Comparing Robustness of AIS-based Middleware Implementations†

Zoltán Micskei[1], István Majzik[1], and Francis Tam[2]

[1] Dept. of Measurement and Information Systems,
Budapest University of Technology and Economics, Budapest, Hungary
{micskeiz, majzik}@mit.bme.hu

[2]Nokia Research Center, Nokia Corporation, Finland
francis.tam@nokia.com

Abstract. To enable the interoperability of high availability (HA) middleware systems the Service Availability Forum has released a set of open specifications. The benefit of having open specifications is the choice of implementations available from different vendors. When one chooses a product, one of the selection criteria (besides performance) is the robustness of the implementation, as the crashing or hanging of such a HA middleware causes the failure of the whole system. The challenge is to develop the appropriate technology for measuring and comparing robustness of HA middleware implementations. Based on our earlier results, we present a set of automatic testing tools and a benchmark suite constructed using these tools. We demonstrate the robustness testing approach by comparing the results of benchmarking carried out on three HA middleware implementations.

Keywords: dependability, robustness testing, HA middleware

1 Introduction

Recently availability became a key factor even in common off-the shelf computing platforms. High availability (HA) can be achieved by introducing manageable redundancy in the system. The common techniques to manage redundancy and achieve minimal system outage can be implemented independently from the application, and can be put on the market as a HA middleware. The standardization of the functionality of such middleware systems has begun as the leading IT companies joined the Service Availability Forum (SA Forum) to elaborate the Application Interface Specification (AIS) [1]. One of the benefits of an open specification is that it enables a company to choose from different vendors, thus reducing the technology risks.

With multiple middleware products developed from the same specification the demand to compare the various implementations naturally arises. The most frequently

† The funding of this work by Strategy and Technology, Nokia Networks under the project HASEK in 2006 is acknowledged.

Final version appeared in: ISAS 2007, LNCS 4526, 2007, pp 20–30. DOI: 10.1007/978-3-540-72736-1_3. The final publication is available at link.springer.com.

examined properties are performance and functionality, but especially in case of HA products the dependability is also an important property to be considered. This paper outlines an approach to compare robustness, one of the attributes of dependability of HA middleware systems.

2 Robustness testing approach

Robustness is defined as the degree to which a system operates correctly in the presence of *exceptional inputs* or *stressful environmental conditions*. Related work includes API robustness testing and dependability benchmarks. In the Ballista project [2] the robustness of several POSIX implementations were compared using type-specific testing, and several failures were found even in well-known commercial operating systems. Dependability benchmarks aim for a slightly broader goal, to assess the dependability of the complete system. In DBench [3] a conceptual framework was designed and several case studies (e.g. for OS and OLTP systems) were carried out. Based on the above results we elaborated an approach for robustness testing of high availability middleware systems [4]. Because of the complex state-based nature of HA middleware, the previous methods had to be extended.

The first step of developing the test strategy was the identification of the potential *sources* for activating robustness faults in the HA middleware. Figure 1 illustrates these sources, considering a typical computing node of a HA distributed system, as follows:

1. External errors: They affect the operation of the application, thus their effects reach the HA middleware only indirectly (through normal, erroneous or missing API calls).
2. Operator errors: In general, operator errors appear as erroneous configuration of the middleware and erroneous calls using the specific management interface.
3. API calls: The calls of the application components using the public interfaces of the HA middleware can lead to failures if they use exceptional values, e.g. NULL pointer or improperly initialized structures.
4. OS calls: The robustness of a system is also characterized by its ability to handle the exceptions or error codes returned by the OS services it uses.
5. Hardware failures: The most significant HW failures in a HA system are host and communication failures (that has to be tolerated in the normal operating mode of the HA middleware) and lack of system resources.

From the above sources the following ones were selected to be included in the first version of the dependability benchmark suite:

- The standardized middleware API calls are considered as a potential source of activating robustness faults. Because of the high number of possible exceptional value combinations and scenarios, the elements of the robustness tests suite were automatically generated by tools. The challenge in testing the API calls was that most of the AIS interface functions are state-based, i.e. a proper initialization call sequence, middleware configuration and test arrangement is required, otherwise a trivial error code is returned.

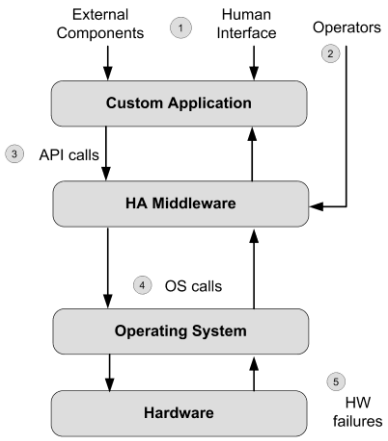

Fig. 1. HA middleware fault model

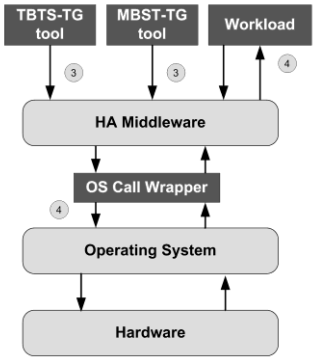

Fig. 2. Testbed tools

- The failures of the OS system calls were included for the following reason. They do not only represent the faults of the OS itself (which has lower probability for mature operating systems), but failures in other software components, in the underlying hardware and in the environment also could manifest in an error code returned by a system call. Possible examples of such conditions are writing data to a full disk, communication errors when sending a message, etc.
- Studies show that operator errors cause also a significant part of service unavailability, however, the configuration of the HA middleware and the system management interface are still under standardization by the SA Forum, thus they were not included in the current version of the benchmark suite.

3 Testbed tools and benchmark suite

Taking into consideration the potential sources of activating robustness faults, a set of tools was developed to assist the activation of these faults by generating proper test values and performing the test calls. This dependability benchmark testbed is depicted in Figure 2. In the following, we describe these tools and the benchmark suite developed for testing version B.02.01 of the AIS Availability Management Framework (AMF). Although the API of the AMF is standardized, the implementations selected for testing (two versions of openais [5] and one version of SAFE4TRY [6], see Section 4) influenced the realization of the test execution environment.

3.1 Template-based type-specific test generator

The template-based type-specific test generator (TBTS-TG) uses the following approach to generate robustness test cases that realize calls to the HA middleware API with exceptional values. Instead of defining the exceptional cases one by one for each

API function, the exceptional values are defined with regard to the parameter *types* that are used in the functions. From the description of these types, the tool generates a *test program* for each API function, and this test program calls the given function with all combinations of the specified values. Each combination is executed in a new process to separate the test cases from each other, and the result code of the call is logged after completion. The test case is considered to detect a robustness failure if the test program or the middleware implementation crashes or hangs (e.g. due to a segmentation fault or a timeout). To help diagnosing the robustness faults, the first calls contain only a single exceptional value (using valid values in the case of the remaining parameters).

The inputs and outputs of the tool are presented in Figure 3. The skeleton of the test program is prepared manually as an XSL template. The metadata of the functions and types to test are specified in XML files. The exceptional and valid values are defined as C code snippets. For simple types, e.g. numbers and enumerations, values recommended by traditional testing techniques were selected, like valid values, boundary values and values outside the domain of the given type. In the case of complex structures the following systematic method was used: for each member there are test cases that assign invalid values to the given member while the other members remain valid.

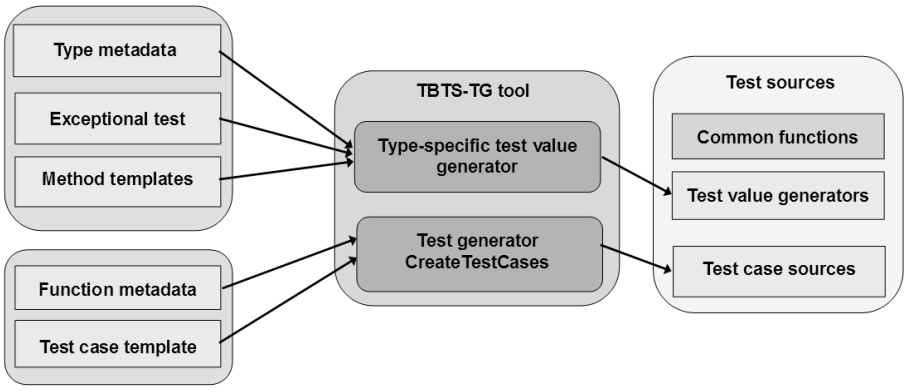

Fig. 3. Architecture of TBTS-TG tool

The first version of the benchmark suite consisted of standalone C programs that called the AIS API functions directly (outside of the AMF). In the current version the AMF service of the middleware starts the test programs configured as SA-aware components. To support the automatic execution of the benchmark suite a *test execution engine* was prepared. This engine runs the same test programs on each HA middleware, only the following tasks are implementation-dependent (as these are not standardized by the SA Forum): (i) construction of an implementation-specific configuration file on the basis of a common abstract configuration (which consists of one service group and one service unit containing the actual test case as a single component), and (ii) restarting the middleware between the runs of the test cases.

3.2 Mutation-based sequential test generator

While the TBTS-TG tool tests mostly individual functions, the mutation-based sequential test generator (MBST-TG) could be used to generate complex call sequences. The basic idea of the tool is that *mutation operators* representing typical robustness faults, like omitting a call or changing the specified order of calls, are applied to valid functional test programs that use the HA middleware. In this way a large number of complex robustness test cases can be obtained automatically.

The challenge of implementing the MBST-TG tool was the parsing and modification of the test programs' C source files. As the available free parsers encountered various problems when system header files were included in the input files, we followed a light-weight approach instead of obtaining the full parse tree (that is required for compilation). The srcML tool [7] was used to build an XML file representing only the *syntactic structure* of the input source files. This syntactic structure is enough to implement the common mutation operators.

Currently five mutation operators are implemented: omission, relocation and swapping of calls, modifying conditions, replacing parameters. The inputs of the MBST-TG (Figure 4) are the source files to be mutated and a configuration file that describes the parameterization of the mutation operator, e.g. the filters to be used when searching for a call to apply the mutation. Note that occasionally the mutation may result in such source code that cannot be compiled (data flow analysis is not performed, this way, for example, changing of function calls may result in using variables that were not assigned a value before).

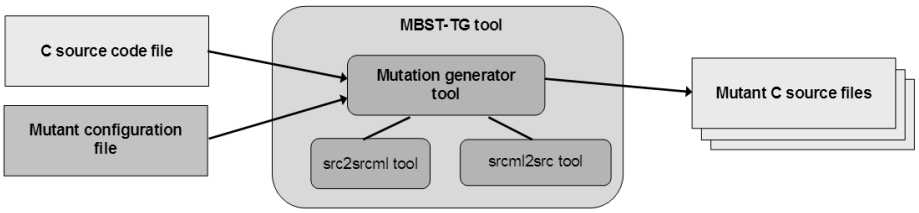

Fig. 4. Architecture of the MBST-TG tool

The mutant candidates came from two sources. The first one was the SAF Test [8] project, which is an open-source conformance test suite for SA Forum specifications. Because the test cases in SAF Test are redundant, 10 source files could be selected that cover the functionality of the others as well. The source files had to be slightly modified, because the current SAF Test does not use the required LDAP Distinguished Name (DN) format for component names. The second source was the functional test suite provided in openais, from which the testamf test file was used for mutation. The MBST-TG tool was configured to generate (i) two mutants using each operator in the case of each input file (using one operator each time) and (ii) ten mutants in case of each input file using two random operators each time. Altogether from these mutants 92 valid mutants were included in the test suite.

3.3 OS Call wrapper tool

The OS call wrapper intercepts system calls executed by the HA middleware and injects exceptional values into their return values (Figure 5). Since the middleware is tested here as a black box, the system calls can be triggered only indirectly, by starting a workload application.

The OS call wrapper can be configured to *intercept* or *delay* selected system calls. The return value of an intercepted call could be (i) the actual value returned by the original system call, if the call was also forwarded to the OS, (ii) a predefined valid or exceptional value or (iii) a randomly selected value from the possible error codes of the function. The wrapper is implemented using the Unix LD_PRELOAD variable, which can be used to load predefined libraries instead of system libraries.

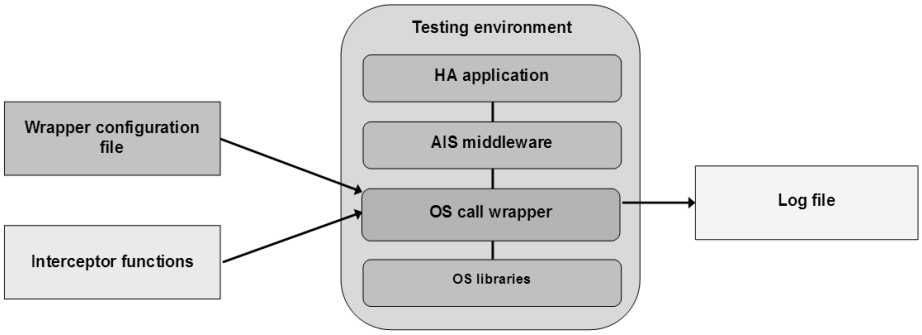

Fig. 5. Architecture of the OS call wrapper based testing

As a *workload* to trigger OS calls from the middleware, a synthetic HA application was prepared that resembles a search and index engine. The application utilizes the AMF and checkpoint service of the middleware. Using the *strace* utility all system calls of the middleware were logged during the execution of the workload application on both openais and SAFE4TRY, and the intersection of the two sets of OS calls was included in the benchmark suite, namely the functions accept, bind, close, gettimeofday, munmap, poll, sendmsg, setsockopt and socket.

4 Robustness testing results

The benchmark suite created by the above tools was used to test the robustness of the following implementations: (1) the *SAFE4TRY* evaluation package from Fujitsu Siemens Computers, which consists of the SAF AIS implementation RTP-SAF-L V2.1A and the PRIMECLUSTER cluster foundation, and (2) *openais*, an open source implementation of the AIS specification, including its version 0.80.1 (the latest stable release) and the trunk (the latest development version directly from the source control system of the project).

4.1 Results from the type-specific tests

Just by trying to compile the test suite on the system under test, several discrepancies were found: The header files used in openais differ in eight places from the official header files of the AIS specification, and thus from the header files used by the test suite. There is also one misspelling in SAFE4TRY's header files. Moreover, there are several types in the specification that are mapped to different types in the implementations, e.g. SaInt32T is mapped to *long* in SAFE4TRY and to *int* in openais.

Table 1 summarizes the exit codes of the test cases that were logged when executing the benchmark suite. Segmentation faults definitely indicate robustness failures, since in a HA middleware even invalid inputs should be handled correctly. Timeouts could indicate normal behavior, because some of the API functions could be parameterized to wait for an event to dispatch. However, while examining the concrete values used in the benchmark it turned out that the large number of timeouts in openais-trunk and openais-0.80.1 is not reasonable. Note for openais-0.80.1 there are less calls listed in the table because in case of saAmfProtectionGroupTrack the test program and the middleware crashed at the beginning of the test and no calls were executed for that functions.

Table 1. The number of test cases that exited with the given status code in case of type-specific testing of the different platforms.

Status code	openais-0.80.1	openais-trunk	SAFE4TRY
0 (success)	24568	26019	29663
11 (seg. fault)	1110	1468	0
14 (timeout)	467	2178	2

Segmentation faults occurred in 13 functions of openais-trunk and in 12 functions of openais-0.80.1. Timeouts were observed in 7 functions of openais-trunk, in 7 different functions of openais-0.80.1, and in one function of SAFE4TRY (namely, in saAmfDispatch when specifying a flag representing blocking; here timeout is the correct behavior). For the details, see Table 2.

Some of the test cases caused fatal error in the middleware. The tests for 14 functions in openais-0.80.1 and for 6 functions in openais-trunk produced an internal *assertion violation* and the middleware exited. The following two assertion violations were observed:

```
aisexec: amf_lib_exit_fn: Assertion `comp != ((void *)0)' failed.
aisexec: amfcomp.c:1142: amf_comp_register: Assertion `0' failed.
```

In the case of SAFE4TRY, after executing the test program for saAmfProtectionGroupTrackStop() the stopping of the middleware was not successful.

Table 3 details the different error codes for the successful calls. Every AMF call has a handle parameter, which is checked first before any operation. All tested AIS implementations could process the incorrectly initialized handles well, as it can be seen from the high number of SA_AIS_ERR_BAD_HANDLE codes. The number of SA_AIS_ERR_INVALID_PARAM codes show that SAFE4TRY detects much more invalid parameter combinations. When an assertion was violated in openais, all the

remaining calls for the given test program resulted in library error, that is the reason of the high number of SA_AIS_ERR_LIBRARY codes. In the case of SAFE4TRY, library errors were observed for the saAmfHealthcheckConfirm and saAmfHealthcheckStop functions. In both versions of openais a significant number of test cases returned invalid error codes, which cannot be considered as a robust behavior.

Table 2. Functions that produced robustness failures in case of type-specific testing

Failure	openais-0.80.1	openais-trunk
seg. fault	saAmfComponentErrorClear, saAmfComponentErrorReport saAmfComponentNameGet, saAmfComponentRegister, saAmfComponentUnregister, saAmfHAStateGet, saAmfHealthcheckConfirm, saAmfHealthcheckStart, saAmfHealthcheckStop, saAmfInitialize, saAmfProtectionGroupTrackStop, saAmfSelectionObjectGet	saAmfComponentErrorClear, saAmfComponentErrorReport saAmfComponentNameGet, saAmfComponentRegister, saAmfComponentUnregister, saAmfHAStateGet, saAmfHealthcheckConfirm, saAmfHealthcheckStart, saAmfHealthcheckStop, saAmfInitialize, saAmfProtectionGroupTrack, saAmfProtectionGroupTrackStop, saAmfSelectionObjectGet
timeout	saAmfComponentErrorClear, saAmfComponentNameGet, saAmfCSIQuiescingComplete, saAmfDispatch, saAmfInitialize, saAmfHealthcheckConfirm, saAmfProtectionGroupTrackStop	saAmfComponentErrorClear, saAmfComponentNameGet, saAmfComponentUnregister, saAmfCSIQuiescingComplete, saAmfDispatch, saAmfProtectionGroupTrack, saAmfProtectionGroupTrackStop

Table 3. The number of test cases that finished and returned the given SaAisErrorT error code in case of type-specific testing of the different platforms.

Error code	openais-0.80.1	openais-trunk	SAFE4TRY
SA_AIS_ERR_BAD_FLAGS	0	0	384
SA_AIS_ERR_BAD_HANDLE	18828	20408	20708
SA_AIS_ERR_EXIST	0	0	1
SA_AIS_ERR_INIT	0	0	6
SA_AIS_ERR_INVALID_PARAM	56	226	6073
SA_AIS_ERR_LIBRARY	3953	2316	52
SA_AIS_ERR_NOT_EXIST	0	1296	1786
SA_AIS_ERR_NOT_SUPPORTED	0	0	144
SA_AIS_ERR_TRY_AGAIN	30	30	0
SA_AIS_ERR_VERSION	336	336	294
SA_AIS_OK	86	128	215
invalid error code	1279	1279	0

In our previous work [4] version 0.69 of openais (based on version A.01.01 of the AMF specification) was used for benchmarking. In comparison with these previous experiments, the following could be observed: the simple method of using only invalid pointers and integer values as exceptional parameters did not activate so many robustness failures in the current versions of openais. One of the reasons for this is that moving to version B.01.01 of AMF the number of pointer parameters decreased significantly. 58.6% of the tests in the type-specific robustness test suite resulted in segmentation fault for version 0.69, while this number was only 4.2% and 4.9% for the 0.80.1 and trunk versions, respectively. Thus, the robustness of openais was definitely improved, although it still lags behind the robustness of SAFE4TRY, where the only robustness problem discovered by the benchmark suite was the error code SA_AIS_ERR_LIBRARY for two functions.

4.2 Results from the mutation-based testing

The mutant test sequences obtained from SAF Test and testamf were executed on the three implementations. The number of observed robustness failures is summarized in Table 4.

Table 4. The number of observed robustness failures / the total number of executed test cases in case of mutation-based testing of the different platforms.

Input	openais-0.80.1	openais-trunk	SAFE4TRY
SAF Test	8 / 63	0 / 63	1 / 63
testamf	22 / 29	28 / 29	0 / 29

The robustness failures discovered by the SAF Test mutants were the following. In case of eight mutants, openais-0.80.1 exited with one of the previous or with the following assertion:

```
./aisexec: symbol lookup error: /opt/openais-
0.80.1/exec//service_amf.lcrso: undefined symbol: assert
```

In SAFE4TRY, when stopping the middleware after one of tests the following error occurred:

```
Error in communication! ERROR: Stopping AMF subsystem was not
successful
```

Note that the SAF Test programs are constructed in such a way that the return value is checked after each function call, and if it does not match the predefined value then the program is aborted with an error message. This feature of the SAF Test programs makes them difficult to be used in robustness tests, because the subsequent calls are not executed if a wrong return value is detected.

When the testamf mutants were executed as AMF components in openais-trunk and openais-0.80.1 the CPU utilization increased to 100% and a hard reset had to be performed. Thus, Table 4 contains the results from running the testamf mutants as standalone programs. During the experiments with the mutants the above detailed assertions were also observed.

It could be observed that mutation based robustness testing highlighted additional robustness failures that were not detected by the type-specific tests. It gives reasons for applying such complex test sequences.

4.3 Results from the OS wrapper

For each of the 9 system calls (see Section 3.3) a separate test case was executed by starting the workload application and after a while forcing a failover. During the execution the system calls were forwarded to the OS, and with a predefined probability a random error code was returned (the probability depended on the frequency of the call, which was determined in probe runs).

Table 5. The system calls that provided the given outcome using the OS call wrapper.

Outcome	openais-0.80.1	openais-trunk	SAFE4TRY
No failure observed	accept, close, gettimeofday, munmap, sendmsg, setsockopt	accept, bind, close, gettimeofday, sendmsg	accept, close, gettimeofday, sendmsg, setsockopt
Application failed	-	munmap, setsockopt	poll
Middleware failed	bind, poll, socket	poll, socket	bind, munmap, socket

The first row of Table 5 lists the system calls in which case the workload application was executed successfully in spite of the injected fault. The second row shows such cases when the application exited but the middleware did not fail. The last row indicates the test cases when also the middleware exited (typically silently, without error messages). Note that due to the random injection of error codes, these latter cases just indicate potential robustness faults without objectively comparing the implementations.

5 Conclusion

In this paper a robustness testing approach for HA middleware systems was presented. The novelty of the approach is the application of *automatic tools* that construct the test cases systematically on the basis of the *standard interface specification* (API functions) and existing functional test suites. The robustness testing of the HA middleware implementations demonstrated that these tools can be used efficiently and their test results are complementary as they detect distinct failure types. It turned out that there are still *several robustness problems* both in version 0.80.1 and in the trunk version of the openais implementation. SAFE4TRY turned out to be much more robust with regard to the exceptional inputs generated by the benchmark suite. It is important to emphasize, however, that robustness testing was used only to observe these problems, and further work is needed to find the causes and to turn the observations into dependability benefits, e.g. by identifying the wrong implementation ap-

proaches or coding errors that shall be corrected. The work with AIS-based implementations will be continued in the HIDENETS project (IST 26979) which develops resilience solutions for distributed applications.

References

1. Service Availability Forum, Application Interface Specification, February 2006., URL: http://www.saforum.org/
2. P. Koopman *et al.*, "Automated Robustness Testing of Off-the-Shelf Software Components," in *Proceedings of Fault Tolerant Computing Symposium*, pp. 230-239, Munich, Germany, June 23-25, 1998.
3. K. Kanoun *et al.*, "Benchmarking Operating System Dependability: Windows 2000 as a Case Study," in *Proceedings of 10th Pacific Rim International Symposium on Dependable Computing*, Papeete, French Polynesia, 2004.
4. Z. Micskei, I. Majzik and F. Tam, "Robustness Testing Techniques For High Availability Middleware Solutions," in *Proc. of Int. Workshop on Engineering of Fault Tolerant Systems (EFTS 2006)*, Luxembourg, Luxembourg, 2006.
5. OpenAIS, AIS implementation, URL: http://developer.osdl.org/dev/openais/
6. Fujitsu Siemens Computers, SAFE4 Continuous Services, SAFE4TRY version, URL: http://www.safe4cs.com
7. Software Development Laboratory, srcML, URL: http://www.sdml.info/projects/srcml/
8. SAF Test, SAF-conformance test suite, URL: http://saftest.sourceforge.net/