be computed as:

$$Av = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}$$

where $MTBF = MTTF + MTTR$ is the mean time between failures. The time between failures (TBF) is the time interval between two consecutive failures. Obviously, this measure makes sense only for the so-called repairable systems. $R(t)$ and $A(t)$ are the dependability attributes of major interest for this dissertation. A complete dissertation about dependability fundamentals can be found in [2], along with a description of dependability measures.

1.1.2 Threat characterization

The causes that lead a system to deliver an incorrect service, i.e., a service deviating from its function, are manifold and can manifest at any phase of its life-cycle. Hardware faults and design errors are just an example of the possible sources of failure. These causes, along with the manifestation of incorrect service, are recognized in the literature as dependability threats, and are commonly categorized as failures, errors, and faults.

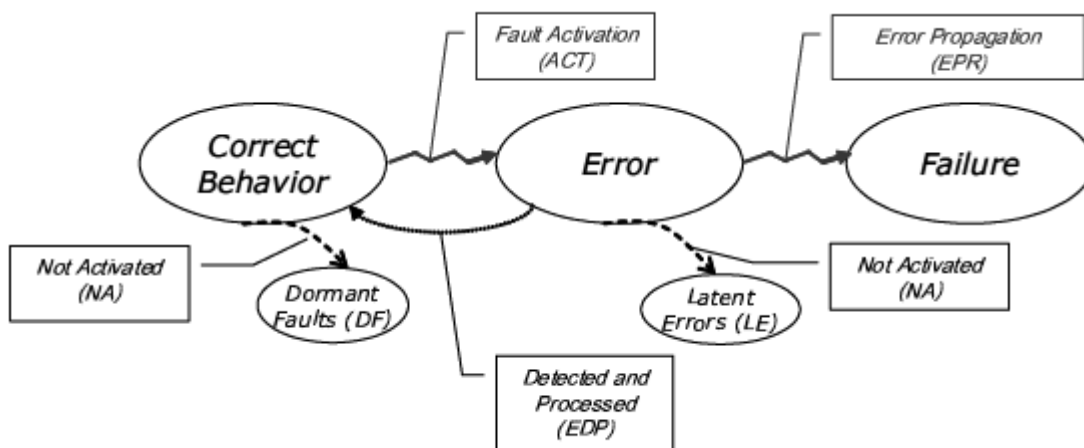


Figure 1.1-Failures, error and faults

A failure is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. A service failure is a transition from correct service to incorrect service, i.e., to not implementing the system function. The period of delivery of incorrect service is a service outage. The transition from incorrect service to correct service is a service recovery or repair. The deviation from correct service may assume different forms that are called service failure modes and are ranked according to failure severities.

An error can be regarded as the part of a system's total state that may lead to a failure. In other words, a failure occurs when the error causes the delivered service to deviate from the correct service. The adjudged or hypothesized cause of an error is called a fault. Faults can be either internal or external of a system, and they can be classified in several ways (e.g., basing on their nature, or the way they manifest in errors). Failures, errors, and faults are related each other in the form of a chain of threats, as sketched in figure 1. A fault is active when it produces an error; otherwise, it is dormant.

An active fault is either i) an internal fault that was previously dormant and that has been activated, or ii) an external fault. A failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. An error which does not lead the system to failure is said to be a latent error. A failure of a system component causes an internal fault of the system that contains such a component, or causes an external fault for the other system(s) that receive service from the given system.

1.2 Robustness

The software robustness degree assesses its capability to operate in a correct way even in presence of not valid inputs or very stressful workloads [3]. Taking into account for the moment just the first event, we can say that a software can be defined robust if, in case of invalid inputs, it keeps executing the task declared in the specifications or, in case an exception occurs, it returns an information consistent with the failure reason [4].

On the contrary, among the non-robust behaviors we can put all the failures events that corrupt partly or completely the system environment: catastrophic failures, thread hangs and thread abort, unknown exception raising, false success, misleading error information.

Robust behaviors	Successful return (no exceptions) Raise CORBA exception
Non-robust behaviors (Robustness failures)	Computer crash (Catastrophic failure) Thread hang (Restart failure) Thread abort (Abort failure) Raise unknown exception False success (Silent failure) Misleading error information (Hindering failure)

Figure 1.2 - Robust and non-robust behaviors

This discrimination will let us make distinguish between robust or non-robust response in the following work.

Robustness testing is a technique that can help designing a system which is more tolerant to exceptional inputs. Robustness is also defined as the degree to which a system can function in the presence of invalid inputs or stressful environmental conditions. These are exceptional conditions.

Exceptional conditions are things that occur in a system that are not expected or are not a part of normal system operation. When the system handles these exceptional conditions improperly, it can lead to failures and system crashes. Exception failures are estimated to cause two thirds of system crashes and fifty percent of computer system security vulnerabilities. Exception handling is especially important in embedded and real-time computer systems because software in these systems cannot easily be fixed or replaced, and they must deal with the unpredictability of the real world.

According to [5], robust exception handling in software can improve software fault tolerance and fault avoidance, but no structured techniques exist for implementing dependable exception handling. However, many exceptional conditions can be anticipated when the system is designed, and protection against these conditions can be incorporated into the system.

Traditional software engineering techniques such as code walkthroughs and software testing can be taken into account to identify more exceptional conditions, such as bad input for functions and memory and data errors. However, it is impossible to cover all exceptional cases. It is also difficult to design a dependable system that can tolerate truly unexpected conditions. In these cases, some form of graceful degradation is necessary to safely bring down the system without causing major hazards.

1.2.1 Robustness fault sources

The first step of developing the test strategy was the identification of the potential sources for activating robustness faults in the middleware. [6] gives a significative classification to the possible fault sources inside a middleware:

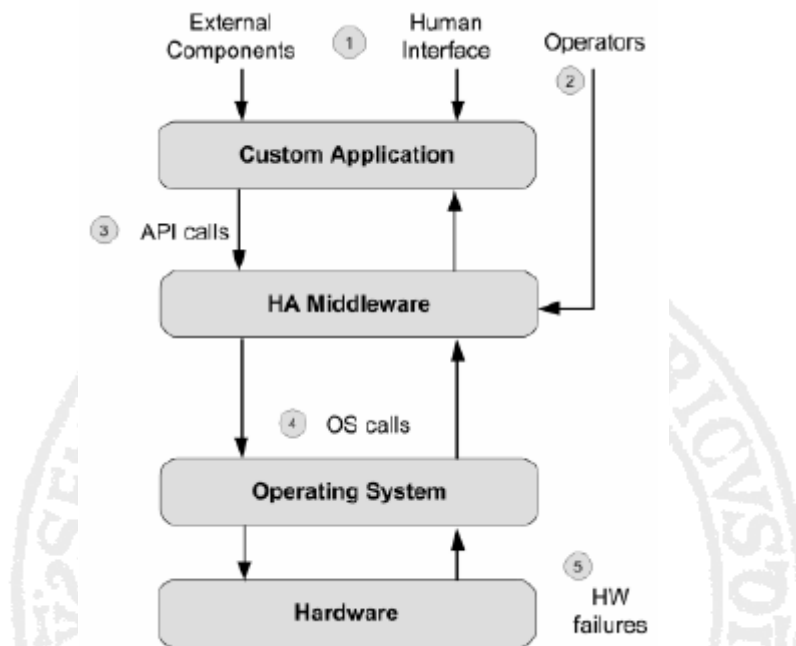


Figure 1.3 - Possible fault sources

Some effective scenarios in which robustness experiments may take place are:

Configuration, maintenance.

A special class of faults are related to errors in configuration files, such as missing or wrong paths, incomplete or completely missing data.

User interface failures.

In this case faults come from system misuse made by the users.

Callee interface fault injection.

Faults may be submitted at the API layer as stated above, this aims to check out-of-specifications responses. The assumption here is that some external software defect may feed an exceptional value to the module under test. This approach is most useful for very widely used application programming interfaces (APIs) for which all uses cannot possibly be foreseen, as well as for faults that may be related to programming errors (e.g. submitting a null value as a pointer parameter to a procedure call).

Caller interface fault injection.

Faults can be injected in values returned to the module under test from other software that it calls through external calling interfaces.

Forced resource exhaustion and environmental exceptions.

Exceptional conditions such as having no more allocable memory or bad connection across the network may arise some unexpected behavior in the system under test. This may be seen as a special case of fault injection into the external caller interface as presented earlier, but in general is an important case to cover.

Failure of cooperating processes.

It affects cooperating processes, tasks and computers. Some exceptional inputs may come from these sources too.

1.3 Existing approaches to robustness testing

Robustness assessment can be performed through fault injection technique. *Fault injection is a phrase covering a variety of techniques for inducing faults in systems to measure their response to those faults. In particular, it can be used in both electronic hardware systems and software systems to measure the fault tolerance of the system.* Many different approaches and tools have been developed in order to submit effective faults (both hardware and software) inside the system. Hardware-implemented fault injection uses exceptional inputs into the target system's hardware and it can be spread into two main categories:

- *Hardware fault injection with contact.* The injector has direct physical contact with the target system, producing voltage out-of-range values or changes at the pin-level.
- *Hardware fault injection without contact.* The injector has not direct physical contact with the target system and the faults are activated by electromagnetic interferences causing currents inside the target chip.

In recent years, researchers have taken more interest in developing software-implemented fault injection tools as their development don't require expensive hardware and can address targets inside software code more precisely. According to [7] we can categorize software injection methods on the basis of when faults are injected:

- *Compile-time injection.* This method injects errors into the source code or assembly code of the target system to emulate the effect of hardware, software and transient faults. The modified code alters the target programs instructions causing the failures to be raised.
- *Runtime injection.* The mechanism here is different. A trigger is necessary to activate the fault injection phase. Commonly used techniques are:
 - *Time-out.* A timer expires at a defined time and it triggers the injection by generating an interrupt used to invoke fault injection phase.
 - *Exception/trap.* A hardware exception or a software trap transfer control to the

fault injector mechanism. Unlike the previous case the injection here may take place at any time, depending on some conditions that may occur.

- *Code insertion.* In this technique instructions are added to the target program and allow fault injection to occur before certain instructions, similarly to code-mutation mechanism. Unlike code-mutation, code insertion adds instruction rather than changing original instructions and this takes place at run-time instead of compile-time.

1.3.1 Black-box testing

The black-box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure. It is also termed data-driven, input/output driven, or requirements-based testing.

Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing -- a testing method emphasized on executing the functions and examination of their input and output data. The tester treats the software under test as a black box -- only the inputs, outputs and specification are visible, and the functionality is determined by observing the outputs to corresponding inputs. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification. No implementation details of the code are considered.

1.3.2 White-box testing

Contrary to black-box testing, software is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure.

Control-flow testing, loop testing, and data-flow testing, all maps the corresponding flow structure of the software into a directed graph. Test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may discover unnecessary "dead" code -- code that is of no use, or never get executed at all, which cannot be discovered by functional testing.

According to these assumptions we may divide the robustness testing into two main families:

- Template-based type-specific test, as a black-box testing case.
- Mutation-based sequential test, as an approach to white-box testing.

1.3.3 Template-based type-specific test

An automatic tool probes the behavior of each API function giving to it an intentionally wrong value taken from a set of parameters. Rather than define the exceptional cases one by one for each function, the exceptional values are selected starting from the input parameters data type. By knowing these data types the tool calls the function with all the possible combinations foreseen for the given data type. Each faulty value is submitted in a different test case so that the failure reason can be easily detected from the log file which will be taken into account during the analysis phase.

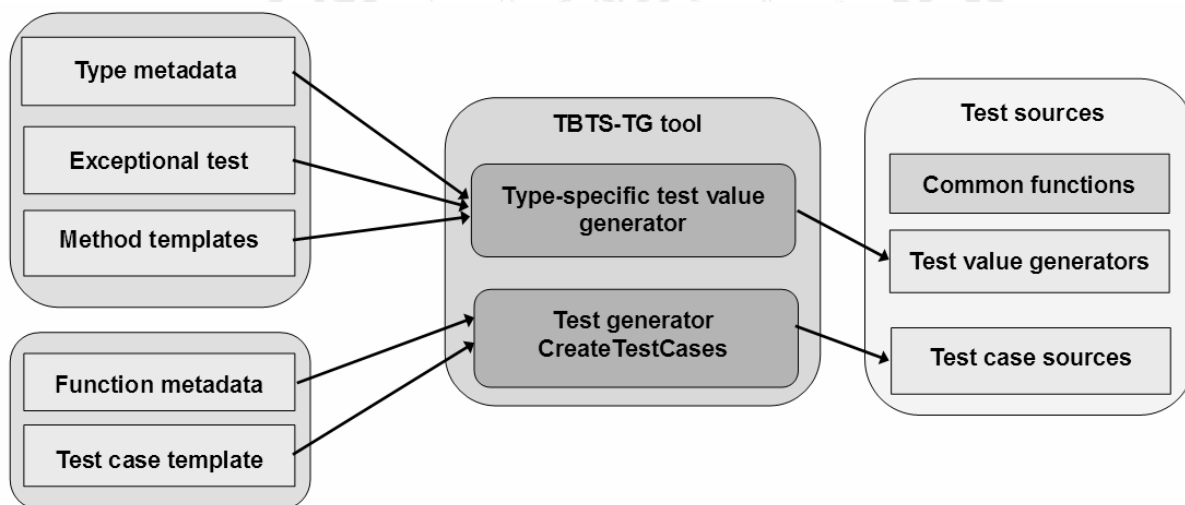


Figure 1.4 - Template-based type-specific test

1.3.4 Mutation-based sequential test

the idea behind this approach consists in reproducing inside the program some typical programming errors (mutation operators) such as the omission, wrong placing or even method call swapping, missing or replaced parameters. Here the automatic injection of the fault into the source code is achieved by tools which first infer the syntactic structure of the program and then apply the mutation operator.

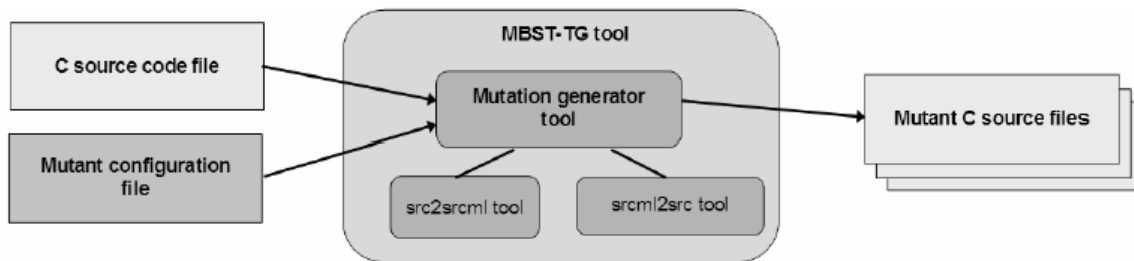


Figure 1.5 - Mutation-based sequential test

Code mutation usage in fault injection has a close intent to “bebugging”. For instance, introducing known faults via source code mutation may help to assess the rate of detection and removal of a particular test methodology.

Given the context in which the middleware operates and the black-box analysis we’re going to carry out, we went for the first approach and so we focus our study on the robustness assessment through the template-based type specific test methodology.

1.4 Metrics for robustness assessment

Automatically generating software tests requires three things: a MuT (module under test), a machine-understandable specification of correct behavior, and an automatic way to compare

results of executing the MuT with the specification [8].

Unfortunately, obtaining or creating a behavioral specification for a COTS or legacy software component is often impractical due to unavailability or cost.

Fortunately, robustness testing need not use a detailed behavioral specification. Instead, the almost trivial specification of “doesn’t crash, doesn’t hang” suffices.

The robustness of the responses of the MuT can be characterized as:

- robust (neither crashes nor hangs, but is not necessarily correct from a detailed behavioral view),
- having a reproducible failure (a crash or hang that is consistently reproduced),
- and an unreproducible failure (a robustness failure that is not readily reproducible).

The response of the module under test can also be measured according to a severity scale.

One very common example is provided by the CRASH scale. In this scale the response lies in one of six categories:

- Catastrophic, the system crashes or hangs.
- Restart, the test process hangs (it must be restart to work properly again).
- Abort, the test process terminates abnormally.
- Silent, the test process exits without an error code, when one should have been returned.
- Hindering, the test process exits with an error code not relevant to any exceptional input parameter value.

In addition to these cases the module under test may exit with an error code and if it matches some reference behavior this event will be considered as a “passed test”.

Chapter 2

The proposed approach: Java fault injection tool (JFIT)

2.1 Overview on fault injection tools

Now we present some tools used to perform robustness evaluation of COTs which inspired our fault injection software JFIT and also provide cases where fault injection methodology successfully addressed robustness assessment. They both fall in the category of runtime/code-insertion technique and take advantage of Java language facilities to submit faults at API level.

2.1.1 The Ballista tool

The Ballista testing framework is designed to test COTS (Commercial off-the-shelf) software modules for exception-handling robustness problems triggered by invalid inputs, some experimental results can be find in [8]. The use of this tool is effective in all those contexts where the use of COTS components is massive and the details about the implementation are not available. For example, Ballista testing can find ways to make operating systems crash in response to exceptional parameters used for system calls, and can find ways to make other software packages suffer abnormal termination instead of gracefully returning error indications.

The robustness methodology is based upon using combinational tests of valid and invalid

parameters values for system calls and functions. In each test case the MuT (module under test) with the faulty value submitted at its interface. The MuT can be a stand-alone program, function, system call, method or any other software that can be invoked with a procedure call [8].

In most cases the MuTs are single calling points into an API. Each test can determinate whether the MuT provides a robustness behavior to some parameter values. These parameter values are drawn from a pool of normal and exceptional values based on the data type of argument passed to the MuT.

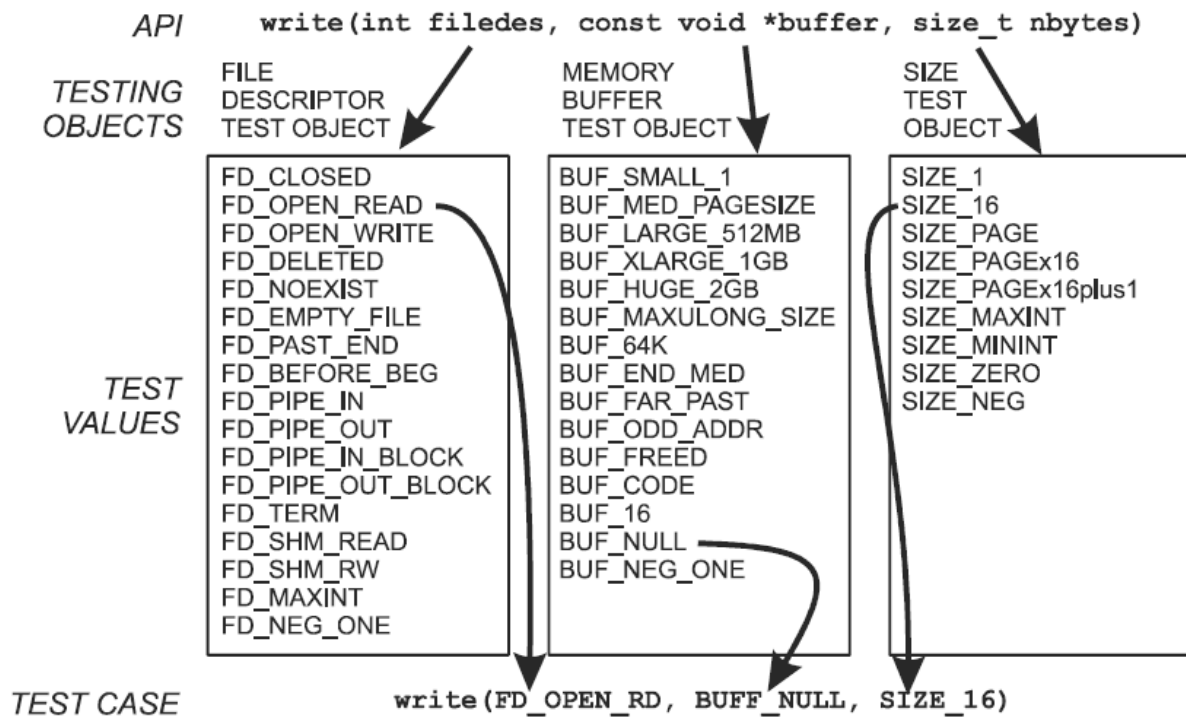


Figure 2.1 - The Ballista tool

2.1.2 SFIT

SFIT [9] is a tool for automatic robustness testing completely based on Java technology which makes use of computational reflection. As it is not often possible to test thoroughly COTs for robustness as their source code is usually not available, this tool adopts some workarounds to interrogate COTs for method interfaces including public, private and

protected ones. This information will be used later to identify which will be the targets of the faults to be submitted.

The tool consists of a number of interrelated components:

- Class inspector. As its name suggests, the main purpose is to investigate COTs in order to obtain details about the COTs interface. The classes names will be the target of the fault injection campaign in a latter moment.
- Fault setting. The definition file that the user may fill with the information concerning where the faults should be submitted, how many faults injected as well as what faults to inject.
- Loader and generator. These components are in charge of automatically generate the code that the user will use to exploit the fault injection process.
- Fault injector. Its purpose is to load the generated code into the Java Virtual Machine (JVM) in order to inject the faults as specified by the test cases in the fault setting.
- Log is the database used to perform offline analysis.

The whole software structure is summarized in the figure below:

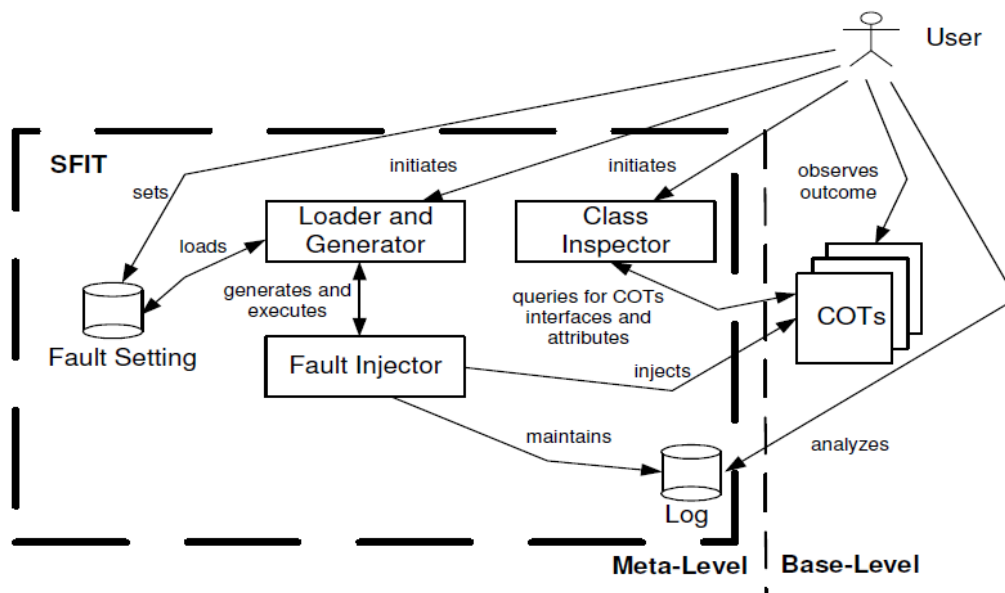


Figure 2.2 - SFIT

SFIT has been experienced to perform robustness testing of JAVA COTs, called Jada, a Linda tuple space implementation.

2.1.3 Jaca

JACA [10] is a software fault injection tool that validates object oriented application written in Java. As SFIT its objective is to reduce tool correctness assessment time in terms of robustness. Jaca is an evolution of another very well known fault injection tool, FIRE that used reflective programming to inject faults into C++ applications. Jaca relies on another toolkit in which reflection is introduced at bytecode level.

Furthermore JACA tool is written around a Fault Injection Pattern System that helps in case of new features to be added. This model helps when a new fault injection tool must be realized because it provides a structure that we often find in a program like that. According to this structure its main elements are described below:

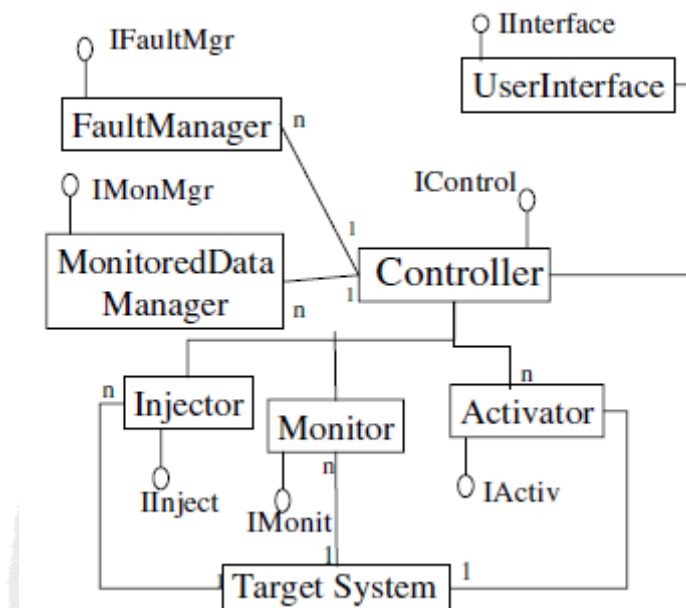


Figure 2.3 - Jaca

- Controller. It coordinates all other components.
- Injector. It's in charge of submitting faults into the system under test.
- Monitor. It collects all the readout about the behavior of the target system.
- Activator. It activates the target system.
- UserInterface which allows the user to specify the experiments, and exploit the results.

Among them just Injector, Monitor and Activator are the only components which communicate directly with the target system during the experiment. In addition to these elements we have a Fault Manager that stores the information about the faults to be injected.

JACA has been used to validate some object oriented applications as presented in [3] such as a OO DBMS component.

2.1.4 Limitations of existing tools

Both SFIT and JACA look to suit our case and both of them offer certain characteristics which were very appreciated in our study:

- Runtime code insertion. As we planned to perform a black-box evaluation of the middleware we can't rely on its source code. There's the need of some way to catch the calls to API methods in a transparent manner without affecting the original code. Both JACA and SFIT offered the opportunity.
- Some case studies were available about successfully robustness campaigns leaded against some complex APIs, as stated in [9] and other papers.
- JACA more than the other looked to be the best choice due to its modular architecture. We might use it to add some other facilities to be used in our work.

As only JACA were available at the moment of the study, we went for it. Given that, we encountered many problems with JACA when we tried to use it to lead some experiments.

- JACA documentation was not so user-friendly or completely missing about some aspects.
- Source code was not available, so that further modifications were impossible to be made.
- Our scenario was too complex to be integrated with JACA. As we needed to use a benchmark application as a workload, given the complexity of the DDS middleware

API itself and our ideas about the usage scenarios, we considered to save just a few ideas from JACA and SFIT. Their use “as is” was not so convenient as it looked at the beginning of the study.

The tools presented above inspired two main idea we successfully used later in the design phase:

- The use of Javassist library to perform the interception mechanism as we will illustrate more ahead.
- The necessity of a modular architecture which might be useful in the future to enrich the tool with other capabilities.

2.2 Our approach to robustness testing of COTS system

In this thesis work we discuss an approach to robustness evaluation of Prismtech's OpenSplice, a DDS-compliant middleware, first considering the design of a tool which may automate this process and then its implementation.

Many studies have been considered to address this problem, among them [11] spreads the methodologies used over the past years to assess the software dependability and more specifically the robustness.

According to it, our software fault injection experiments will consist of injecting a fault into the data or program of a module and evaluating whether the result matches that of a so called golden run. This approach is supported by many considerations: first of all, the software fault injection is by far the cheapest way to lead these experiments and also a very little knowledge of the software to be tested is required. Furthermore, a behavioral model of the software must be highlighted. This is achieved by stating what we consider as a dependable (and then robust) behavior, which typically means that the software is not vulnerable to any kind of attack or, in case of failure, it returns a report consistent with the failure raised.

Time to market and development costs typically reduce the quality of software and they prevent developers from producing reliable programs, but some good programming rules

can be adopted to reduce as much as possible failures occurrences.

Still in [11] we find that the general idea behind this optimizations is to avoid repeating validity checks and avoid providing unnecessary exception handlers. If a set of related modules is designed to be used together, such checks can be optimized across module boundaries and only invoked at external points into the set of modules.

This statement gives a preliminary direction to our work. Possible defects may lie in the external layers of the module under test of which we want to assess the robustness degree.

Moreover if there are no correctness defects in the module under test, this can only correspond to incorrect data being presented at the external inputs of the module, that is, an ordinary software module is defective if it gives a different answer from the expected one.

To be more precise, any study about the robustness assessment must verify if such a failure raised by a fault injection test may take place in a real system which operates in effective operative conditions [12] and if it's reasonable that the developers might have ignore it at the development time.

In effect there's the risk that some experiments might not reflect real use cases and hence be not representative. Thus the fault injection technique is a useful tool to evaluate the software robustness only if it is used under some restrictions.

According to this analysis we list here the guidelines we followed approaching the design phase of the fault injection tool:

- Representativeness. The faults that we take into account will reflect real use cases, first we will state if they can be originated by an effective use of the APIs.
- Scenario. The faults will emulate a not appropriated use of the Opensplice's APIs from an overlying application in terms of erroneous parameters submitted at the public methods layer.
- Black-box approach. We won't modify the API's source code so that the target of our robustness campaigns is the same that a real-world application might use.
- Automation. The choices we have made are automation-oriented, that is, when possible, we designed the tool so that the tests would be repetitive several times.

2.3 Fault model

When a study of robustness is going to be approached, one of the first issue to deal with is to define a suitable fault model that will drive our next choices. To this aim, we present here some assumptions based on the experience with the middleware APIs and the works related to robustness assessment.

Potential causes.

The faults that we will take into account will reflect effective use cases, in other words we will state first if they can have origin from a real use of middleware APIs. So, the faults will emulate a not appropriated use of the interface methods (APIs) made by an overlying application in terms of erroneous parameters submitted to the middleware public functions.

Component.

Only the JAVA implementation of the DDS middleware APIs will be taken into account in this study. This doesn't represent such a big limitation as all the Opensplice APIs use the same core set of underlying methods. The interception mechanism will be placed outside the middleware APIs, this is for two main reasons. First the representativeness of the fault injection study as stated above and then because we're going to harden the *validity checks* that are likely placed in the more peripheral layers of the middleware APIs structure.

Duration.

We may model the occurring faults according another set of dimension. With respect to the duration of each fault we may say that the faults here are *continuous*, i.e. deterministically applied between the trigger activation (the interception mechanism) and the termination of the experiment .

Erroneous values.

It is not possible to exhaust the input space, but it is possible to exhaustively test a subset of the input space. Here the erroneous values to submit at API level are taken from the literature. [4] worked as a model, but some faults couldn't be considered as Java language already provides mechanisms to filter out of boundaries values.

Type	Value	Description
Boolean	true	Replace the original value by "true"
	false	Replace by "false"
int	0	Replace by 0
	1	Replace by 1
	10	Replace by 10
	100	Replace by 100
	1000	Replace by 1000
	-1	Replace by -1
	-100	Replace by -100
	-1000	Replace by -1000
	2147483647	Replace by the maximum valid value for integers
	-2147483648	Replace by the minimum valid value for integers
long	0	Replace by 0
	1	Replace by 1
	-1	Replace by -1
	9223372036854775807	Replace by the maximum valid value for long type
	-9223372036854775808	Replace by the minimum valid value for long type
float	0	Replace by 0
	1	Replace by 1
	-1	Replace by -1
	3.4028235E38	Replace by the maximum value for float type
	1.4E-45	Replace by the minimum value for float type
string	null	Replace by null value
	empty string	Replace the original string by an empty string
	large string	Replace by a 64KB long string
	\n\n\n\n\n\n\n\n\n	Replace by non printable characters
	wrong string	Replace the original string by a wrong one
	#\$%+ '?! \ ~ *	Replace by non alphanumeric string
byte	null	Replace by null value
java.lang.Object	null	Replace by null value

Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary value analysis requires one or more boundary values selected as representative test cases.

2.4 The proposed approach: the Java Fault Injection Tool (JFIT)

2.4.1 Features in general

The above features motivate why we placed the tool in an intermediate layer between the workload application and the middleware as showed in the next figure:

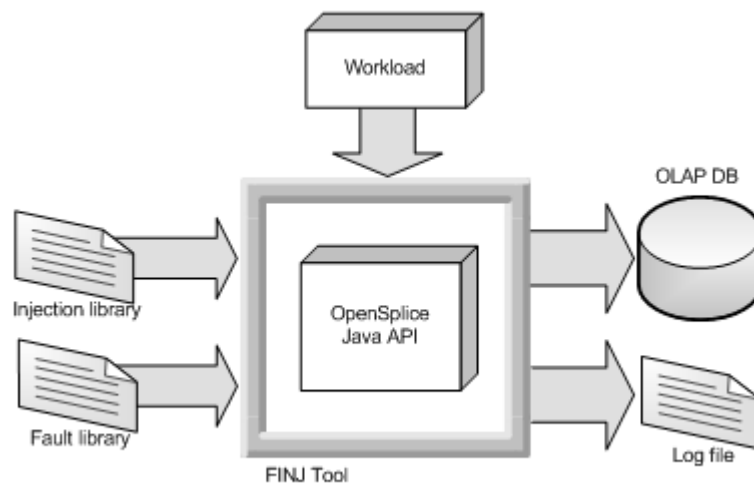


Figure 2.4 - The Java Fault Injection Tool

Our JAVA fault injection tool main features are:

Not intrusive interception mechanism.

Our fault injection tool acts as a wrapper layer of DDS middleware APIs. It intercepts the calls to the public interface methods, then substitutes the original value with the faulty one taken from the fault library presented above and lastly records the exception raised by the underlying level.

Totally automatic faulty parameters submission.

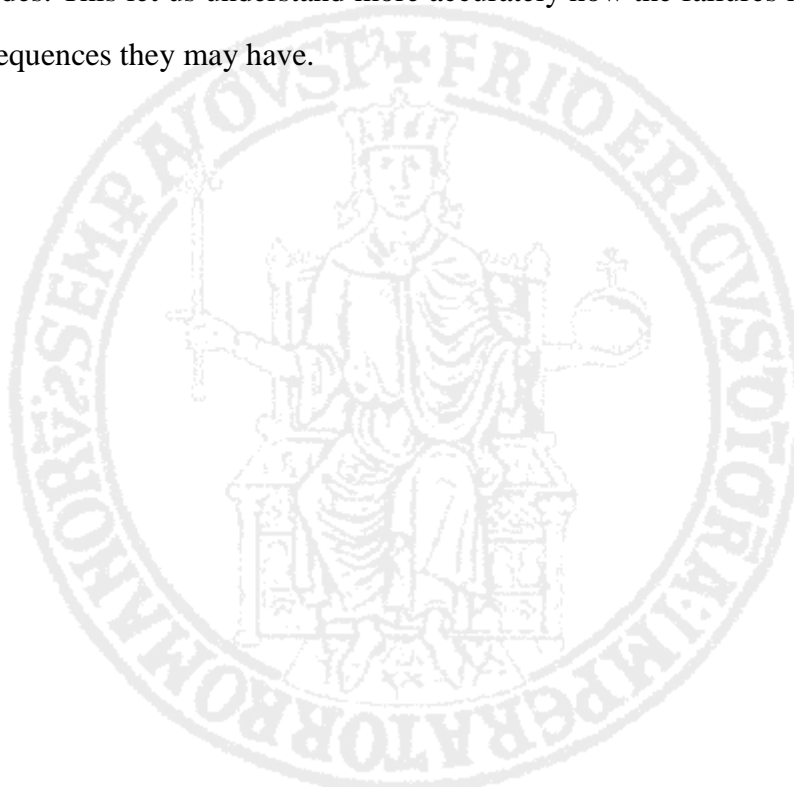
Once the list of the methods to be tested is filled, the campaign is completely automated. The experiments results are collected in separate log files and the more relevant data are stored in the OLAP database to simplify further analysis.

Real-world workload application.

Another important factor of this tool lies in the representativeness of the scenario used to lead the experiments. Here we considered the benchmark Touchstone as our workload application. Touchstone by Prismtech is a scenario-driven open source benchmarking framework for evaluating the performance of OMG DDS compliant implementation (at the moment only available for DDS Opensplice).

Two different usage scenarios.

The main scenario consists of two entities, one transmitting some messages (transmitter) that another one (receiver) will get. This scenario lives in only one machine as the both entities are modeled as threads. But, as DDS compliant middlewares are often involved in distributed environments where many actors are part of the communication, we also foresaw a version of our testing tool in which transmitter and receiver are placed in two different nodes. This let us understand more accurately how the failures may propagate and which consequences they may have.



2.4.2 Architecture

The designing phase of this tool was inspired by some guidelines so that its functionalities may be expanded in the future. To assess this requirement its structure is organized as follows where some elements have already been presented before.

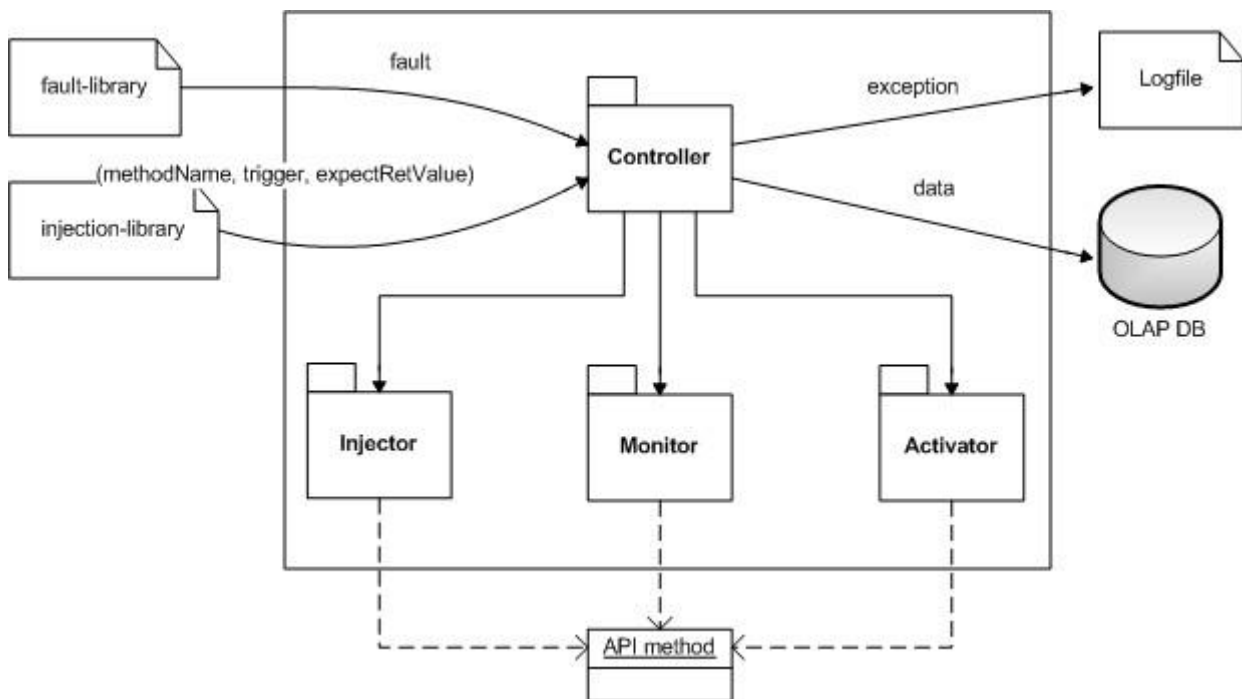


Figure 2.5 - JFIT architecture

This model quite reflects what we already have in JACA tool and SFIT but the different elements here are implemented so that they could work well in our scenario.

Just some of these elements are in contact with the MuT (i.e. the API method during each run):

Injector.

This class is in charge to perform the fault injection process. It works in pair with another class “Interceptor” which realizes the interception mechanism. Any time a method, whose signature corresponds to the one present in the injection library (see more ahead) is called by the workload application, the Interceptor class recognizes it and substitutes the value of one of its parameters with the foreseen one listed in the fault library.

Monitor.

This class accomplishes the monitoring task. It logs the experiments results in a text file, so that they can be analyzed more accurately later. One example of these results will be presented in the next paragraph.

Activator.

It works so that the class with the method to intercept is replaced by another one where the bytecode relative to the interception mechanism is included. This is possible through the class loading mechanism offered by Javassist library.

Controller.

It acts as the orchestra director. First it collects all the necessary data so that the previous classes can perform the injection process (among them the list of method to be injected, the erroneous values we want to submit and many others). Then it produces an xml document which will lead all the foreseen tests during the campaign. Following this document the Controller activates the workload so that it can reach the necessary state to perform the injection (i.e. the method call we want to use as a vehicle to the fault submission). Then it collects the data recording them into the DBMS (see more ahead).

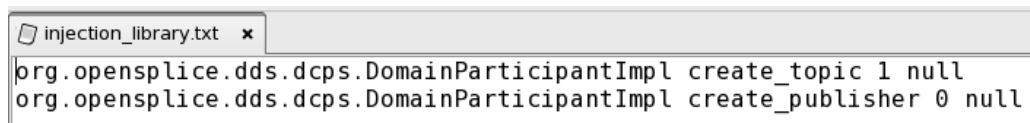
In addition to that we have some other elements which contribute to the overall scenario.

Injection library.

Here we put all the data about WHERE and WHEN submit the faults.

- Method list. The list of methods we want to harden. The package name must be also provided so that methods with the same name are discriminated by the package name they belong to.
- Trigger list. Since the same method can be called many times during the workload execution, we need a way to distinguish among these calls. The way we can do that is through the trigger value. So if we want to submit the fault when the method *foo* is called for the first time, we have to bind a trigger with value 0 to it.
- Expected return value. This is useful during the analysis phase and we can get it from the reference manual. If we know that the method returns a special value in case of exception (for instance the NULL value if it normally returns an object), it's easier to

state if the submitted fault has produced the foreseen effect or not.

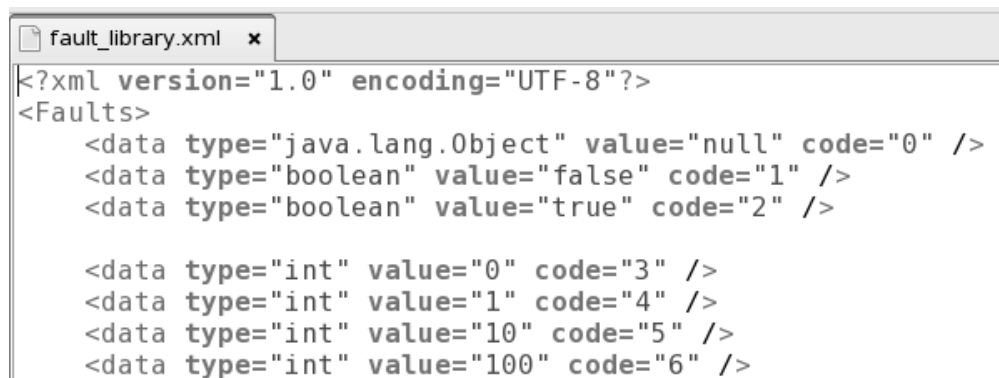


```
org.opensplice.dds.dcps.DomainParticipantImpl create_topic 1 null
org.opensplice.dds.dcps.DomainParticipantImpl create_publisher 0 null
```

Figure 2.6 - Injection library

Fault library.

The XML file where the faults are stored. For each type there's a different set of faults to use.



```
<?xml version="1.0" encoding="UTF-8"?>
<Faults>
  <data type="java.lang.Object" value="null" code="0" />
  <data type="boolean" value="false" code="1" />
  <data type="boolean" value="true" code="2" />

  <data type="int" value="0" code="3" />
  <data type="int" value="1" code="4" />
  <data type="int" value="10" code="5" />
  <data type="int" value="100" code="6" />
</Faults>
```

Figure 2.7 - Fault library

OLAP Database.

To assess better analysis we also considered to integrate the tool with an external database. It's design will be presented more accurately in a dedicated paragraph.

Process handler.

The tool provides some utility classes through which the different processes are managed. Transmitter and receiver have been modeled as two concurrent threads so that their execution could take place in a parallel way, as foreseen in one of the scenarios.

Alterator classes.

Each parameter type has its specific alterator class, so that different kinds of modifications can take into account. Since the fault parameter substitution only work with primitive input values, at the moment only boolean, byte, char, short, int, long, float, double and java.lang.String input parameter types are considered. The reference type, such us class types, array types are simply substituted by null value (not-valid reference).

Terminal (only in the distributed version of the tool).

The distributed scenario consists of Transmitter and Receiver running on two different nodes. To synchronize them the Terminal service uses a sync signal, so that transmitter and receiver can start at the same time both the FINJ experiment run and the golden run.

2.4.3 OLAP Database

The large amount of data produced in dependability evaluation experiments must be analyzed and furthermore the results from similar experiments across different usage scenarios must be compared.

To achieve that here we adopted an OLAP (On-Line Analytical Processing) analysis over a star scheme where in the center we find one large fact table surrounded by several dimensional tables related to the fact table by foreign keys.

The definition of a data warehouse star schema is a three steps process:

- Identification of the facts that characterize the problem under analysis.
- Identification of the dimensions that may influence the facts.
- Definition of the granularity of the data stored in the star schema.

Basically the fact table (Experiment table) contains the basic numerical facts obtained from the readouts collected in the experiments and the dimensions include all the different perspectives that may be used to analyze those numerical facts.

- InjectionTarget table. Here we collect all the data about the target method of our injection test.
- Fault table. It contains the attributes related to the fault we're going to submit.
- Middleware table. The middleware main features are stored in this table.
- JFIT table. Here we put some information about the tool itself.
- Scenario and SUT tables include respectively the description about the usage scenario we want to consider and some generic information about the hardware on which the testbed relies.

Each row of the fact table is linked to one row in each dimension, using the usual

normal referential mechanisms of relational databases [13].

Once the star schema is defined, the corresponding tables are created and the data warehouse is ready to be loaded with the data collected in the experiment.

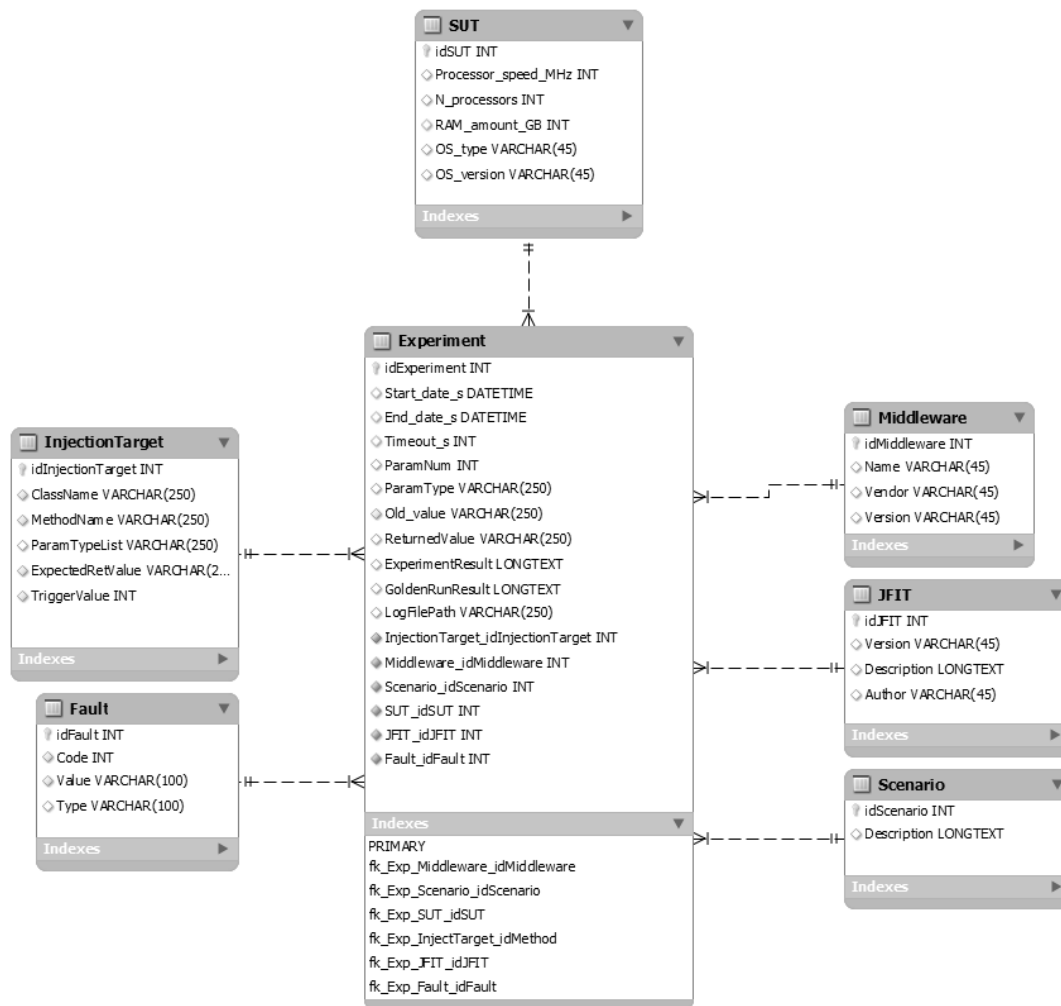


Figure 2.8 - The OLAP Database

2.4.4 Dynamic analysis: the experiment run in detail

The automation of the fault injection process is dramatically important as a robustness assessment campaign consists of a lot of runs. This is because the robustness theory says that only one fault can be submitted each time. The reason is that an eventual failure cause may be identified much easier this way, as its cause is isolated.

Thus, the fault injection (FINJ) campaign foresees several runs and during each of them we submit only one fault. This must be repeated for each fault of the set detected for each input parameter of any method, as presented in the following picture:

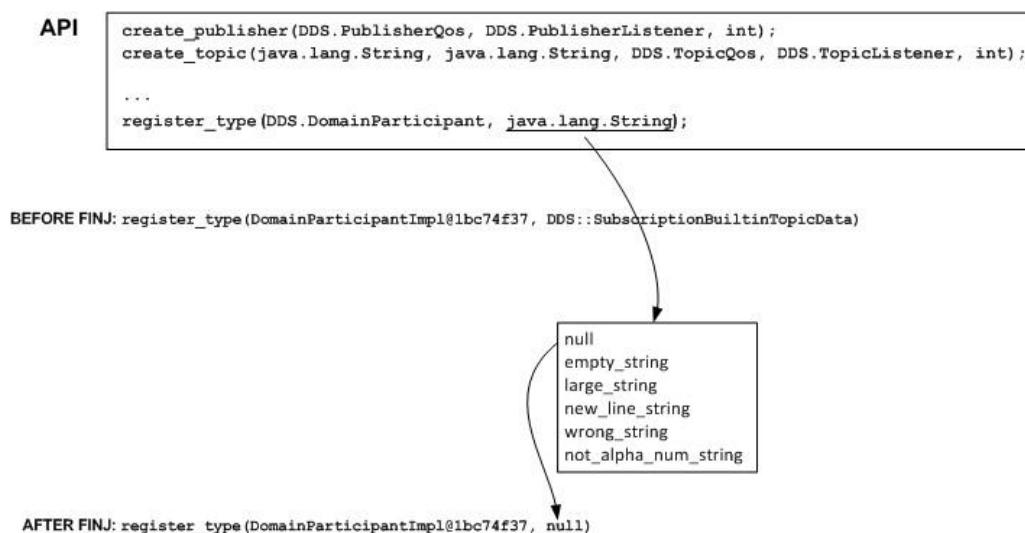


Figure 2.9 - FINJ experiment

This process must be now automated so that the following actions are repeated in sequence.

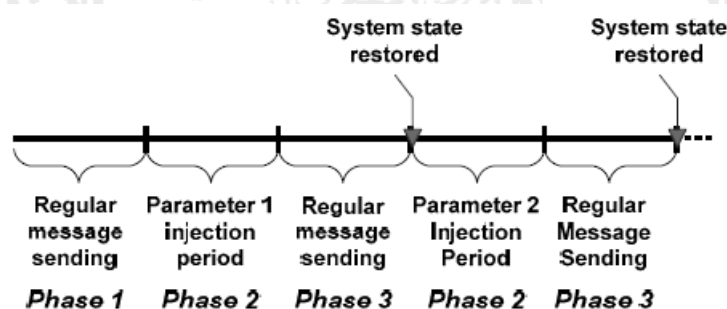


Figure 2.10 - FINJ campaign

As the picture shows the campaign consists of experiments and regular runs (golden runs) that are alternately executed. After any regular run could take place and before the very next fault injection occurs, the system is restored by shutting down the OpenSplice daemon and starting it again.

Each fault injection experiment is in turn a sequence of several steps.

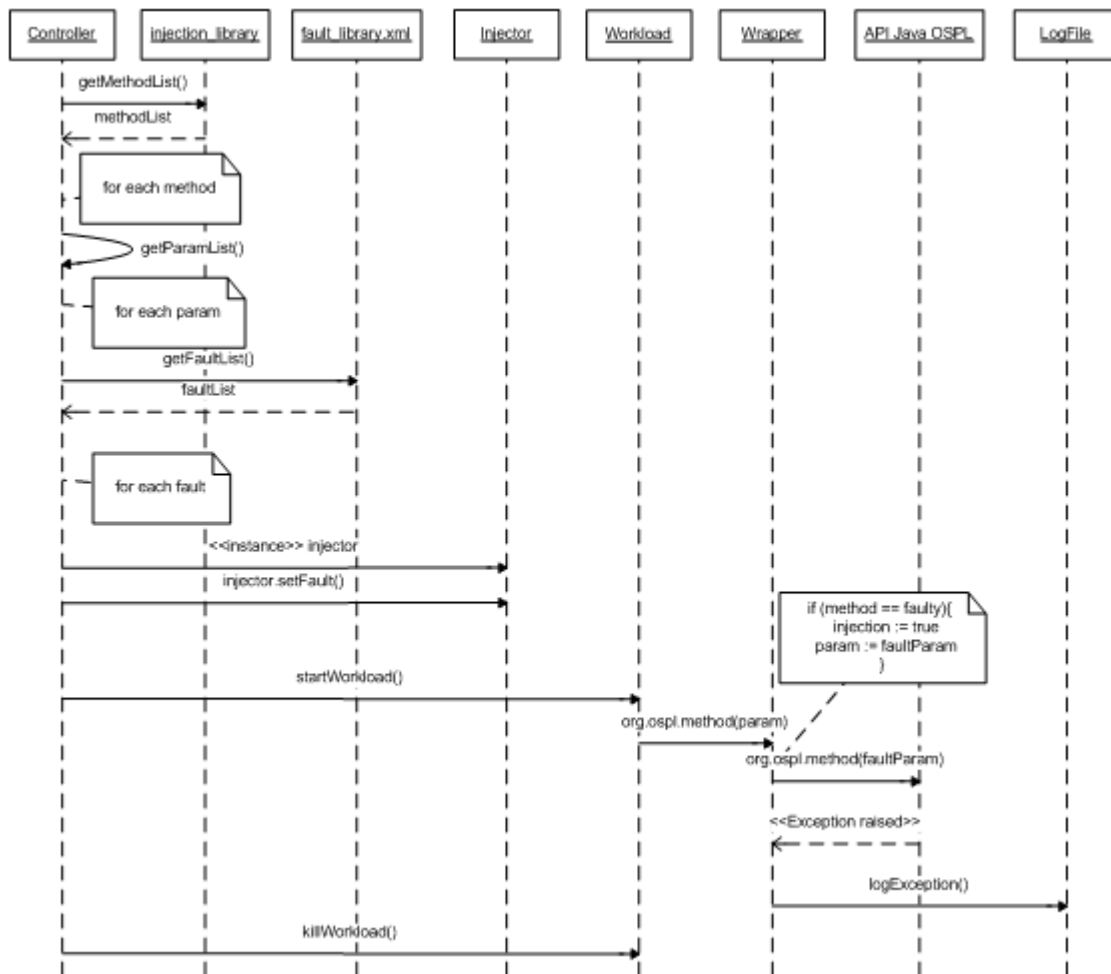


Figure 2.11 - FINJ process

First the method we want to test must be selected from the method list. Then we load the current value from the fault library related to both the method and the type the parameter belong to. Now the workload can start, so that the system can reach the necessary state to submit the fault (method call). Once the method has been intercepted we can substitute the targeted parameter value with the faulty one.

Finally, the result of the experiment must be recorded in the log-file and then the main data

are collected in the database.

2.5 Results example

As an example, we present here the content of a log-file that comes out from a sample experiment.

First we see a summary of the experiment with the main information related to it:

- the name of the workload: the application Touchstone in this case, but we may use a different one too.
- the data related to the experiment, that is, the kind of the fault that we're going to submit (fault.value), the parameter number we're going to inject (they start with the index 0), the type of the parameter we're going to alter (fault.type) and the method occurrence (trigger.value) we want to target.

```
**|** campaign started with "touchstone.Main" workload

*** finj experiment n. 10
** SUMMARY **
** fault injection on "org.opensplice.dds.dcps.DomainParticipantImpl.create_topic"
**
** fault.value: wrong_string
** fault.paramNum: 1
** fault.type: String
** trigger.value: 0

** process "ospl_start" starting
** output from "ospl_start" (if any)
Starting up domain "OpenSpliceV4.3" . Ready
** error from "ospl_start" (if any)

** process "ospl_start" terminated
** FINJ experiment starting..
```

Figure 2.12 - Experiment log-file, part 1

The experiment goes on and the submitted value makes the Opensplice APIs raise an exception while returning a NULL value. The stack trace is also logged for further analysis. After that the transmitter terminates its execution and also the fault injection run ends.

```
** method "create_topic" return value: null
** EXPERIMENT TERMINATING
run returned
shutdown hook called
** error from "transmitter" (if any)
Exiting with exception:
ddshelp.DDSError:
  operation: DomainParticipant::create_topic
  description: FAILED
  identifier: errorReportTopic
  at ddshelp.DDSError.check(DDSError.java:53)
  at ddshelp.TopicDescriptionMgr.check(TopicDescriptionMgr.java:60)
  at ddshelp.TopicMgr.create(TopicMgr.java:62)
  at touchstone.Processor.run(Processor.java:238)
  at touchstone.Main.main(Main.java:56)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at javassist.Loader.run(Loader.java:290)
  at my.wrapper.Activator.run(Activator.java:32)
  at my.wrapper.Activator.main(Activator.java:93)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at javassist.Loader.run(Loader.java:290)
  at javassist.Loader.run(Loader.java:277)
  at javassist.tools.reflect.Loader.main(Loader.java:125)
Exception in thread "Thread-0" java.lang.NullPointerException
  at touchstone.Main$1.run(Main.java:47)
** process "transmitter" terminated
** experiment executed
```

Figure 2.14 - Experiment log-file, part 3

Once the experiment returned and the output also have been logged, the golden run can start. The golden run is an execution of the application as we 'd see it in normal conditions, that is, without activating fault injection process. We use it to verify the occurrence of other failures that may not come out during the fault injection experiment phase, also known as "hindering faults".

```
** Golden run starting..

** process "transmitter" starting
** output from "transmitter" (if any)
touchstone.Main.main()
Processor.Processor()
Processor.run()
Dispatcher::run()
wait() returned 1 conditions
condition: org.opensplice.dds.dcps.ReadConditionImpl@b34bed0
Processor::read_transmitter_qos()
wait() returned 1 conditions
condition: org.opensplice.dds.dcps.ReadConditionImpl@6f7a29a1
Processor::read_transmitter_def()
Transmitter::Transmitter(50)
Transmitter::create(50, 10)
Transmitter::writer_thread(50, 10)
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) wrote message with 40 byte payload
Transmitter::writer_thread(50, 10) exiting
shutdown hook called
Processor.shutdown()
Dispatcher::shutdown()
** error from "transmitter" (if any)
** touchstone run terminating..
** process "transmitter" terminated
** golden run executed
```

Figure 2.15 - Experiment log-file, part 4

2.6 Implementation details

Some further details are provided here in relation to both the tool design and implementation. As the fault injection tool is designed to “wrap” the public methods of the middleware we decided to place it right between the workload application and the jar file where the APIs are located.

The following picture clarifies this statement.

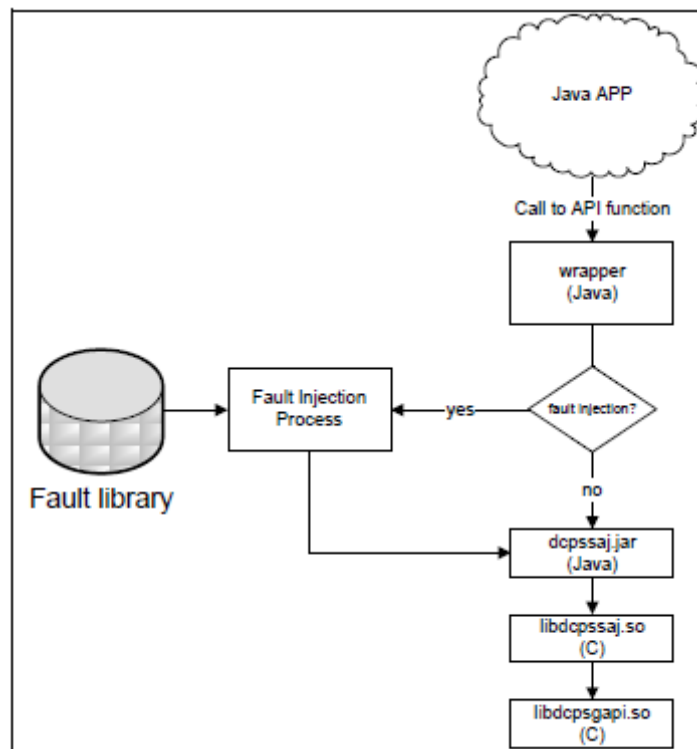


Figure 2.16 - Tool placement

What we state as “Java APP” is our JAVA-based application which makes use of the middleware APIs. The jar file, namely dcpssaj.jar, is where we expect to find the “Opensplice” APIs. When we’re going to activate the fault injection process, the fault injection tool first collects the necessary data from the fault library (and also other files), then substitutes the parameter value, and finally performs the usual call to the intercepted method.

So the call is the same as the one we would see in case of a normal execution except the fact

that here we observe the parameter value alteration.

2.6.1 Javassist library

While the standard reflection API provides the ability to only introspect a program to get information from it, Javassist [14] library overcomes this limitation by enabling program behavior alteration through load-time structural reflection and behavioral reflection.

Structural reflection is the ability to allow a program to alter the definition of data structures such as classes and methods. Since the runtime systems contain internal data representing the definition of data structures such as classes, this mechanism allows a program to directly read and change those internal data.

Behavioral reflection still provides the ability to alter the behavior of operations in a program but not provides the ability to alter data structures used in the program. To accomplish the behavioral modification of programs, Javassist enables an interception mechanism which we'll present more ahead in detail.

2.6.2 The interception mechanism

Javassist allows alteration of the behavior of specific operations such as method calls, field accesses and object creation. The programmers can select some of those operations and alter their behavior. The compilers or the runtime systems of those extensions insert hooks in programs so that the execution of the selected operations is intercepted. If these operations are intercepted, the runtime system calls a method on an object (namely a metaobject) associated with the operations or the target objects. The execution of the intercepted operation is implemented by that method. The programmers can define their own version of metaobject for implementing new behavior of the intercepted operations.

Let a metaobject be an instance of MyMetaobject, which is a subclass of Metaobject:

```
public class MyMetaobject extends Metaobject {  
    public Object trapMethodcall(String methodName, Object[]  
    args) {...}  
}
```

If method calls on an instance of class C:

```
public class C {  
    public int m(int x) { return x + f; }  
    public int f;  
}
```

are intercepted by the metaobject, then the user class loader alters the definition of the class C into the following:

```
public class C implements Metalevel {  
    public int m(int x) {  
        /* notify a metaobject */  
    }  
    public int f;  
    private Metaobject _metaobject = new MyMetaobject(this);  
    public Metaobject _getMetaobject() {  
        return _metaobject;  
    }  
    public int orig_m(int x) { return x + f; }  
}
```

For intercepting method calls, the user class loader first makes a copy of every method in class C, for example, it adds `orig_m()` as a copy of `m()`. Then it replaces the body of every method in C with a copy of the body of the method `trap()` in Exemplar (see next example).

```
class Exemplar {  
    private Metaobject _metaobject;  
    public Object trap(Object[] args, String methodName) {  
        return _metaobject.trapMethodcall(methodName, args);  
    }  
}
```

The gap between the signatures of `m()` and `trap()` is filled by hidden methods activated by Javassist library in a fully transparent manner (through Javassist class loading facilities, see next figure). The substituted method body notifies a metaobject of interception. The first parameter *args* is a list of actual parameters and the second one name is the name of the copy of the original method such as "orig_m". These two parameters are used for the metaobject to invoke the original method through the Java reflection API.

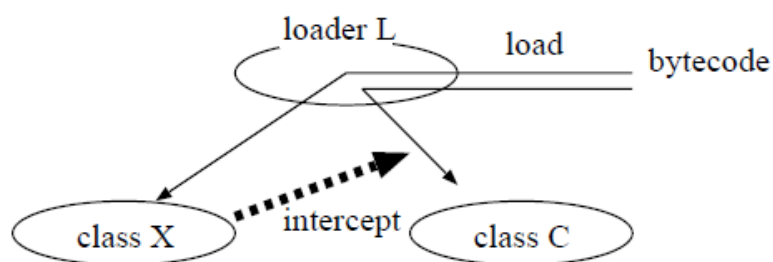


Figure 2.17 - The interception mechanism

The class loader provided by Javassist allows a loaded program to control the class loading by that class loader. If a program is loaded by Javassist's class loader L and it includes a class C, then it can intercept the loading of C by L to self-reflectively modify the bytecode of C, in the way we presented above.

2.6.3 Implementation issues

We will summarize here some lesson learned during the implementation phase of the tool. They can come in handy if further modifications or addition will be evaluated.

Workload integration.

One of the major issues to face in the developing phase was the integration between the fault injection tool and the applications that in turn were considered to solicit the middleware API (workload).

At the beginning of this thesis work a different workload had been taken into account: the SWIMBOX tool [15]. SWIMBOX is part of a technology designed to facilitate the wide sharing of ATM (Air Traffic Management) system information, such as airport operational status, flight and surveillance data. It acts as a middleware itself by making easier to different legacy systems to cooperate and it takes advantage of DDS middleware facilities to spread the information among them. As SWIMBOX makes use of DDS OpenSplice APIs to implement the Publish/Subscribe paradigm, it was a suitable use case for our tests, but some evaluations dissuaded us from considering it as our main workload application.

First we had to face the complexity of SWIMBOX itself, as it is deployed in the JBoss Application Server. This fact represents an issue when the method calls interception mechanism has to be performed.

If a program is running on a web application server such as JBoss and Tomcat, the Javassist's ClassPool object (the one in charge of finding class files from given class paths [9]) may not be able to find user classes since such a web application server uses multiple class loaders as well as the system class loader. Furthermore, the SWIMBOX startup phase takes about one minute only to reach the necessary state (the method call we use to submit the fault) to perform the fault injection and this is not acceptable when hundreds of tests must be leaded. Other reasons, such as the API coverage (as discussed more in detail in the next chapter), convinced us that another workload application had to be considered to assess the robustness campaign.

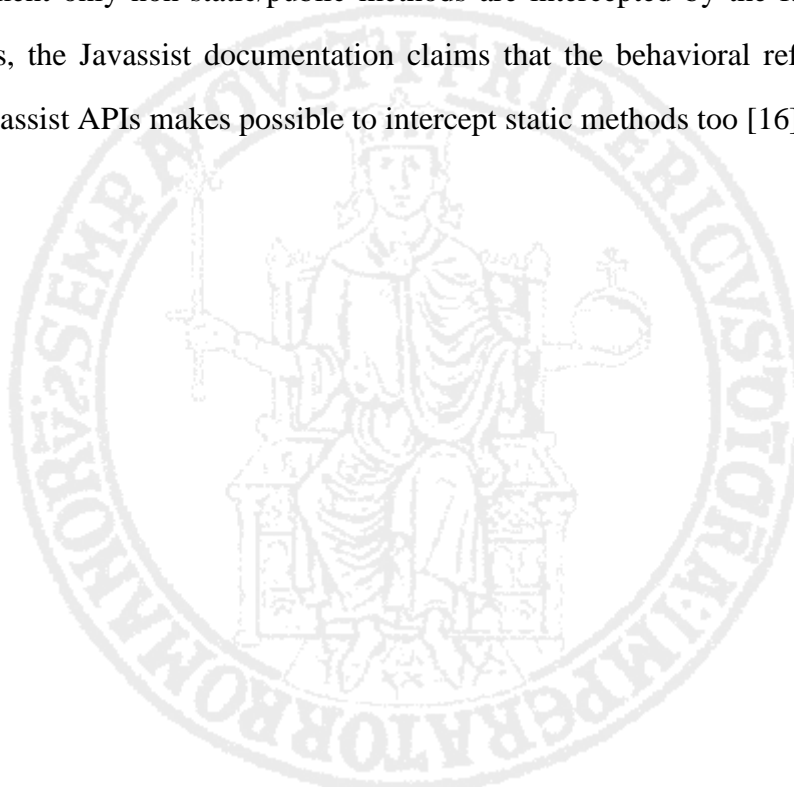
2.7 Limitations

Primitive parameters.

The tool only considers primitive values passed as input parameters as vehicle to fault submission. This issue obviously limits the scope of action to the methods with at least one primitive input parameter. This limitation can be overcome using the reflection APIs provided by JAVA. In case of structured input parameters (such as the instances of a class) the fault injection can take place by inspecting the constructors tree until reaching the elements which do not call any further constructors. Once these *leaves* will have been recognized, they can be substituted by the values provided in the fault library according to the representativeness of the test case. Finally the object must be *reconstructed* so that it can be used to submit the fault.

Static methods.

At the moment only non-static/public methods are intercepted by the fault injection tool. Despite this, the Javassist documentation claims that the behavioral reflection performed through Javassist APIs makes possible to intercept static methods too [16].



Chapter 3

Middleware for Data Distribution

3.1 Publish - Subscribe paradigm

The publish/subscribe interaction scheme is getting increasing attention and is claimed to provide a loosely coupled form of interaction, that can be decomposed among three dimensions: time, space and synchronization [17, 18]. It also offers a natural support to multi-point communication patterns.

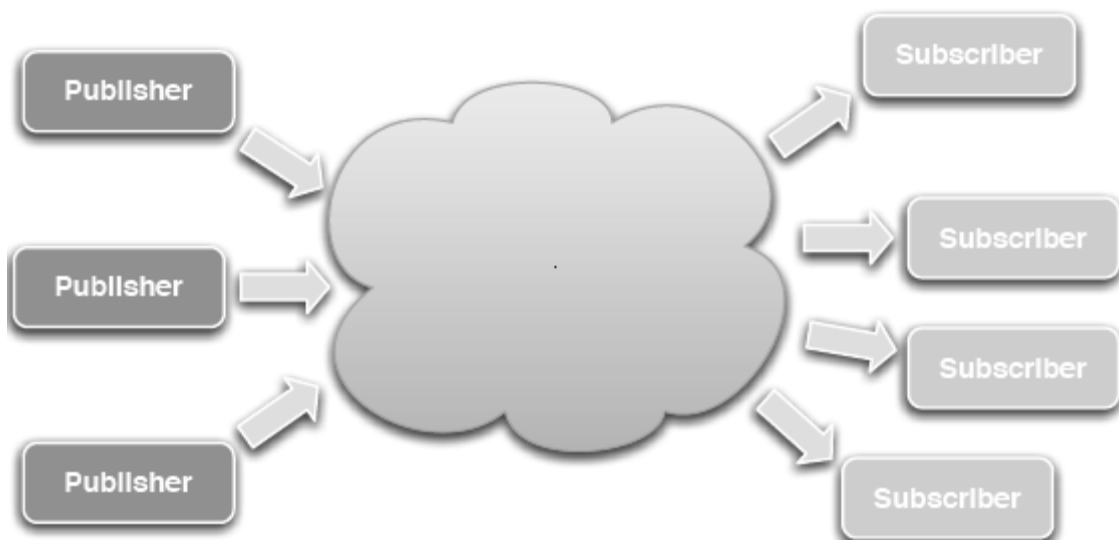


Figure 3.1 - Publish/Subscribe paradigm

A Publish/Subscribe System is composed of several entities:

Publishers, the active components of the system: produce messages to be disseminated;

Subscribers, the passive elements of the system: receive the messages of interest. A subscriber indicates the information of interest via a subscription mechanism;

Event Service, the glue between publishers and subscribers: the overall system interaction schema relies on it to provide storage and management for subscriptions and efficient delivery of messages.

A significant amount of publish/subscribe systems have been developed and implemented, both by industry and by academia. It is possible to group these different implementations according to:

Adopted Architecture.

Centralized, i.e. a server placed between publishers and subscribers;

Distributed, i.e. communication primitives implementing a store and forward mechanisms both on the producer and on the consumer sides;

Federated, i.e. a distributed network of servers.

Subscription Model.

Topic-based: participants can publish and subscribe to individual subjects;

Content-based: events are classified according to a matching function on their content;

Type-based: events belongs to a specific type, encapsulating attributes as well as methods.

3.2 DDS technology (a brief description)

The OMG Data-Distribution Service (DDS) is an emerging specification for Publish/Subscribe Data Distribution Systems. OMG DDS differs from typical CORBA-based OMG specifications in many ways, including its form and contents.

OMG DDS is built around Model Driven Architecture (MDA): it defines a Platform Independent Model (PIM) of its core constructs. This PIM is defined using UML, fully describing the API and semantics of the DDS entities, their role, parameters and the various QoS policies. From this PIM, middleware developers are free to derive any Platform

Specific Model (PSM) that matches their design goals. The mapping has to preserve the overall semantics, but can rely on any intermediate technologies before producing the final implementation.

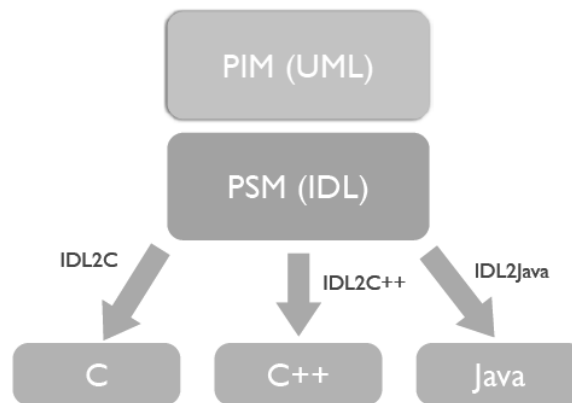


Figure 3.2 - DDS language mapping

OMG DDS purpose is to provide a common application-level interface for interoperable Publish/Subscribe middleware and targets real-time communications, balancing predictable behaviour and implementation efficiency/performance. It relies on the use of different QoS to tailor the service to the application requirements.

The DDS specification describes two levels of interfaces.

A lower Data-Centric Publish-Subscribe (DCPS) level that is targeted towards the efficient delivery of the proper information to the proper recipients. It allows:

- Publishing applications to identify the data objects they intend to publish, and then provide values for these objects.
- Subscribing applications to identify which data objects they are interested in, and then access their data values.
- Applications to define topics, to attach type information to the topics, to create publisher and subscriber entities.
- Attach QoS policies to all these entities and, in summary, to make all these entities operate.



Figure 3.3 - DCPS layer

An optional overlying Data-Local Reconstruction Layer (DLRL) level, which allows for a simpler integration into the application layer. It allows:

- Incapsulate DCPS entities into classes of objects with their methods, data fields and relations;
- Manipulate those objects using the native language constructs, in order to activate the attached DCPS entities in the most appropriate way;
- Have those objects managed in a cache of objects, ensuring that all the references that point to a given object actually point to the same language cell.



Figure 3.4 - DLRL layer

As far as possible, DLRL is designed to allow the application developer to use the underlying DCPS features. However, this may conflict with the main purpose of this layer, which is ease of use and seamless integration into the application. Therefore, some DCPS features may only be used through DCPS and are not accessible from the DLRL.

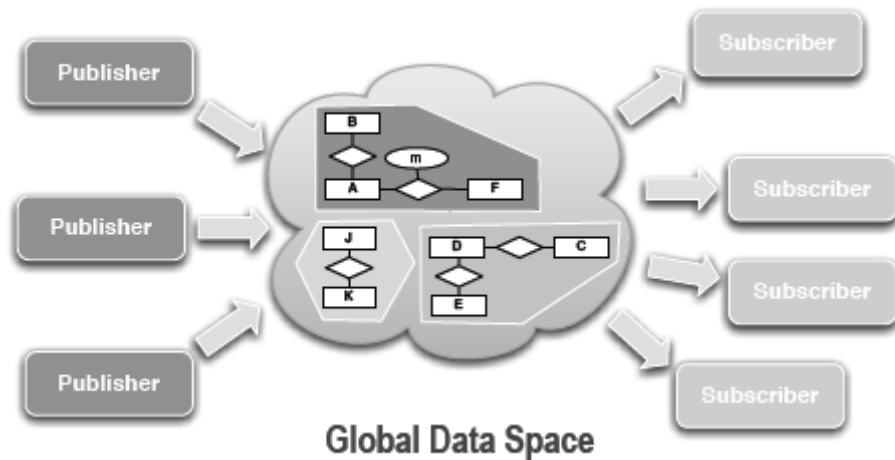


Figure 3.5 - The global data space

OMG DDS realizes a data-centric communication, as shown in the above figure, where the information exchanges refer to values of an imaginary global data object. Given that new values typically override prior values, both application and middleware need to identify the actual instance of the Global data object the value applies to. In other words, a publisher writing the value of a data-object must have the means to indicate uniquely the data object it is writing. This way, the middleware can distinguish the instance being written and decide, for example to keep only the most current value.

The basic communication model of DDS is one of unidirectional data exchange, where the applications that publish data “push” the relevant data updates to the local caches of co-located subscribers to the data. The communication patterns typically include many-to-many style configurations.

3.2.1 OMG DDS conceptual model

Information flows into the system with the aid of the following entities [17]: Publisher and DataWriter on the sending side, Subscriber and DataReader on the receiving side, as shown in the figure below:

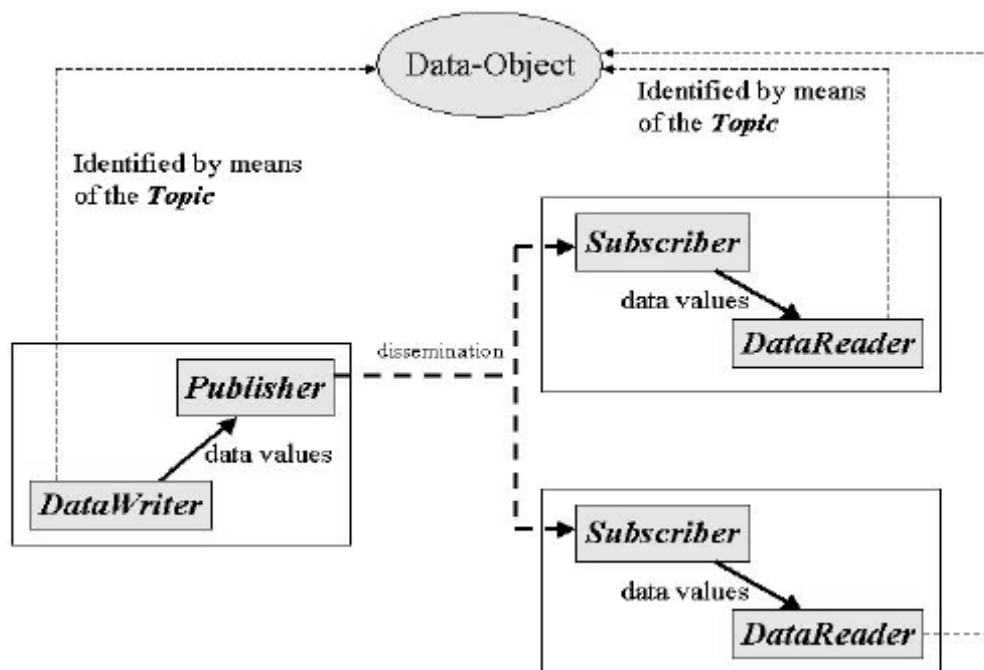


Figure 3.6 - OMG DDS conceptual model

- A Publisher is an object responsible for data distribution. It may publish data of different data types.
- A DataWriter acts as a typed (or dedicated to one application data-type) access to a publisher. The DataWriter is the object the application must use to communicate to a publisher the existence and value of data-objects of a given type. When data-object values have been communicated to the publisher through the appropriate data-writer, it is the publisher's responsibility to perform the distribution (the publisher will do this according to its own QoS, or the QoS attached to the corresponding data-writer).

A publication is defined by the association of a data-writer to a publisher. This association

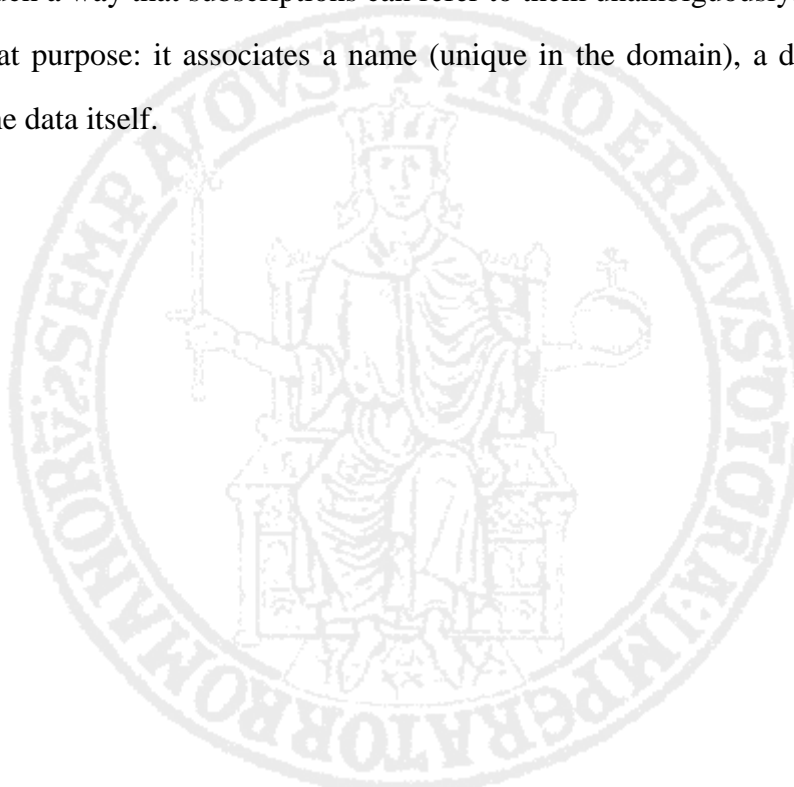
expresses the intent of the application to publish the data described by the data-writer in the context provided by the publisher.

- A Subscriber is an object responsible for receiving published data and making it available (according to the Subscriber's QoS) to the receiving application. It may receive and dispatch data of different specified types.
- A DataReader behaves as a typed access attached to the subscriber to the received data.

A subscription is defined by the association of a data-reader with a subscriber. This association expresses the intent of the application to subscribe to the data described by the data-reader in the context provided by the subscriber.

This information flow is regulated by QoS contracts implicitly established between the DataWriters and the DataReaders. The DataWriter specifies its QoS contract at the time it declares its intent to publish data and the DataReader does it at the time it declares its intent to subscribe to data.

Topic objects conceptually fit between publications and subscriptions. Publications must be known in such a way that subscriptions can refer to them unambiguously. A Topic is meant to fulfill that purpose: it associates a name (unique in the domain), a data-type, and QoS related to the data itself.



3.2.2 OMG DDS profiles

The OMG-DDS service specifies a coherent set of profiles, that target real-time information-availability for domains ranging from small-scale embedded control systems up to large-scale enterprise information management systems. Each DDS-profile adds distinct capabilities that define the service-levels offered by DDS in order to realize this 'right data at the right time at the right place' paradigm:

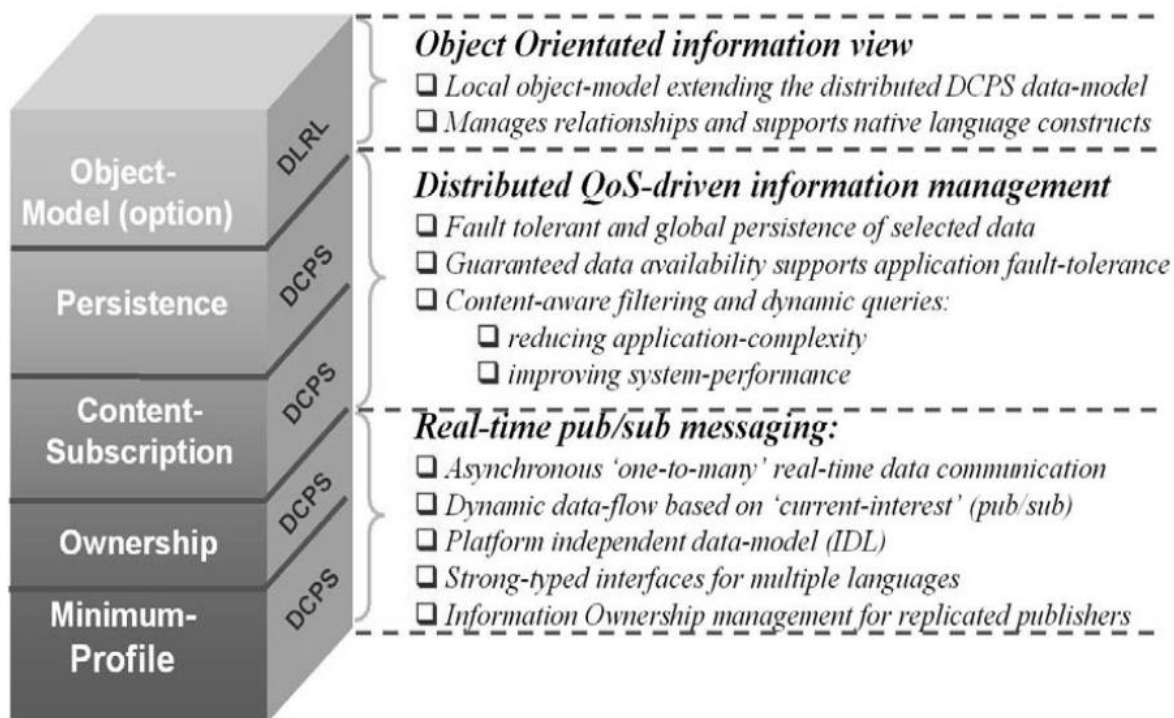


Figure 3.7 - DDS profiles

Minimum Profile. This 'basic' profile utilizes the well known publish/subscribe paradigm to implement highly efficient information dissemination between multiple publishers and subscribers that share interest in so called 'topics'. Topics are the basic data structures expressed in the OMG's IDL language (allowing for automatic generation of typed 'Readers' and 'Writers' of those 'topics' for any mix of languages desired). This profile also includes the QoS framework that allows the middleware to 'match' requested and offered Quality of Service parameters (the minimum profile offering basic QoS attributes such as 'reliability',

'ordering' or 'urgency').

Ownership Profile. This 'replication' profile offers support for replicated publishers of the same information by allowing a 'strength' to be expressed by each publisher so that only the 'highest strength' information will be made available to interested parties.

Content Subscription Profile. This 'content awareness' profile offers powerful features to express fine grained interest in specific information content (content filters). This profile also allows applications to specify projection views and aggregation of data as well as dynamic queries for subscribed 'topics' by utilizing a subset of the well known SQL language whilst preserving the real-time requirements for the information access.

Persistence Profile. This 'durability' profile offers transparent and fault tolerant availability of 'non volatile' data that may either represent persistent 'settings' (to be stored on mass media throughout the distributed system) or 'state' preserved in a fault tolerant manner outside the scope of transient publishers (allowing late joining applications and dynamic reallocation).

DLRL Profile. This 'object model' (Data Local Reconstruction Layer) extends the previous four data centric 'DCPS' profiles with an object-oriented view on a set of related topics thus providing typical OO features such as navigation, inheritance and use of value types.

3.3 PrismTech OpenSplice

OpenSplice core modules cover the "Minimum" and "Ownership" profiles that provide the basic pub-sub messaging functions. The minimum profile is meant to address real time messaging requirements, where performance and low footprint are essential. The ownership profile provides basic support for replicated publishers where 'ownership' of published data is governed by 'strength' indicating the quality of published information. OpenSplice "content subscription" and "persistence" profiles provide the additional information management features, important for assuring high information-availability (fault-tolerant persistence of non-volatile information) as well as 'contentaware' features (filters and queries). The "Data Local Reconstruction Layer" profile provides an object-oriented view

on a set of related topics allowing for typical OO-features such as value-types with inheritance, object-navigation and others. This 'DLRL' profile of the DDS-spec, and therefore also the OpenSplice implementation are still a 'work-in-progress'.

Its internal architecture utilizes shared-memory to 'interconnect' the applications that reside within one computing node, but also 'hosts' with a configurable and extensible set of services. These services provide 'pluggable' functionality such as networking (providing QoS-driven real-time networking based on multiple reliable multicast 'channels'), durability (providing fault-tolerant storage for both real time 'state' data as well as persistent 'settings'), and remote control & monitoring 'soap-service'.

Utilizing a shared-memory architecture, data is physically present only once on any machine, while each subscriber has his own private 'view' on this data; this allows a subscriber's data cache to be perceived as an individual 'database' that can content-filtered, queried, etc. (using the content-subscription profile). This shared-memory architecture results in low footprint, good scalability and optimal performance when considering a high number of publishers/subscribers; in fact, in such situation, when multiple entities are located on a single machine, it is possible to limit to the maximum extent the need to perform multiple copies of the data to be transferred among publishers/subscribers.

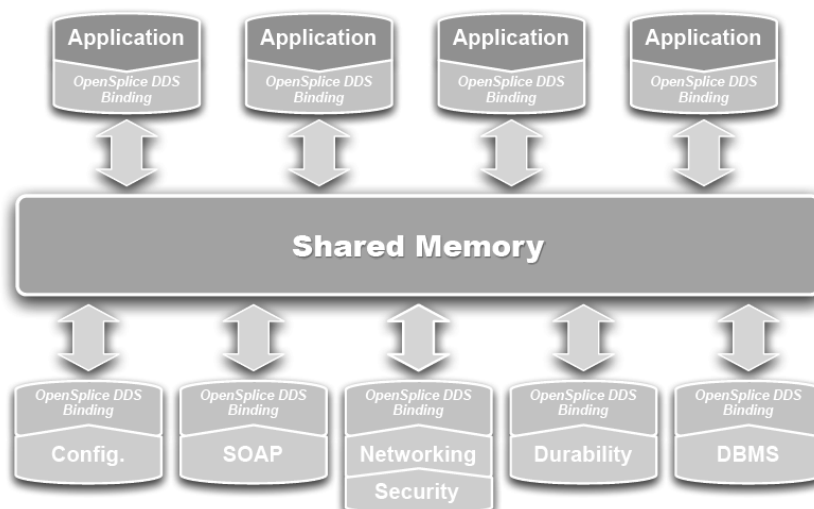


Figure 3.8 - PrismTech OpenSplice

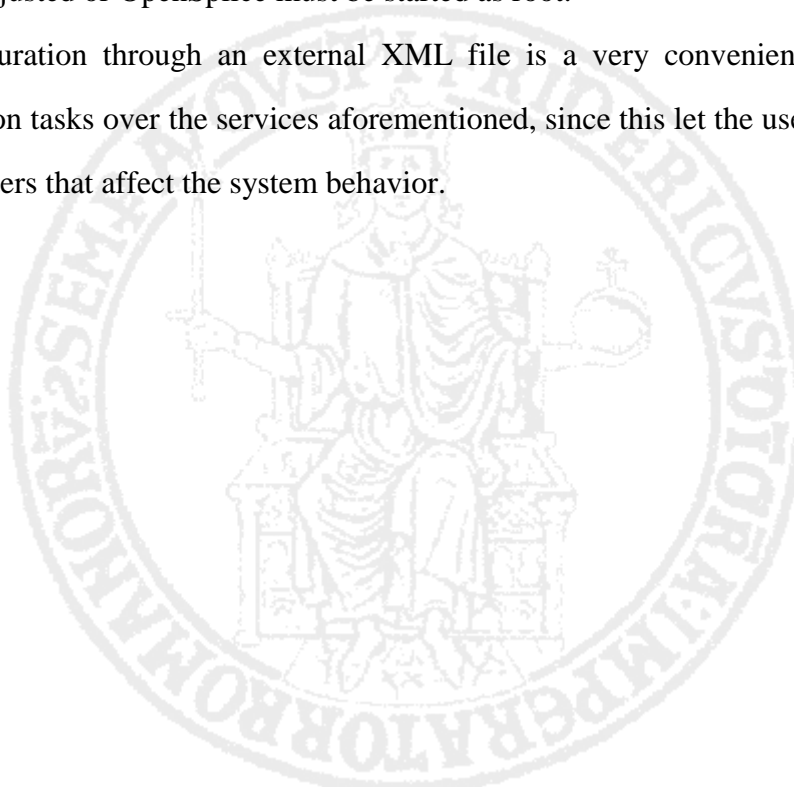
As it can be seen by the picture, each application will link the OpenSplice libraries in order to utilize the DDS features and to (transparently) communicate with the pluggable services via the common shared memory.

OpenSplice is configured using an XML configuration file; the configuration file defines and configures the following services:

- the default service, also called the *domain service* - it is responsible for starting and monitoring all other services;
- *durability service* - responsible for storing non-volatile data and keeping it consistent within the domain (optional);
- *networking service* - realizes user-configured communication between the nodes in a domain;
- *tuner service* - provides a SOAP interface for the OpenSplice Tuner to connect to the node remotely from any other reachable node.

The default Database Size that is mapped on a shared-memory segment is 10 Megabyte; the maximum user-creatable shared-memory segment is limited on certain machines, so it must either be adjusted or OpenSplice must be started as root.

The configuration through an external XML file is a very convenient way to perform configuration tasks over the services aforementioned, since this let the user to easily modify the parameters that affect the system behavior.



Chapter 4

The experimental campaign

In this chapter we focus on two main aspects. In the first part we present the middleware's API study performed at the early stage of this thesis work, then in the last part, we will introduce and analyze the results collected from the robustness campaigns.

4.1 Preparation phase

Here we summarize some aspects that we took into account when assessing the experimental campaign.

First we defined our hardware and software platform so that it might be representative of a typical usage scenario. From the beginning, this thesis work has been carried on relying on the same software tools (operating system, developing and analysis tools, etc..) that might have been found in the industrial environment where the middleware is deployed.

Every robustness study must state what it's considered as a normal behavior. To do this we executed our workload application several times. It allowed us to understand what we should have observed during a regular execution of the workload application from a behavioral point of view.

Finally, after the results had been collected, we have produced some statistical reports to highlight the main data according to well known past studies about robustness.

4.2 Analysis of DDS API specification

We outline here a summary description of the middleware's APIs as presented in [19].

OpenSplice APIs are designed following the DDS specifications [17] and rely on five main modules. Each module consists of several classes as defined at PIM level (see previous chapter).

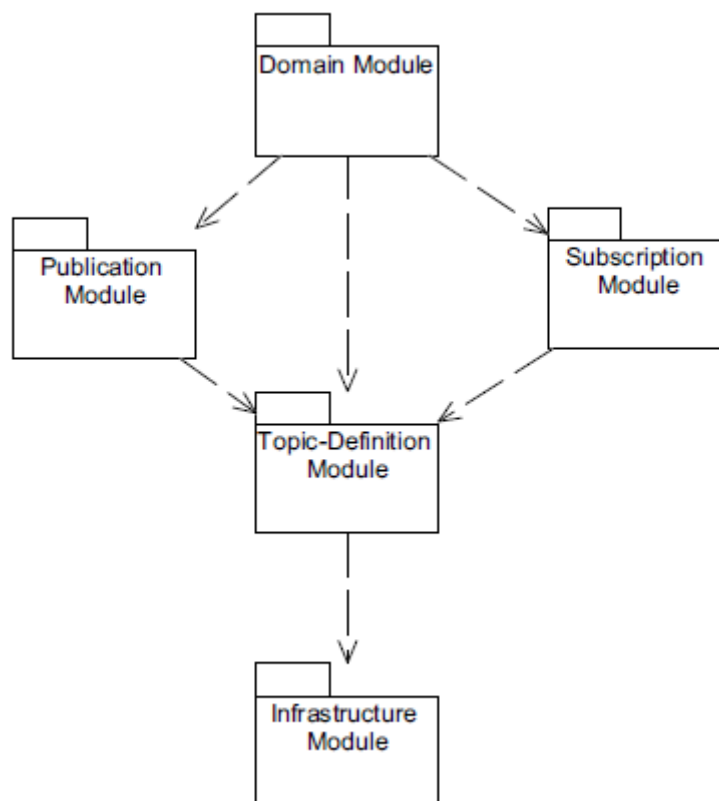


Figure 4.1 - API modules

The modules have the following function in the Data Distribution Service:

Infrastructure Module. Defines the abstract classes and interfaces, which are refined by the other modules. It also provides the support for the interaction between the application and the Data Distribution Service (event-based and state-based);

Domain Module. Contains the DomainParticipant class, which is the entry point of the

application and DomainParticipantListener interface;

Topic-Definition Module. Contains the Topic, ContentFilteredTopic and MultiTopic classes. It also contains the TopicListener interface and all support to define Topic objects and assign QosPolicy settings to them;

Publication Module. Contains the Publisher and DataWriter classes. It also contains the PublisherListener and DataWriterListener interfaces;

Subscription Module. Contains the Subscriber, DataReader, ReadCondition and QueryCondition classes. It also contains the SubscriberListener and DataReaderListener interfaces.

As stated in the previous chapter the DDS standard is composed by a PIM specified in UML and a PSM level specified in IDL. Language specific APIs (C, C++, Java, etc...) are derived via IDL-based mappings [20], this mainly because IDL's biggest strength is the language independence, which at the same time can turn into its biggest weakness as the developers try to integrate APIs that exploit all the capabilities of programming languages. IDL language is also involved in topics definition, allowing for automatic generation of typed Readers and Writers for any mix of language desired.



The classes that are provided in OpenSplice's Java APIs are included in `dcpsaj.jar` library file. The most of methods implemented there don't do nothing else but refer to C methods placed in `libdcpsaj.so` library file, through JNI (Java Native Interface) binding facilities.

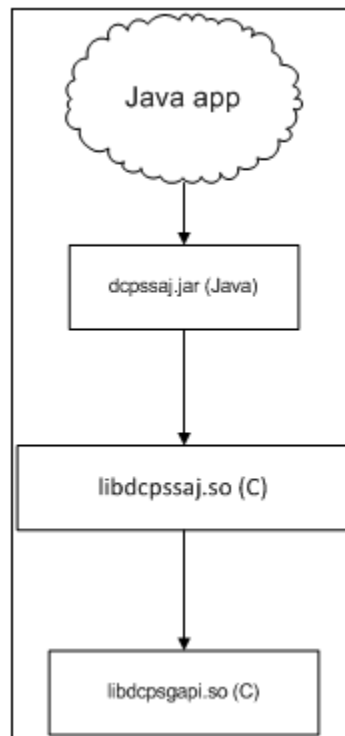


Figure 4.2 - OpenSplice API hierarchy

This choice makes it possible domain logic layer abstraction and simpler developing through high level language programming (Java, C++, C#) as well as better performance of low level functions as implemented in C language.

We had to consider that when assessing the API coverage study which led us to the choice of the most suitable workload application for our robustness tests.

4.2.1 API coverage

The coverage analysis has been performed with four different Java application. Obviously we only considered public methods with at least one input parameters.

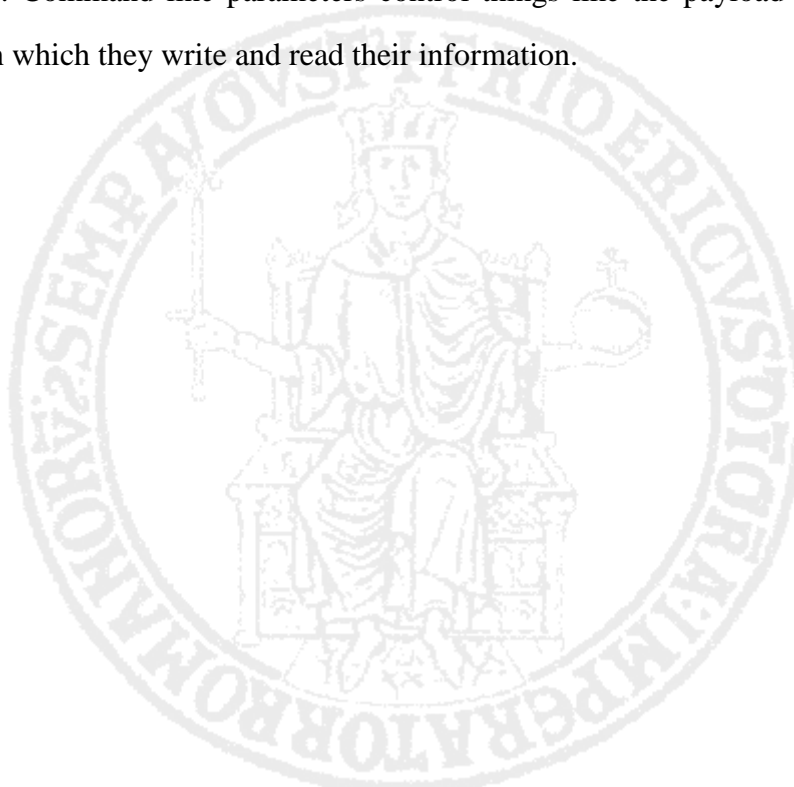
We first give an overall idea about the different workloads which has been used, then we summarize this data by giving the percentage of how many methods are used by each workload application against the total amount of API methods.

Simple Java Publish Subscribe application.

A simple application which relies on middleware's APIs to send one *Hello World* message from one Publisher to one Subscriber.

Ping Pong (tutorial) application.

The *PingPong* example is a small benchmarking program that bounces a topic back and forth between *ping* and a *pong* applications. It measures and displays the round-trip times of these topics, giving a first impression on some performance characteristics of the middleware. Command line parameters control things like the payload for the topics and partitions in which they write and read their information.



SWIMBOX FNM Demo.

SWIMBOX is part of a technology designed to facilitate the wide sharing of ATM (Air Traffic Management) system information, such as airport operational status, flight and surveillance data. It acts as a middleware itself by making easier to different legacy systems to cooperate and it takes advantage of DDS middleware facilities to spread the information among them [15].

Here two legacy applications running on the same computer are interacting through SWIMBOX mediation.

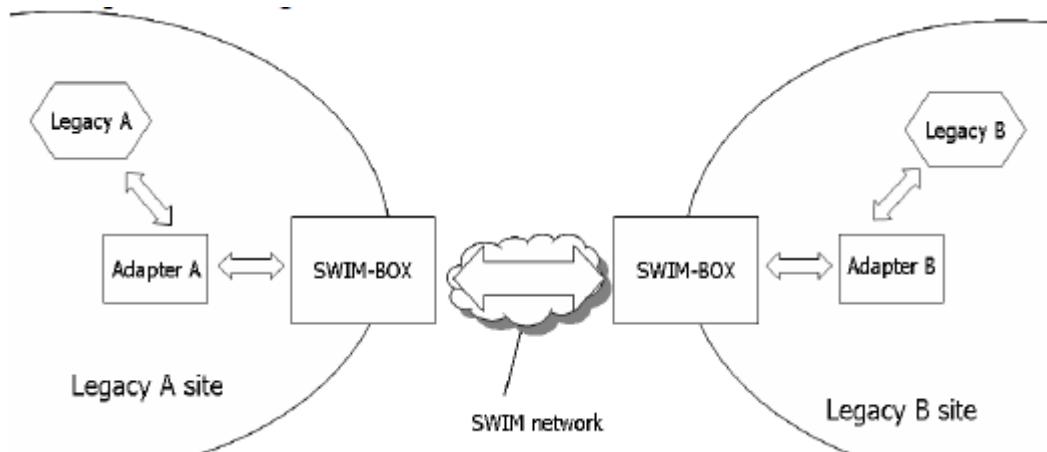


Figure 4.3 - SWIMBOX Scenario

PrismTech Touchstone Benchmark, Tutorial Scenario.

DDS Touchstone [21] is a framework for benchmarking Data Distribution Service (DDS) implementations. The scenario we used here foresees that one Transmitter application sends 10 messages to one Receiver, they both are placed in the same computer and modeled as concurrent threads.

<u>Workload application</u>	<u>API methods occurrences</u>	<u>%</u>
Simple Java Publish/Subscribe application	27	12
Ping Pong (tutorial) application	32	14
SWIMBOX FNM Demo	37	16
PrismTech Touchstone benchmark	51	22
Total (API public methods with at least one input parameter)	227	100

As we can see from the above table, the scenario that seems to solicit more than others the middleware's APIs, as it uses 51 public methods with at least one input parameter, is the PrismTech Touchstone benchmark application. It will provide the set of methods that we will take into account for our experiments.

4.2.2 Data collection issues

Many tools have been evaluated when we had to face the API coverage study.

Profiling tools like those ones we will enumerate here allow on the one hand black-box tracing, without affecting the source code at all, but on the other hand they work fine only in some cases.

Some of them, such as valgrind and callgrind, only worked fine with C/C++ applications and failed with our Java-based workload applications[22]. Later we also assessed this task through Java memory profiling tools, such as Jmemprof [23].

These tools only gave as an estimation about the number of the methods which our workload applications called, this because the tool takes snapshots of the JVM memory, when the application is running. So, when garbage collector removes the methods reference no longer used, some of them are lost and we're not capable of catching them anymore.

The most accurate way to lead this study was by instrumenting the Java source code of the middleware so that any time a public method of OpenSplice APIs was called its signature were logged in a text file. While we lost the black-box approach here, we could be sure this way that all the methods had been collected if used by the workload scenario.

Other ways might have been followed, less intrusive, such as the usage of the Java Reflection APIs or by the interception mechanism performed by the Javassist APIs.

4.3 Real world scenarios

Here we introduce some applications which take advantage of DDS facilities to operate well in two different areas. The first of them, SWIMBOX, was available in SESM and contributed to some preliminary studies, besides the fact that we used it to perform API coverage analysis of middleware's functionalities. SWIMBOX gives an excellent idea of which may be the use of DDS capabilities in the world of industry. The second one is a benchmark application. As we stated before the open-source benchmark utility provided by PrismTech was more suitable for our robustness assessment campaigns, as it exploits in a wider way the DDS middleware's APIs.

4.3.1 SWIM-BOX

SWIM is the technology program designed to facilitate the wide sharing of ATM system information, such as airport operational status, weather information, flight and surveillance data. It is intended to use Commercial Off-The-Shelf (COTS) hardware and software to support a Service Oriented Architecture (SOA) aiming to facilitate systems dynamic composition and to increase common situational awareness.

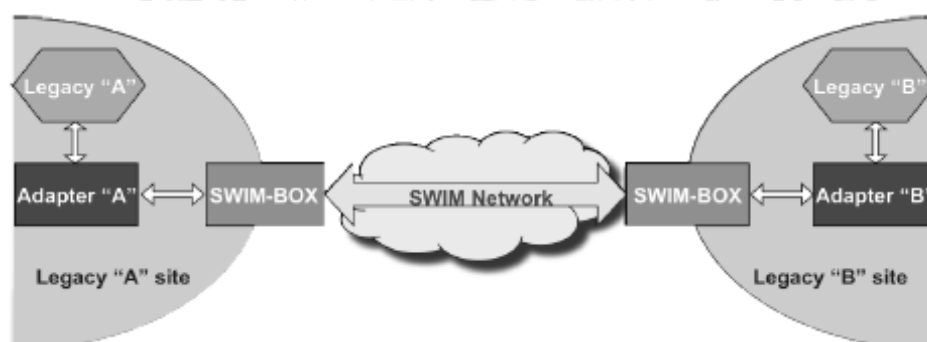


Figure 4.4 - SWIM Network

The overall system is a network of SWIM nodes, physically deployed at stakeholders premises and named legacy, which are the actual users of the SWIM common infrastructure. These nodes are allowed to access the SWIM bus through a SWIMBOX component (see figure above). Only SWIMBOX instances are allowed to directly exchange data and to invoke services over this network, acting as mediators with respect to the legacy components. Since the existing legacy systems were not aware of the SWIM service semantics, it has been necessary to implement a further software level, named ADAPTER , to let them access the SWIM services and interoperate via the SWIMBOX [15].

The SWIM-BOX Core has no knowledge of the data representation it manages, as it provides services for data delivery and QoS management (in different technologies for each pattern - publish/subscriber or request/reply). It is also loosely impacted by changes in the data representation and by changes in the services exposed via the SWIM Data Domains (act as much as possible as a transport layer).

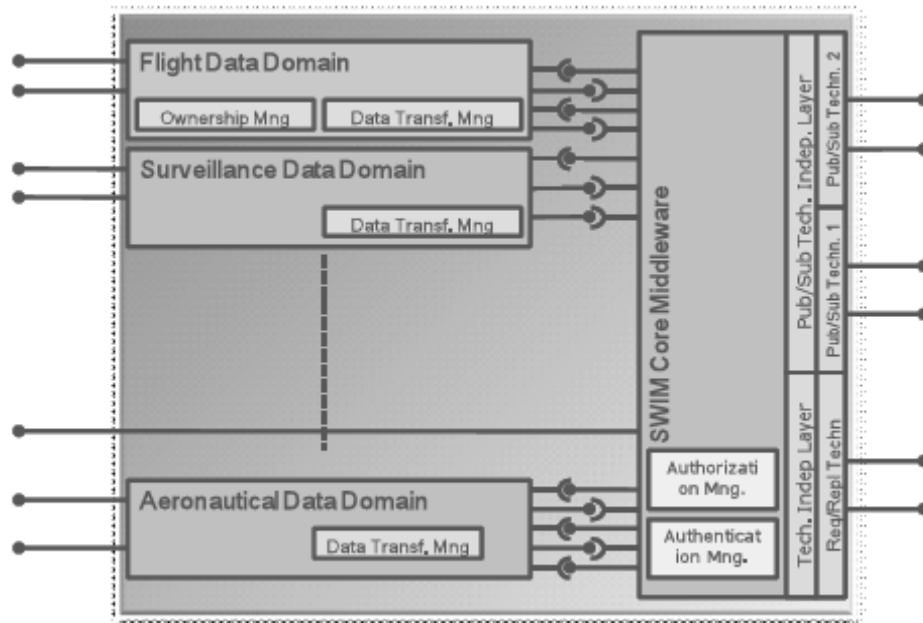


Figure 4.5 - SWIM-BOX architecture

Publish/Subscribe pattern, thanks to its intrinsic characteristics (e.g. decoupling, scalability, flexibility, etc..) is the ideal choice to support a “many to many” interaction and

SWIMBOX adopted DDS and JMS as technologies to implement this pattern.

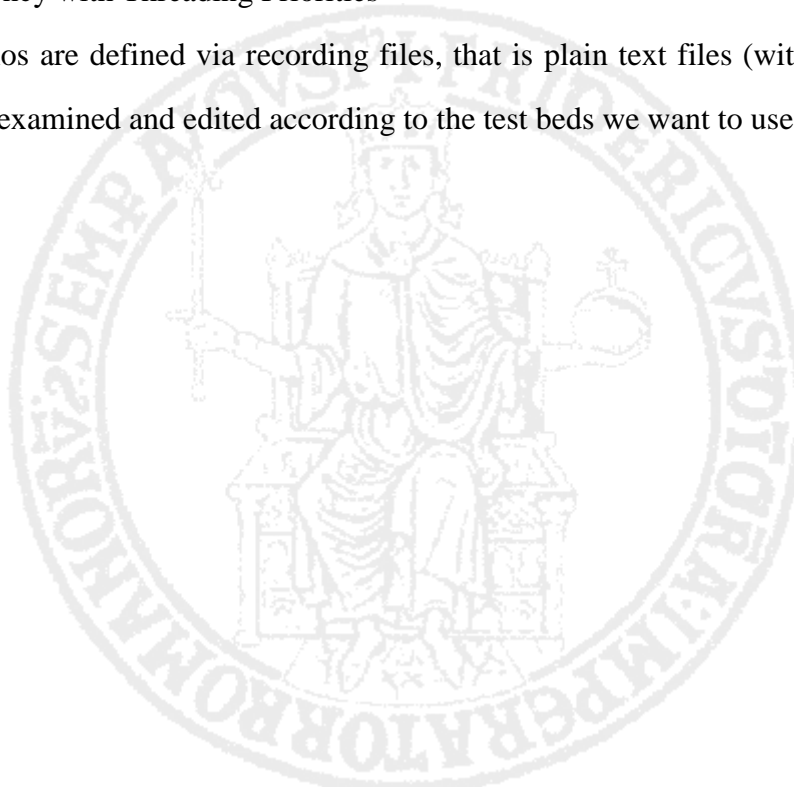
4.3.2 PrismTech Touchstone benchmark

As a benchmarking suite, PrismTech's Touchstone [21] main purpose is to create sensible DDS benchmarking tests that are representative of their application domains. It also helps users to get familiar with DDS without having to write a lot of code, accelerate the acceptance of DDS as an alternative to other middleware, test performance and compliance of their products as well as test interoperability between DDS implementations.

Touchstone is released fully open-source and it is available for many different Operating Systems (Linux, Windows, Solaris ...) and application languages (C, C++ and Java). What matters most in this study is that Touchstone came already providing some "ready to use" examples scenarios, that can produce benchmarking results for the following areas:

- Throughput
- Latency
- Latency with Threading Priorities

The scenarios are defined via recording files, that is plain text files (with .dat extensions), that can be examined and edited according to the test beds we want to use.



4.4 Test-bed description

Now we give more detail about the test-bed from both a hardware and software point of view.

In order to lead the tests in the same environment where the middleware under examination typically works, we adopted the software configuration used by SESM computers in Giugliano facilities, in terms of Operating System, Java Virtual Machine version and so on.

Test-bed software configuration:

- Operating System: 64 bit Red-Hat Linux Operating System, version 5
- Middleware: PrismTech OpenSplice Community Edition version 4.3
- Java Virtual Machine: SUN's JVM version 1.5
- Javassist: library version 3.4

Concerning the workload application we elected the throughput scenario as the one which best fitted our case. As it is available as open source software we also had the opportunity to tailor somehow the code of the application so that it better matched our needs. This tiny modification was necessary as the benchmark application runs in an infinite loop and we wanted the test run to last for a definite time interval.

Taking advantage of the API functionalities provided by the middleware, we set a timer, through which each test could terminate after 20 seconds. This modification met our requirements about tests time.

The throughput scenario foresees one Transmitter process that sends ten messages to another process, namely the Receiver. Concerning where Transmitter and Receiver are running the scenario can be defined as local or distributed.

OpenSplice comes with a configuration file defined in the XML format. There some configuration parameters are given so that behavior of several aspects of the middleware itself can be tailored to the user's needs.

Some attribute there are more meaningful than others to our study. For instance attribute

reliable in NetworkService node in the configuration file allows to set the channel behavior. For our purposes we decided to leave it default, this way the communication channel will perform a “best-effort” delivery of the messages.

Local scenario

This scenario consists of both Transmitter and Receiver processes running on the same machine. They are modeled as concurrent threads so that their execution is pseudo-parallel, as required by the scenario.

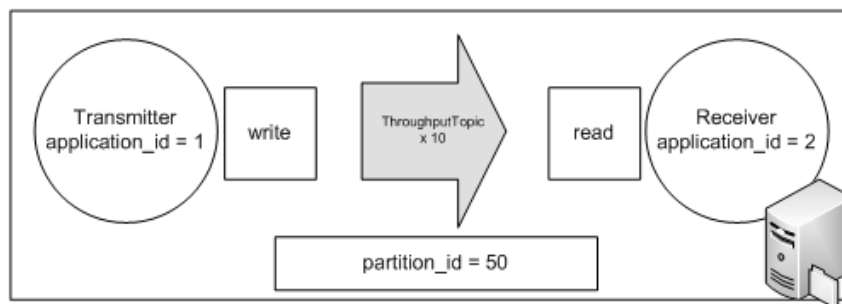


Figure 4.6 - Local scenario

Distributed scenario

In this case the Transmitter and the Receiver are placed in two different computers. Another process, running on Transmitter's machine, synchronizes both Transmitter and Receiver by sending them a *start* message any time a fault injection experiment or a golden run is scheduled to begin.

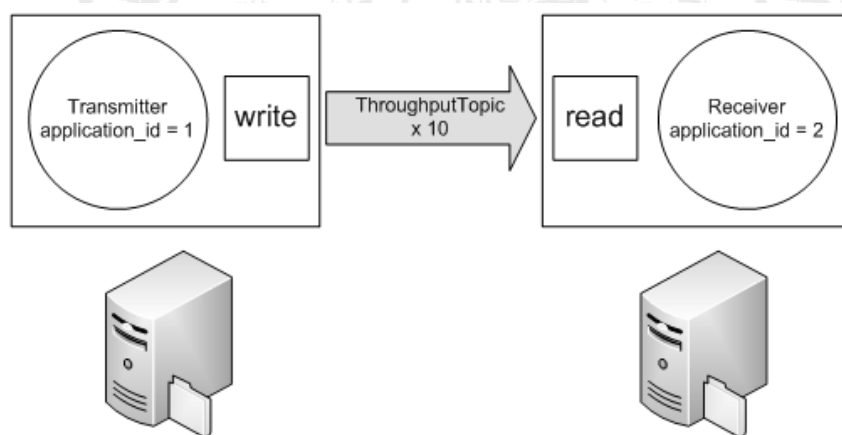


Figure 4.7 - Distributed scenario

4.5 Tests execution

The fault injection tool has been used to test 14 public methods from both DDS.* and org.opensplice.dds.dcps.* packages that define the middleware's API. From the initial set of method we considered only those ones were public, non-static and with at least one primitive input parameter. Although this set of methods might appear limited against the total amount of public methods provided by the public interface (227 methods), it includes the core set of operations we always need to establish a publish-subscribe interaction.

We list them all here, giving a brief description of their usage, the returned parameter and the value given in case of exception.

- **org.opensplice.dds.dcps.DomainParticipantImpl.create_topic**

This operation creates a reference to a new or existing Topic under the given name. Return value is a reference to the new or existing Topic. In case of an error, the null reference is returned for a specific type, with the desired QosPolicy settings.

- **org.opensplice.dds.dcps.PublisherImpl.create_datawriter**

This operation creates a DataWriter with the desired DataWriterQos, for the desired Topic and attaches the optionally specified DataWriterListener to it. Return value is a reference to the newly created DataWriter. In case of an error, the null reference is returned.

- **org.opensplice.dds.dcps.DomainParticipantImpl.create_contentfilteredtopic**

This operation creates a ContentFilteredTopic for a DomainParticipant in order to allow DataReaders to subscribe to a subset of the topic content. Return value is the reference to the newly created ContentFilteredTopic. In case of an error, a null reference is returned.

- **org.opensplice.dds.dcps.DataReaderImpl.create_readcondition**

This operation creates a new ReadCondition for the DataReader. The returned ReadCondition is attached (and belongs) to the DataReader. When the operation fails, the

null reference is returned.

– **org.opensplice.dds.dcps.DomainParticipantImpl.create_publisher**

This operation creates a Publisher with the desired QosPolicy settings. Return value is a reference to the newly created Publisher. In case of an error, the null reference is returned.

– **org.opensplice.dds.dcps.SubscriberImpl.create_datareader**

This operation creates a DataReader with the desired QosPolicy settings, for the desired TopicDescription. Return value is a reference to the newly created DataReader. In case of an error, the null reference is returned.

– **DDS.SubscriptionBuiltinTopicDataTypeSupport.register_type**

– **DDS.PublicationBuiltinTopicDataTypeSupport.register_type**

– **DDS.ParticipantBuiltinTopicDataTypeSupport.register_type**

– **DDS.TopicBuiltinTopicDataTypeSupport.register_type**

Prior to creating a Topic, MultiTopic or ContentFilteredTopic, the data type must have been registered with the Data Distribution Service. This is done using the data type specific register_type operation on the <type>TypeSupport class for each data type. A class is generated for each data type used by the application, by calling the pre-processor.

When the operation returns:

- RETCODE_OK - the FooTypeSupport class is registered with the new data type name to the DomainParticipant or the FooTypeSupport class was already registered.
- RETCODE_ERROR - an internal error has occurred.
- RETCODE_BAD_PARAMETER - the domain parameter is a null reference or the parameter type_name has zero length.
- RETCODE_OUT_OF_RESOURCES - the Data Distribution Service ran out of resources to complete this operation.
- RETCODE_PRECONDITION_NOT_MET - this type_name is already registered with this DomainParticipant for a different <type>TypeSupport class.

– **org.opensplice.dds.dcps.DomainParticipantImpl.create_subscriber**

This operation creates a Subscriber with the desired QosPolicy settings. Return value is a reference to the newly created Subscriber. In case of an error, the null reference is returned.

– **DDS.DomainParticipantFactory.create_participant**

This operation creates a new DomainParticipant which will join the domain identified by domainId, with the desired DomainParticipantQos. The DomainParticipant signifies that the calling application intends to join the Domain identified by the domainId argument. If the specified QosPolicy settings are not consistent, the operation will fail; no DomainParticipant is created and the operation returns the null reference.

– **DDS.GuardCondition.set_trigger_value**

A GuardCondition object is a specific Condition which trigger_value is completely under the control of the application. This operation must be used by the application to manually wake-up a WaitSet. This operation sets the trigger_value of the GuardCondition to the parameter value.

The GuardCondition is directly created using the GuardCondition constructor. When a GuardCondition is initially created, the trigger_value is false.

When the operation returns:

- RETCODE_OK - the specified trigger_value has successfully been applied.
- RETCODE_ERROR - an internal error has occurred.

– **org.opensplice.dds.dcps.DataReaderImpl.create_querycondition**

This operation creates a new QueryCondition for the DataReader. Result value is a reference to the QueryCondition. When the operation fails, the null reference is returned.

We always set the trigger value to 0, so that, if any of the methods above is called more than once during the application workflow, the fault injection process takes place only the first time.

The last remark is about the way we detect the hang failures. Every experiment has a timeout (set to 30 seconds in the campaign presented here) so that, if the fault injection run takes more than the time limit to finish, it will be stated as a hanged run.

4.6 Results presentation

Now we give some details about the results we obtained from the tests we performed. For each one of the two scenarios there are about 200 experiments.

To collect the results we first inspected the log-files produced by the fault injection tool. At the end of this activity we filled some tables like the one we present here, as an example:

org.opensplice.dds.dcps.DomainParticipantImpl.create_topic						
	null_string	empty_string	large_string	new_line_string	wrong_string	not_alpha_num
String topic_name	3	1	4	1	1	1
	null_string	empty_string	large_string	new_line_string	wrong_string	not_alpha_num
String type_name	1	1	4	1	1	1
TopicQos qos	1					
TopicListener a_listener	2					

Figure 4.8 - Snippet of the results collection file

For each method we have a list of input parameter and, for each of them, we gave a score to every fault injection experiment. We can see from the above table that there's a variable number of experiments leaded against each input parameter, as the fault values we submit have been chosen according to the type the parameter belongs to.

1	The workload (Tx) terminates raising an exception whose value corresponds to reference behavior (Robust)				
2	The faulty value and the original ones are identical (Robust)				
3	The faulty value is submitted, no exception raised, messages transmitted (Silent)				
4	Tx hangs, Rx exits without getting any message (Restart)				
5	The faulty value is submitted (Tx), no exception raised, no messages transmitted (Silent)				

Figure 4.9 - JFIT scale

We observed 5 different events during the experiments. For example, considering the `org.opensplice.dds.dcps.DomainParticipantImpl.create_topic` operation table given before, when we submitted an *empty string* in place of the original value of the `topic_name` input parameter, the middleware raised an exception compliant with the specification. On the contrary, when we replaced the `topic_name` input parameter by a null reference, no exception has been raised.

Then we mapped our failure scale on the CRASH scale we presented in one of the previous chapters, so that the failures we detect in our test could be analyzed through this metric too.

CRASH scale	JFIT scale
Catastrophic	N/A
Restart	4
Abort	N/A
Silent	3, 5
Hindering	N/A
Robust behavior	1, 2

As we see in the table, no catastrophic, abort or hindering events have been observed in our tests.

We now summarize the overall results according to the CRASH scale. The total number of test cases and the total number of operations tested are given as well as the kind of failure highlighted by the tests campaign.

Scenario	Total function tested	Total test cases	# of robust test cases	# of silent failures (messages transmitted)	# of silent failures (NO messages transmitted)	# of hang failures
Local	14	202	86	90	1	25
Distributed	14	194	81	94	4	15

Figure 4.10 - Overall results

If the experiment ends up raising an exception compliant with the specifications, we can state that a robust behavior has been observed.

When submitting a faulty value we expect that the middleware's APIs raise an exception, but in case of silent failure no one has been thrown and the experiments ended up doing delivering all the messages (less serious event) or not delivering at all (more serious event).

Hang events are the most serious ones failures we observed in our campaign. Most of the time, this happened when a large_string fault value had been submitted, but hangs have been observed as a result of other inputs too. These events (except the case of large_string fault) are not deterministic. So if we obtain a hang event in one experiment, we cannot be sure that the very same experiment will have the same result if we perform it again.

4.7 Results analysis

The overall results demonstrate that on average the set of methods under examinations are robust against our set of faults. About 40% of the tests end up rising an exception compliant with the specification, it means that the middleware behaves as we expected.

The results distribution is quite similar among the two scenarios, this let us say that, for our test set, the process deployment barely affects the robustness assessment.

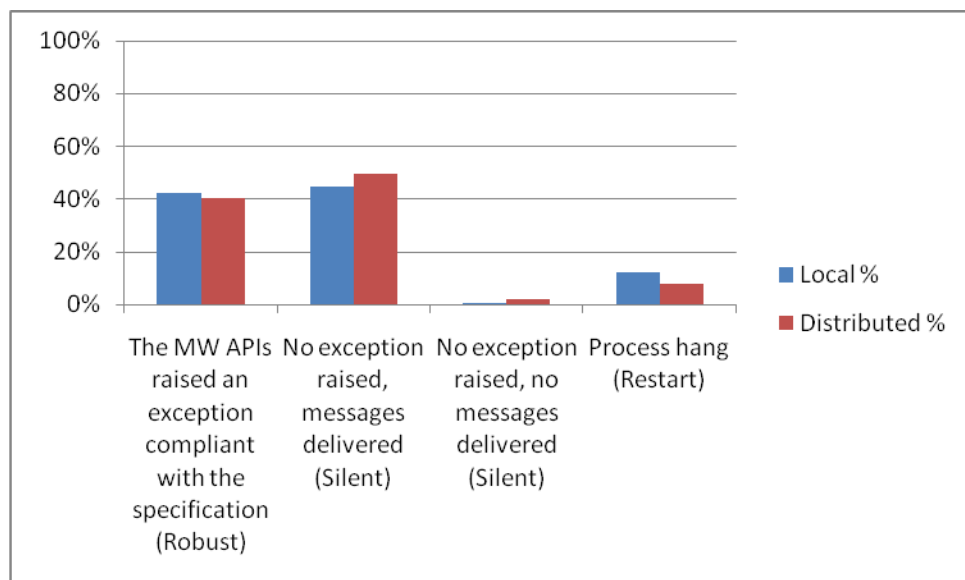


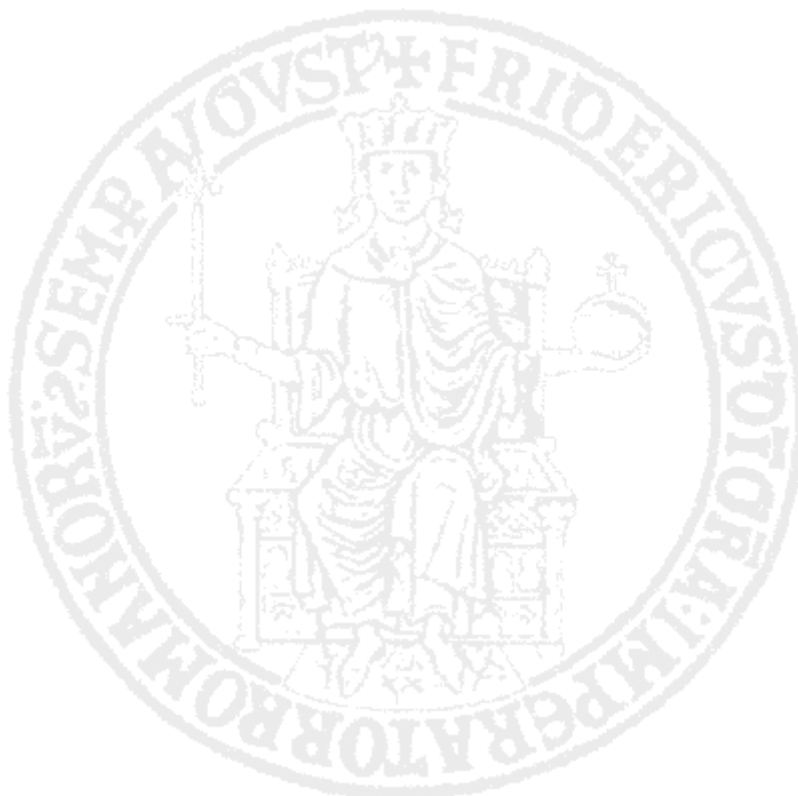
Figure 4.11 - Scenarios comparision

We observe a slightly higher percentage (less than 50%) of tests indicating an unexpected behavior: although the input values have been replaced by others invalid, no exception has been raised and the experiment terminated gracefully by shipping the messages. From a mere behavioral point of view this can be stated as a robust run, but probably the experiment didn't fail because the value we submitted has changed a propriety that our scenario didn't take into account. So as we can't state here the generality of this result we decided to consider it as a "silent" failure.

Some other experiments terminated properly but without shipping any message and this was still considered as a silent failure. The issue here is more serious than the previous case as

the fault modifies the behavior of the system and no exception is raised. In a critical context this failure is even more dangerous as the users will still think about the system as behaving properly even if it's not working properly indeed.

Hang events are more frequent (about 10%) but often related to a very special input parameter (i.e. a 64KB length string provided as a input) and then we can't say here if the hang is related to some JVM internal failure or the middleware's itself. This doesn't change the fact that the middleware's API should have performed a dimension check with this parameter.



Conclusions and future works

Our main purpose was on one hand to find a methodology that could match some requirements such as representativeness and effectiveness, on the other hand our concern was to get some experimental results from this approach. We can say that we met the both expectations. The testing results in previous chapter have shown a promising picture of this robustness assessment tool.

Many challenges have been overcome during this work, first the complexity of the middleware's API itself, then the workload integration, the interception mechanism and finally the results collection. We also must highlight that the fault injection tool implementation has started from scratch as other tools were not suitable for our case study.

The final result is a comprehensive tool which functionalities can be expanded by a deeper study of the several technologies which compose it.

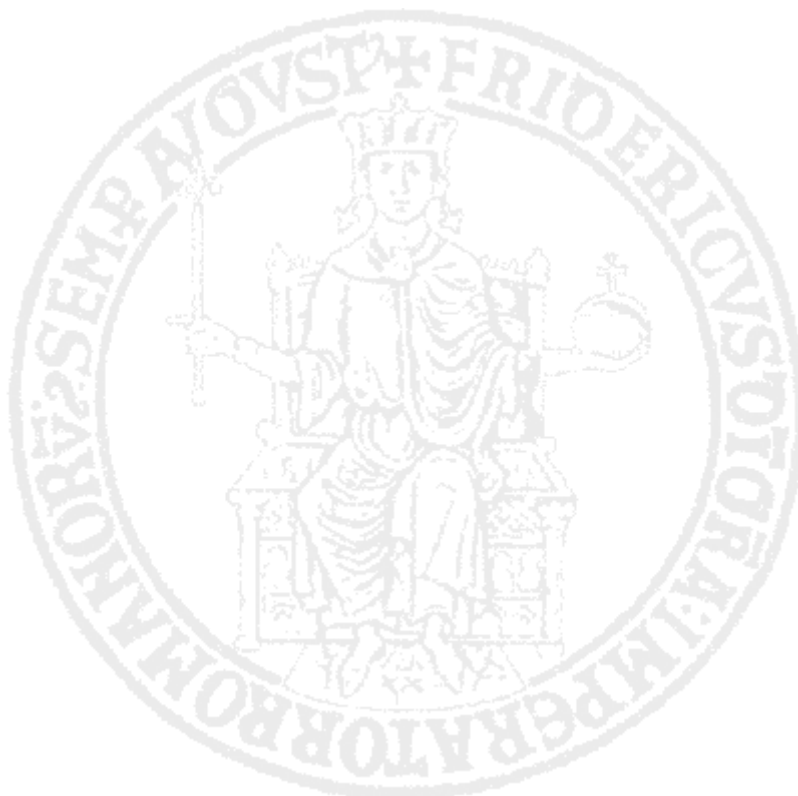
By exploiting the Javassist APIs capabilities we can enlarge the set of methods to test and an algorithm based on Java reflection could be taken into account to perform fault injection in structured input parameters.

Concerning the workload application, the most significative results got from several robustness campaigns can be used to perform dedicated experiments with more complex or industry scenarios where the system under examination is involved.

Since our methodology is general, the effort necessary to perform robustness assessment

studies with other COTS is small, provided that a different workload application is given.

Finally the OLAP database we used to collect the campaign results may be tailored to the new case studies the tool deals with.



Acknowledgments

Ringraziamenti

Eccomi giunto al momento dei ringraziamenti, epilogo di questa tesi. Faccio il bilancio di questi sette lunghi mesi di tirocinio e la conclusione è che mi scopro più maturo sotto il piano umano, oltre che professionale.

Questa esperienza per me non ha solo rappresentato l'atto finale dei miei anni di studi, ma l'ingresso in una nuova fase della mia vita, alla quale dedicare un rinnovato impegno e nuove motivazioni. Perciò il mio primo ringraziamento va al prof. Cotroneo che ha reso possibile questo lavoro e ai miei correlatori Gabriella, Lello, Antonio e Christian con cui ho collaborato affinché potessi concluderlo.

Un grazie speciale ai ragazzi del SESM, che mi hanno accolto con grande entusiasmo nella loro famiglia, tutti loro. Andare ogni giorno a Giugliano sapendo di trovarci voi era di gran consolazione nei momenti più bui.

Grazie ai miei amici, di università e non solo. Farò di tutto per restarvi accanto, qualsiasi strada dovessi prendere da qui in avanti. Sapete che potete sempre contare su di me.

Gracias, thank you e obrigado a tutte le persone meravigliose che ho conosciuto viaggiando all'estero durante questi anni. La distanza non ha mai impedito che il vostro affetto arrivasse fino a me.

Ma il grazie più grande va alla mia famiglia, il cui sostegno incondizionato è stato indispensabile durante tutto questo tempo. Non sarei mai arrivato a questo punto se non avessi avuto voi, non vi ringrazierò mai abbastanza.

Bibliography

- [1] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, 2004, pp. 11-33
- [2] G.Carrozza, "SOFTWARE FAULTS DIAGNOSIS IN COMPLEX, OTS-BASED, CRITICAL SYSTEMS", Ph.D. Thesis, defended at University of Naples FEDERICO II, December 2008
- [3] IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), IEEE Computer Soc., Dec. 10, 1990
- [4] Jiantao Pan, Philip Koopman, Daniel Siewiorek, Yennun Huang, Robert Gruber, Mimi Ling Jiang, "Robustness Testing and Hardening of CORBA ORB Implementations," Dependable Systems and Networks, International Conference on, p. 0141, The International Conference on Dependable Systems and Networks (DSN'01), 2001
- [5] http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/index.html
- [6] Micskei, Majzik, "Comparing Robustness of AIS-Based Middleware Implementations", 2007
- [7] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer, "Fault injection techniques and tools," Computer, vol. 30, Apr. 1997, pp. 75-82.
- [8] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek, "Automated robustness testing of off-the-shelf software components," Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on, 1998, p. 230-239.
- [9] K.Z. Zamli, M.D.A. Hassan, N.A.M. Isa, and S.N. Azizan, "An automated software

- fault injection tool for robustness assessment of java COTs,” Computing & Informatics, 2006. ICOCI’06. International Conference on, 2006, p. 1–6.
- [10] E. Martins, C.M.F. Rubira, and N.G.M. Leme, “Jaca: A reflective fault injection tool based on patterns,” Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, 2002, p. 483–487.
- [11] P. Koopman, K. DeVale, and J. DeVale, “INTERFACE ROBUSTNESS TESTING: EXPERIENCES AND LESSONS LEARNED FROM THE BALLISTA PROJECT,” Dependability benchmarking for computer systems, 2008, p. 201.
- [12] P. Koopman, “What’s Wrong With Fault Injection As A Benchmarking Tool?,” Workshop on Dependability Benchmarking, p. 31.
- [13] H. Madeira and others, “The OLAP and data warehousing approaches for analysis and sharing of results from dependability evaluation experiments,” 2003.
- [14] S. Chiba, “Load-time structural reflection in Java,” ECOOP 2000—Object-Oriented Programming, 2000, p. 313–336.
- [15] G. Carrozza, D. Di Crescenzo, and A. Strano, “Data distribution technologies in wide area systems: lessons learned from the SWIM-SUIT project experience,” Proceedings of the First International Workshop on Data Dissemination for Large Scale Complex Critical Infrastructures, 2010, p. 9–13.
- [16] <http://www.csg.is.titech.ac.jp/~chiba/javassist/tutorial/tutorial3.html>
- [17] OMG Available Specification, “Data Distribution Service for Real-time Systems Version 1.2”
- [18] Christian Esposito, “DDS-related material for beginners”
- [19] PrismTech, “OpenSplice DDS Version 4.x Java Reference Guide”
- [20] PrismTech, “OpenSplice DDS Version 4.1 IDL Pre-processor Guide”
- [21] <http://www.prismtech.com/opensplice/opensplice-dds-community/related-projects>
- [22] http://www.network-theory.co.uk/docs/valgrind/valgrind_128.html
- [23] <http://oss.metaparadigm.com/jmemprof/>