ROBUSTNESS COMPARISON OF HIGH AVAILABILITY MIDDLEWARE SYSTEMS

Zoltán MICSKEI
Advisor: István MAJZIK

I. 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 to the Service Availability Forum (SA Forum) to elaborate the Application Interface Specification (AIS [1]). The benefit of an open specification would be for example the easier integration of different off-the shelf components.

With multiple middleware products developed from the same specification the demand to compare the various implementations naturally arises. The most frequently 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 one of the attributes of dependability, *the robustness* of HA middleware systems.

II. 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*. Based on earlier robustness testing projects (like *Ballista* [2]) and dependability benchmarks (like *DBench* [3]) we elaborated an approach for high availability middleware systems, which was presented in [4].

The first step of developing the test strategy was to identify 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*: These errors are related to the application, thus they are not covered by the robustness testing of the middleware.
2. *Operator errors*: In general, operator errors mainly 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 systems 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.

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 today's 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 messages, 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 benchmark suite.

III. 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 on Figure 2. In the following, we describe these tools and their application during robustness testing.

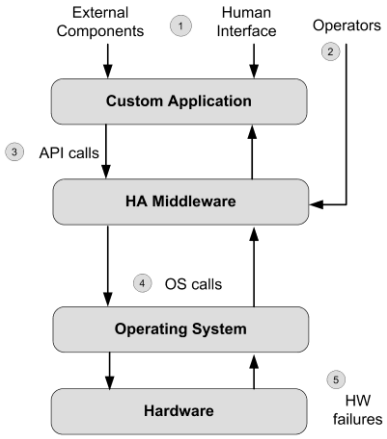

Figure 1: HA middleware fault model

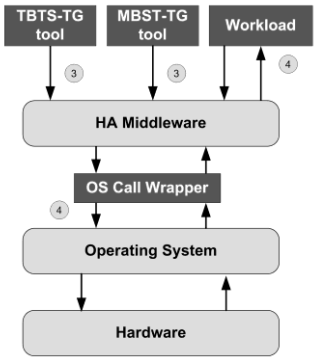

Figure 2: Testbed tools

A. 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 function, the exceptional values are defined for the parameter *types* that are used in the API functions. From the description of these types, the tool generates a *test program* for each API function, that 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 it has caused a segmentation fault or a timeout. To help diagnosing the robustness faults, in the first calls only a single parameter was assigned an exceptional value (using valid values in the case of the remaining parameters).

The input for the tool consists of the following elements. The skeleton of the test program is given 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. For complex structures, each member is assigned a combination of invalid values while the others remained valid.

B. *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 using 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 input source files. Parsing standard C files is not as trivial as it seems, the available free parsers encountered various problems when system header files were included in the input files. Thus instead of obtaining the full parse tree (that is required for compilation) a light-weight method was used. The srcML tool 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. The inputs of the MBST-TG 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 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). Finally 92 valid mutants were included in the test suite.

C. *OS Call wrapper*

The OS call wrapper intercepts system calls executed by the HA middleware and injects exceptional values into them. To include all important system calls used by the middleware a workload application is needed, which generates calls to the middleware resembling typical application activities.

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.

As a workload to explore the OS system calls from the middleware, a general application was used that implements a search and index engine. The application utilizes the failover and checkpoint service of the middleware. Using the *strace* utility the system calls of the middleware were monitored, and 10 calls were selected for interception.

IV. Robustness comparison results

Using the benchmark suite created by the above tools, the robustness of *openais*, an open source implementation of the AIS specification was evaluated. Two versions of openais were used in the tests: 0.80.1 (the latest stable release) and the trunk (the latest development version directly from the source control system of the project).

Results from the type-specic 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 several places from the official header files of the AIS specification, and thus from the header files used by the test suite.

Table 1 summarizes the results obtained by executing the test cases. Segmentation faults definitely indicate robustness failures, since in a HA middleware even invalid inputs should be handled correctly. Timeout could indicate normal behavior, because some of the API functions could be parameterized to wait for an event to dispatch. However, the large number of timeouts in openais-0.80.1 is not reasonable taking into account the concrete values used in the benchmark.

Some of the test cases caused fatal error in the middleware. The tests for 5 functions produced an internal *assertion violation* and the middleware exited. Different assertion violations were observed, they were produced at the registration phase of the application component.

In openais-0.80.1 segmentation faults were observed in 12 functions while timeout were found in 7 functions. These numbers were slightly lower in openais-trunk (9 and 2, respectively), which indicates an improvement achieved during the development of the openais implementation.

Table 1: Resulting status code of test cases for two versions of openais

Status code	Number of test calls in openais-0.80.1	Number of test calls in openais-trunk
0 (success)	25610 (91%)	27863 (96,1%)
11 (segmentation fault)	1336 (4,8%)	1048 (3,6%)
14 (timeout)	1183 (4,2%)	94 (0,3%)
Total	28129	29005

Results from the mutation-based testing: Some of the mutants produced a third kind of assertion violation in the middleware, while others resulted in timeout. In the case of openais-trunk, the test results were the following: assertion violation 47,6%, timeout 33,3% and finished 19%.

Results from the OS wrapper: For each of the identified system calls a separate test was executed, where the workload application was started and after a while a failover was forced. During the execution the system calls were forwarded to the OS, and with a predefined probability a random error code was returned (the probability was adaptive as it depended on the frequency of the call). In the case of two OS functions no robustness failures were observed, another two functions were not called by the actual version of the workload. For three OS functions the CPU utilization was raised to 100% and the test could only be terminated by a hard reset. In two cases the middleware detected the failure and exited because it could not recover in other way. In one case an assertion was triggered during the execution. These latter results indicate robustness failures.

V. 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 an open source HA middleware 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. It is important to emphasize that robustness testing can be 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 approaches or coding errors that shall be corrected.

References

[1] Service Availability Forum, URL: http://www.saforum.org/

[2] P. Koopman et al, “Automated Robustness Testing of Off-the-Shelf Software Components,” in *Proc. 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 *Proc. of 10th Pacific Rim Int. 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*, Luxembourg, Luxembourg, 2006.