IMPROVED METHOD OF TESTING DISTRIBUTED SYSTEM INTERFACES USING SIMULATION TESTS

VADYM MUKHIN, YAROSLAV KORNAGA, YURII BAZAKA, ANDRII BARABASH AND OLEG MUKHIN

National Technical University of Ukraine,
Igor Sikorsky Kyiv Polytechnic Institute,
Peremohy Ave, 37, Kiev, Ukraine

(received: 1 July 2021; revised: 19 August 2021;
accepted: 21 August 2021; published online: 30 Ocotober 2021)

Abstract: In this paper a modification of Mike Cohn's test pyramid is described for adaptation during testing in distributed information processing systems which allows expanding the possibilities of testing and applying the features of such systems. Recommendations for further use of the mechanisms of modified Mike Cohn's pyramid are developed.

The method of testing the user interface software of the nodes of a distributed system was improved to differ from the existing techniques by including a mechanism of simulation of its operation to allow testing of individual components of the system interface.

It is shown that in comparison with end-to-end testing of user interfaces the advantages of using the mechanisms of user interface test simulators allow reducing the time spent on testing any UI service. The time is reduced by decreasing the number of simultaneous user interface services.

With a small number of nodes, end-to-end testing of user interfaces is faster than simulation testing of the same user interfaces. As the number of nodes increases, the time required to test the services of a distributed system by simulation tests becomes shorter than the time required to test the same system by a traditional method.

Keywords: software testing, distributed information processing system, information system.
DOI: https://doi.org/10.17466/tq2021/25.2/f

1. Introduction

Modern systems are divided into different types which use different testing variants. For different levels of systems, finding errors in the software leads to the development of automation mechanisms [1–5]. It can be helped by continuous testing which ensures the development speed compliance and provides the required quality. In continuous testing one uses an assembly of automated tests and their

application to the developed environments. The assembly, testing and deployment of the environment with a continuously increasing number of system nodes are ineffective with the use of manual methods [2, 6, 7].

In automated testing, **one** uses the basic concept expressed in the pyramid of tests. It was suggested by Mike Cohn in his book *Succeeding with Agile: Software Development Using Scrum.* In his pyramid, there is a relationship between the levels of testing and the number of tests to be performed.
Mike Cohn's original test pyramid consists of three levels (from bottom to top) [1, 8, 9]:

- unit tests;
- service tests (API testing);
- end-to-end user interface tests.

When using Mike Cohn's pyramid, the speed and isolation of tests vary according to the following rules [1, 7, 10]:

- the speed of tests decreases from the bottom up (from unit tests to user interface tests);
- isolation of the tested objects decreases from the bottom up (from unit tests to user interface tests).

As it is easy to use, the test pyramid is a mechanism used to create test sets. We can conclude that there are two principles of using the test pyramid [5, 9]:

- write tests with various details;
- the higher the level, the fewer the tests.

2. Testing approach based on Mike Cohn's test pyramid modification

In Mike Cohn's test pyramid, the components are responsible for: block tests (testing certain areas of the program code, calling methods), service tests (testing system components to bypass the interface) and end-to-end user interface tests (testing the entire system using a graphical interface). In the "bottom-up" direction, the scope of the tests increases because the simplest components of the system and its individual functions are tested first, then the interaction of its components, and the next step is to test the entire system operation. The size of the tests, the complexity of their formation and the time spent on testing also increase. Accordingly, the "top-down" direction reduces the difficulty of finding the cause of failure of the tests: the failure of the block test location can be determined quite accurately, until the wrong line in the program code in some cases. It is very difficult to establish the cause of failure of the end-to-end tests due to their size and complexity of formation.

Since Cohn's pyramid was originally developed for testing of non-distributed systems, we will modify it so that it fully uses the architecture of distributed

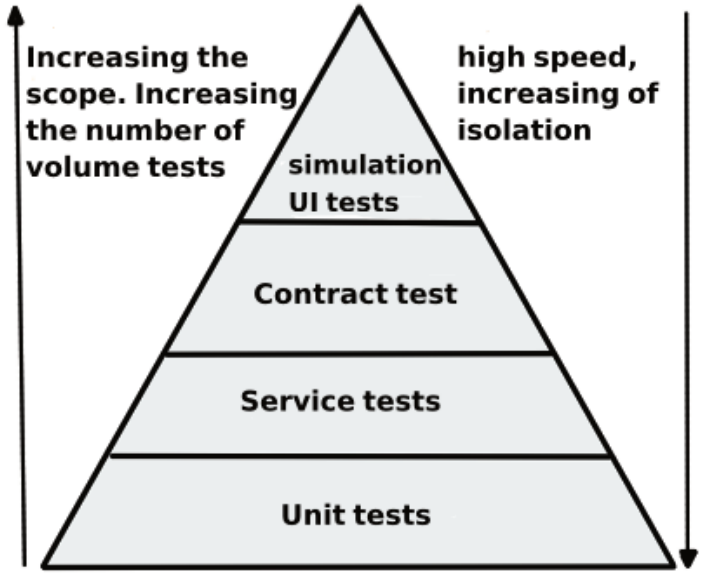

Figure 1. Modified Mike Cohn's pyramid for testing of distributed data processing systems

data processing systems. To do this, we will consider different types of tests and determine the effectiveness of their implementation.

Block tests are performed at the level of code fragments, hence, it makes no sense to replace them with tests of another type. In modified Cohn's pyramid, which is shown in Fig. 1, the lower level remains unchanged. In this case, service tests and end-to-end tests are replaced by taking into account the architecture of distributed systems.

Obviously, using end-to-end tests in an architecture of distributed systems is an inefficient approach, as it requires a guarantee that when deploying a new subsystem in real applications, the changes will not conflict with the operation of other subsystems.

One way to implement it without the use of real subsystems is to use the so-called "contracts" which are compiled on the basis of requests from the subsystem. A contract means a code test that runs in the sender mode. When contracts are used, it is necessary to determine the consumer expectations from this service. These tests must be performed for a single supplier who is in isolation, therefore, such tests will be much faster and more reliable than the usual end-to-end tests for testing API services.

3. Functional substitution mechanism for testing distributed system services

The advantage of a distributed data system architecture in comparison to monolithic architectures is that each part (service) is an independent unit. This unit has its own API and can connect to other APIs. The method of substituting

the functionality with simulators allows test service functionalities with contract tests. This testing mechanism is applied to user interfaces. This mechanism allows a significant decrease in the time for interface components testing, as there is no need to install the entire system, but simply install one interface module and test it. It will also reduce the disk space and the memory resources required on virtual machines.

Let us consider a fragment of a distributed data processing system which has services that are responsible for the information exchange with users, in addition to the usual services (Figure 2).

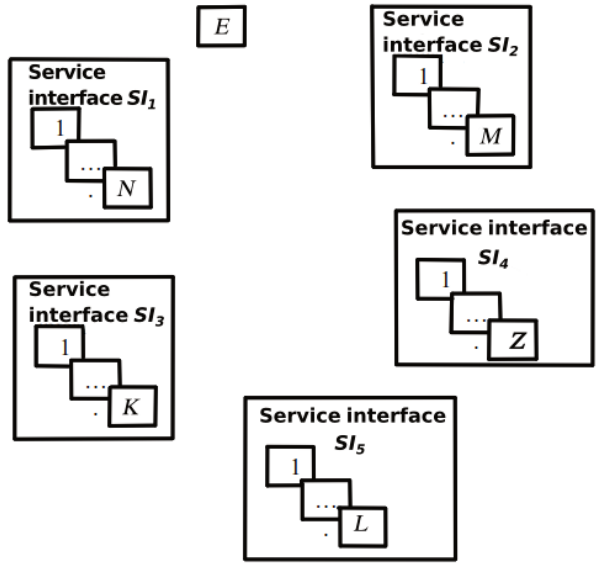

Figure 2. Distributed system of user service interfaces

Service interfaces SI_1, SI_2, SI_3 contain a number of components N, M, K, respectively. E is the entry point into the system from the external environment through which the user may operate the system. Service interfaces SI_4, SI_5 contain a number of Z and L components, respectively.

Let $ti_1(i)$ be the time of deployment of the service interface SI_1, where $i = 1,2,...,N$ is the number of components, $tl_1(i)$ is time of the service interface SI_1 components loading, $tt_1(i)$ is the time of service the interface SI_1 testing. The procedure is similar for other distributed system service interfaces.

During testing of the user interface components, another important parameter must also be taken into account: the deployment time ts of the test environment, as this stage is very time consuming in large systems.

Accordingly, the total time T that will be spent on testing all the distributed system components will be equal to the sum of the environment deployment time, the deployment time of the services and service interfaces, the loading time of the

interfaces, the testing time of services and service interfaces, as well as the whole environment clotting time, i.e.:

$$T = \sum_{i=1}^{N}(ti1(i)+tl1(i)+tt1(i)) + \sum_{i=1}^{M}(ti2(i)+tl2(i)+tt2(i)) + \sum_{i=1}^{L}(ti3(i)+tl3(i)+tt3(i)) + \sum_{i=1}^{K}(ti4(i)+tl4(i)+tt4(i)) + \sum_{i=1}^{Z}(ti5(i)+tl5(i)+tt5(i)) + t(s) \quad (1)$$

During end-to-end testing of a distributed system interface, difficulties arise when there are two or more services. For example, in order to test the SI_1 service interface, first we need to deploy all services and then go through the loading of the service interface components.

The time T_{SI2} that is spent on testing the SI_2 service interface is the testing time of the whole system because all the services need to be installed for full end-to-end testing. Therefore, this formula demonstrates that end-to-end tests in the architecture of distributed systems require installing the entire system and, accordingly, spending the time and resources thereon.

We develop a new mechanism for testing which is based on the use of simulators. The simulator (stub) is a service that is installed along with the service interface and allows testing it.

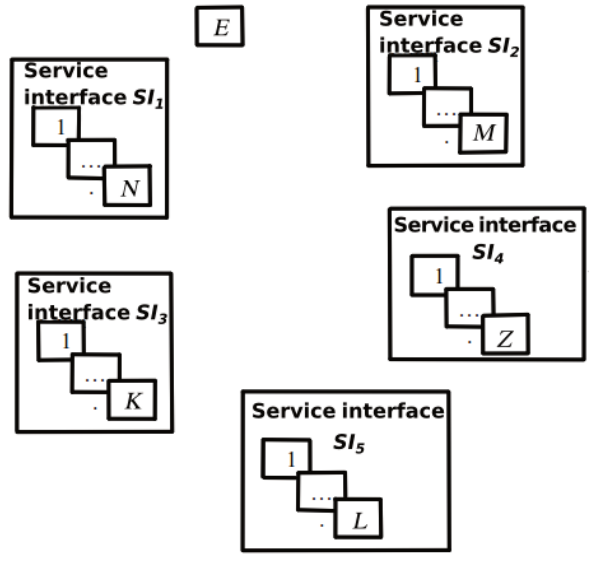

Figure 3. Distributed structure of user service interfaces with simulators

Let us add simulators for the testing of service interfaces into the general system. In this case, we use two types of simulators: input simulator V and output

simulator W. Input simulators send prepared information to the service interface, which is necessary to download all the user interface components by the normal service. Moreover, there may be a situation, when the input requires several input simulators to cover different types of requests to the interface. The output simulator operates in the opposite direction, i.e. it checks the responses that were sent by the service interface. The service interface considers the input and output simulators as services for the information exchange (Figure 3).

The parameters of input and output simulators consist of an array of "request-response" pairs, and each component contains the following parameters:

1 Request: request headers (Headers), request type (POST, PUT, DELETE), entry point (the controller where the request will go), the body of the request;
2 Response: the status (code) of the response, the body of the response.

Despite the fact that the input and output simulators have the same parameters, they operate in significantly different ways.

The input simulator sends the entire array of requests to the tested service at once. The request is sent automatically before the tests run. Accordingly, during and before testing the service, there is possibility to correctly load all user interface components, for the full operation of which the information from other services is required.

The output simulator contains an array of expected queries and responses to them. When the tested service sends a request to the simulator, the simulator is looking for a similar request in the array of its expected requests. When a similar request is found, the simulator sends the prepared response to the service. If there is no answer found, the simulator sends the answer with the code 404 (Not found) and informs the tester about this answer. It is the tester's responsibility to find the cause of such behavior, and may be associated with the system malfunction, i.e. is a valuable error signal in the service.

Then one service testing time of the interface SI_1 is equal to:

$$T = \sum_{i=1}^{N} (to1(i) + tl1(i) + tt1(i) + tsi2(i) + tsi4(i) + tso5(i)) + t(s) \qquad (2)$$

where $t_{SI2}(i)$ is the deployment time of the input simulator from the service interface SI_2, $t_{SI4}(i)$ is the deployment time of the input simulator from the service interface SI_4, and $t_{SI5}(i)$ is the deployment time of the input simulator from the service interface SI_5.

The advantages of this approach are obvious compared to end-to-end testing. The time spent on testing any service interface is much shorter than for the end-to-end testing of the same service. For example, in a distributed system that was considered to test service interface SI_1 components, we do not need to deploy four services and wait for service components SI_2, SI_3, SI_4, SI_5 to load, and to test service components SI_2, we do not need to deploy all five services and wait for the loading components of service interfaces SI_1, SI_3, SI_4, SI_5, etc.

4. Experimental analysis of the average time of user interface tests with input-output simulators

A website with a distributed structure of interfaces was chosen for the experiment. The VueJs framework and the TypeScript programming language were used to develop the interfaces. The development environment was Visual Studio Code 1.53.

Static interface files were sent to Amazon S3 and CloudFront to deploy the distributed system.

During the experiment, two types of testing were performed: end-to-end testing and simulation testing. The Google Chrome browser was used for both testing types.

The Selenium WebDriver framework was used to interact with the browser during end-to-end testing, which allows the user's behavior in the browser to be reproduced using code commands. During the end-to-end testing, all the distributed system interfaces were deployed, a browser was launched, and an array of all tests was launched.

In the input simulation the shallowMount library was used for the input simulator, and the Jest library was used for the operation of the output simulator. ShallowMount is a library of mountebank – a framework to simulate the behavior of ARI services [11]. Jest is a framework for writing API tests in Javascript. It makes it very easy to create simulators for any objects and has very clear documentation [12]

We used the Catcher framework and an improved method using simulation tests to test distributed system interfaces. It took only 12 tests to test this system with Catcher, but the time to pass one test was 21 seconds. It took 18 tests of 12 seconds each to test the same system using simulation tests. Since each service is deployed separately during the simulation test, the total deployment time of the services during the simulation tests was 30 seconds, while it took 14 seconds to deploy the entire system with the Catcher framework test. The total time required to test the entire distributed interface system with Catcher was 286 seconds, while the testing time for the same system using simulation tests was 265 seconds. Accordingly, the improved simulation testing method was 7.3% faster than the existing Catcher framework.

Table 1. Test results

Number of nodes	Selenium WebDriver	Simulation	%
2	120	150	-25
5	286	265	7.3
10	915	830	9.3
15	1812	1556	14.2
20	3633	2930	19.4

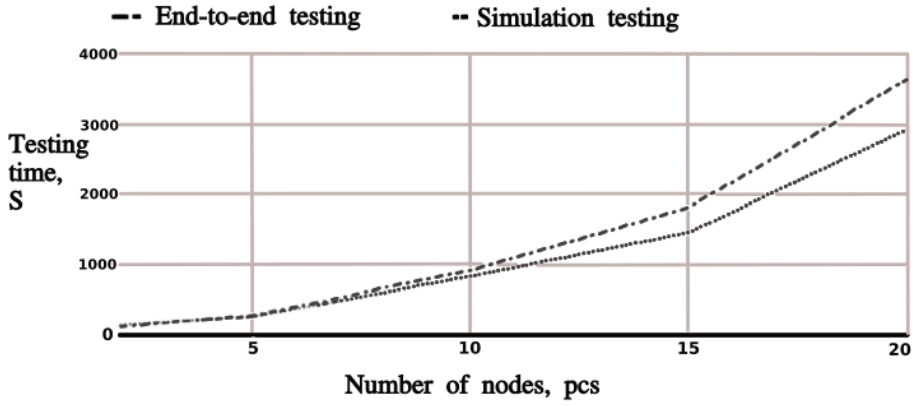

Figure 4. Comparison of the average time of testing interfaces by different methods

Both methods were tested on distributed systems with 2, 5, 10, 15 and 20 nodes. For each method, the average test time was measured and the average time was calculated, as is shown in Table 1 and in Figure 4.

Hence, the end-to-end testing is faster than simulation testing, and as the number of nodes increases, the time required to test distributed interface system services with simulation tests is shorter than the time required to test the same system with the end-to-end method.

5. Conclusion

A modification of Mike Cohn's test pyramid was considered in this paper. According to the pyramid testing mechanisms, it is worth writing a lot of small-volume tests and quick unit tests. To write more general tests, very few high-level end-to-end tests that test the software from start to finish need to be used. At the same time it is necessary to watch that the used pyramid mechanisms do not lead to big time expenses as a result.

To conduct the experimental research, a framework was developed for testing distributed information processing system services with the use of mechanisms based on the application of of simulation tests. During the experiment, two types of distributed system testing were performed: end-to-end testing and simulation testing.

The method of testing the user interface of the distributed system nodes software was improved to differ from the existing techniques by including a mechanism of simulation of its operation to allow testing of individual components of the system interface.

It is shown that in comparison with end-to-end user interfaces testing, the advantages of user interface testing simulators allow reducing the time spent on testing any service of the user interface. The time is reduced by decreasing the number of simultaneous user service interfaces.

In particular, it is shown that with a small number of nodes, end-to-end testing of user interfaces by the Selenium WebDriver framework is faster than simulation testing of the same user interfaces. As the number of nodes increases, the time required to test the services of a distributed system by simulation tests becomes shorter than the time required to test the same system by the end-to-end method.

For 5 interfaces of the distributed system the testing time decreased by 7.3% in comparison with Selenium WebDriver.

The proposed method can be used for microservices architectures, as well as for the ARI distribution system. The main advantage of its use in microservices architectures would be the ability to test the interaction of microservices, if they are supported by different development teams. In this case, changes made to the microservice by one team will be immediately caught by tests of the other team. Thus, the probability of service interaction failure is minimized because such errors will be found at the development stage.

References

[1] Cohn M 2009 *Succeeding with Agile: Software Development Using Scrum*, Addison-Wesley
[2] Mukhin V, Kornaga Ya, Mostovyi Y and Bazaka Y 2016 *A Model For Events Monitoring Heterogeneous Distributed Databases Based on Vector-matrix Operations*, The Far East Journal of Electronics and Communications, **16** (3) 645
[3] Zhenbing H, Mukhin V, Kornaga Ya, Herasymenko O and Bazaka Y 2017 *The scheduler for the grid system based on the parameters monitoring of the computer components*, Eastern European Journal of Enterprise Technologies, **1** (2-85) 31
[4] Mukhin V, Kornaga V Y, Tkach M, Herasymenko O, Bazaka Y and Mukhin O *Subtask Prioritization on Workflow Execution in Distributed Wireless Computer System with Network-Centric Approach to Resource Control*, 5th IEEE International Symposium on Smart and Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems, IDAACS-SWS 2020, 17 September 2020, Dortmund, Germany
[5] Kosenko V, Persiyanova E, Belotskyy O and Malyeyeva O 2017 *Methods of managing traffic distribution in information and communication networks of critical infrastructure systems"*, Innovative technologies and scientific solutions for industries, **2** (2) 48
[6] Martsenyuk V, Didmanidze I, Sverstiuk A, Andrushchak I and Rud K 2020 *Automated method of building exploites in analysis software testing*, Computer-integrated technologies: education, science, production, **39** 146
[7] Martynyuk N A, Ahmesh T, Drozd V O and Stepova S H 2018 *Checkability of hierarchical transmitions for behavioral check*, Systems and Technologies, **1** (56) 30
[8] Schmidt C 2016 *Agile Software Development Teams: the Impact of Agile Development on Team Performance*, Progress in IS 184
[9] Siavvas G M, Chatzidimitriou C K and Symeonidis L A 2017 *QATCH An adaptive framework for software product quality assessment*, Expert Systems with Applications, **86** 350
[10] Zeina S, Salleha N and Grundyb J 2016 *A systematic mapping study of mobile application testing techniques*, Journal of Systems and Software, **117** 334
[11] Mountebank - open source testing tool *http://www.mbtest.org*
[12] Jest - JavaScript Testing Framework *https://jestjs.io/*

Vadym Mukhin is a Professor at the Department of Mathematical Methods of System Analysis at the National Technical University of Ukraine (Igor Sikorsky Kiev Polytechnic Institute), D.Sc. Born on November 1, 1971. M.Sc. (1994), Ph.D. (1997), D.Sc. (2015) at the National Technical University of Ukraine (Igor Sikorsky Kiev Polytechnic Institute); Professor (2015). Major interests: security of distributed computer systems and risk analysis; design of information security systems; mechanisms for adaptive security control in distributed computing systems; security policy development for computer systems and networks.

Yaroslav Kornaga is a Professor at the Department of Information Systems and Technologies at the National Technical University of Ukraine (Igor Sikorsky Kiev Polytechnic Institute), D.Sc. Born on January 1, 1982. M.Sc. (2005), Ph.D. (2015), D.Sc. (2020) at the State University of Telecommunications; Associate Professor (2015) at the Technical Cybernetics Department. Major interests: distributed database security and risk analysis; distributed database design; mechanisms for adaptive distributed database security control; security policy development for distributed databases.

Yurii Bazaka is an assistant at the Computer Systems Department at the National Technical University of Ukraine (Igor Sikorsky Kiev Polytechnic Institute), Ph.D. Born on April, 1992. M.Sc. (2015), Ph.D. (2021) at the West Ukrainian National University Major interests: quality assurance, quality control, automation testing, server-side testing, testing of distributed systems

Andrii Barabash is a Ph.D. student at the Faculty of Informatics and Computer Science at the National Technical University of Ukraine (Igor Sikorsky Kiev Polytechnic Institute). SRE at Raiffeisen Bank. Born on April, 1992. M.Sc. (2015), Ph.D. (2021) at the West Ukrainian National University Major interests: quality assurance, quality control, automation testing, server-side testing, testing of distributed systems

Oleg Mukhin is a student at the Department of Mathematical Methods of System Analysis at the National Technical University of Ukraine (Igor Sikorsky Kiev Polytechnic Institute) Major interests: applied software for computer systems and networks.