# Patterns for Testing Distributed Systems Interaction

***Eduardo Guerra**, National Institute of Space Research (INPE)*
***Paulo Bittencourt Moura**, Department of Computer Science - University of São Paulo (USP)*
***Felipe Meneses Besson**, Department of Computer Science - University of São Paulo (USP)*
***Ayla Rebouças**, Federal University of Paraiba (UFPB)*
***Fabio Kon**, Department of Computer Science - University of São Paulo (USP)*

**Abstract:** *Testing distributed systems is a hard task, since the test needs to handle software pieces distributed across multiple network nodes. There are several techniques that can be used in this kind of tests to capture the communication among components, to verify the internal state of remote components or to emulate the behavior of one of the components. Each technique has its positive and negative consequences, being better applicable in some contexts. This paper presents three patterns that can be used in the test of distributed systems, focusing on the consequences and scenario in which each one is more suitable to be applied.*

## 1. Introduction

The goal of this paper is to present alternative techniques used to test the interaction among systems in a distributed architecture. All the patterns presented in this paper have the same context, however the solutions are different. Since they have distinct consequences, depending on the context, a different set of forces can have a higher weight and lead to one of this particular solutions.

The paper is organized as follows: Section 2 presents a general context, general forces and a problem that applies to all patterns; Section 3, 4, and 5 document three patterns that can be used in this context, including the solution, consequences, and known uses. Concluding the paper, Section 6 presents a discussion that compares the patterns, identifying scenarios in which each one is more suitable to be used.

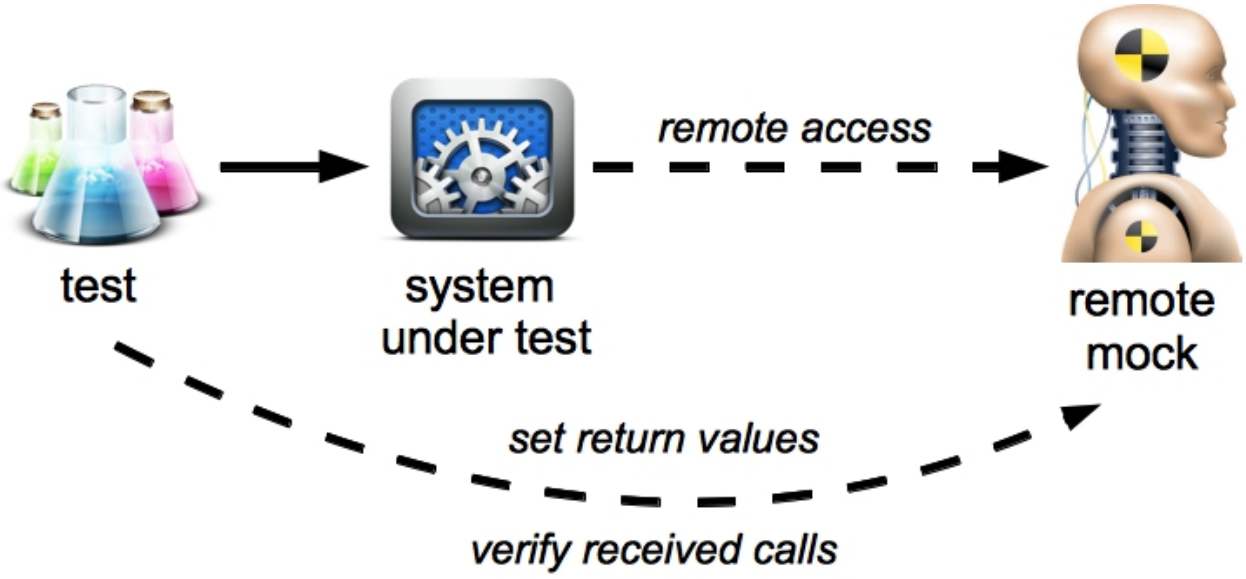
## 2. General Context

In the test of distributed systems, the message exchange among the involved systems is an important source of information. It can be used, for instance, to verify an expected response in a particular interaction or if a node is performing the expected remote calls in a given scenario. Automated tests, in such kind of architecture, should be able to access the exchanged messages to verify the functional behavior and to emulate different behaviors in a node to verify whether its neighbors are reacting accordingly.

**How to emulate remote systems behavior and monitor the exchanged messages when testing distributed systems?**

- ***Remote system access***. Depending on the level of access that the development team has on the remote system, it will be possible to use a more intrusive solution. In some cases, the team does not have access to the remote system during development due to restrictions of whom provide its services.
- ***Variation of the remote system implementation.*** In some environments, the remote access will happen in a specific system. However, in other cases, it can be connected to any system that implements a given interface. The inclusion of a remote system in the test environment can be influenced by the possibility of having different systems being accessed by it.
- ***Changing of implementation order***. Even when the same development team is responsible for all nodes of the distributed system, depending on the implementation order, the remote component may or may not exist yet.
- ***Test goal.*** The goal of the test being performed is important to determine which parts of the system are mandatory to participate on the test, and the level of intrusion that is acceptable. Some aspects that can be verified in distributed tests are expected message format (specially when you use **Consumer-Driven Contracts**), expected general behavior, and the interaction protocol (sequences, ordering, failures). The patterns then are related to one or more of those aspects. For instance, if the aim is to create an integration test of a single node of the system, it is acceptable to create a test that does not involve the other concrete nodes. In this case, the node neighbors can be mocked (emulated). However, for an entire system testing, all the nodes should be included.
- **Cost.** Each kind of test has a cost, in terms of time and money, to be created and to be maintained. The cost depends on the type of test that is being created. It is possible to make tradeoffs with the creation and maintenance cost, considering its level of automation. For instance, a test that need a manual environment configuration to be executed, can have a lower cost to create, but will need an additional effort to be frequently executed.

## 3. Remote Mock
## Also known as Remote Stub, Remote Test Double




**Use a test double to play the role of the remote component, allowing the test to configure its return values and verify its received calls.**

In this solution, the remote component will be replaced by a test double, which can also be called a remote mock or a remote stub. The system under test will still perform the remote calls through the network, but they will be handled by the test double. A configuration should be made in the network channel or in the system itself to redirect the calls to the remote mock. In order to verify the expected calls performed by the system under test, the test should have access to invoke the mock as well as to configure its responses.

The **Remote Mock** usually needs to be deployed in a server to be available for the system under test and for the test. They must provide the same interface as the remote component. If the remote system is a web service, for instance, the **Remote Mock** should be also deployed as a web service. In some cases, embedded servers can be used to make the test setup easier.

A remote mock replaces completely the remote component for testing purposes, making this approach suitable for scenarios where the remote system is not accessible or does not exist yet. This approach focuses on a single node of the distributed system, performing the test isolated from its neighbors.

For a matter of simplicity, doubles' configuration are limited - they are not a real system. Usually, they are configured with fixed responses, predefined by the tester. Thus, tests should not be intended to assert the content of the communication, but the format. Test goal should be to guarantee that the system under test communicates with the external system according to a given contract or interface. Another issue is related to the accuracy of those tests. They are only worthwhile if the tester can guarantee that the test double is faithfully implementing the remote system interface.

Additionally, it is important to reproduce the runtime environment as realistically as possible to test distributed systems. Non-functional behaviors in a remote system can affect the quality of the service under test. Hence, allowing to configure non-functional characteristics, such as response time, of the **Remote Mock** is worth for performance and scalability testing and, even, to enable functional testing of the SUT for unexpected behavior of the remote systems, when it could be expected to rollback an operation, for instance. Note that the accuracy of those tests is also limited by how reliable the runtime environment and remote system behavior are reproduced.

Dependency mocking is a valuable practice in Test-Driven Development (TDD) [Beck 2003] of traditional (non-distributed) software. In a similar way, in the development of distributed systems, remote mocking also helps the developer to narrow the testing scope, to deal with remote software dependencies, and to keep a testable software and sustainable architecture.

* * *

The test pattern **Mock Object** [Meszaros 2007] is similar to this one, but in a small scale. While the **Mock Object** substitutes an object that is dependent of a class under test, the **Remote Mock** substitutes a remote component accessed by a system under test. The **Annotated Test Step** [Floriano et al. 2011a] uses code annotations to configure actions and assertion for the test. An application of this pattern can be for the deployment configuration of the **Remote Mock**.

**Integration Contract Test** [Fowler, 2011] can be applied to guarantee the accuracy of the double's interface. It consists of running tests against the double to certify that they return the same content as the actual system. **Integration Contract Tests** are used by consumers to detect if the producing system is actually fulfilling the contract. They are also used by producers to indicate when they need to warn consumers about contract changes.

*The framework MakeATest* [Floriano et al. 2011b] *provides an infrastructure to use annotations to define initialization or assertions on test classes or methods. A plugin for that framework called MakeATest RMI [MakeATest RMI 2011] uses annotations to start and stop RMI servers and to add mocks as a remote object. Consequently, the test can interact remotely with the mock on the RMI server.*

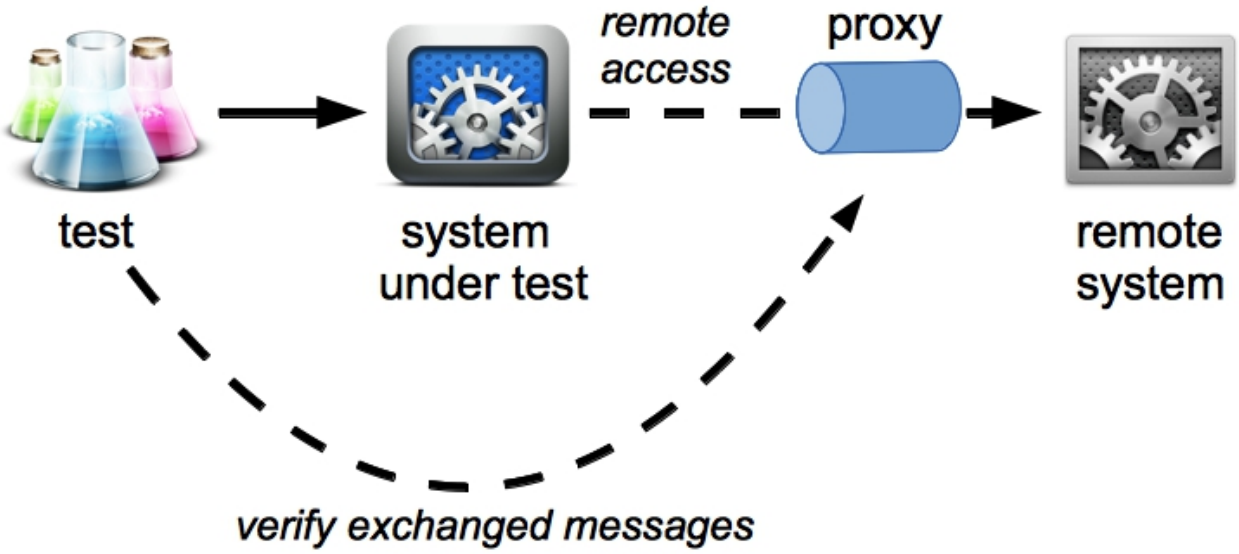
*SoapUI [SmartBear 2014] , an easy-to-use solution to create and run functional tests for web services, includes a mocking feature. Mock services can be created, configured and monitored through a graphical user interface. An embedded server is used to deploy the mocks. The tests in project LEONA, Transient Luminous Event and Thunderstorm High Energy Emission Collaborative Network in Latin America, uses SoapUI to mock a web service to perform functional tests in the main application.*

*Rehearsal [Besson et al. 2012] supports Test-Driven Development of web services. It includes a*

*Java API to create, configure and run service mocks programmatically, and is intended to be used with traditional automated testing frameworks, such as JUnit [JUnit 2014]. Rehearsal's mocks can be configured to provide specific response values for specific or general request values. They may also do not respond or provide faulty responses.*

*PUPPET [Bertolino et al. 2008] generates web service stubs with functional and non-functional behavior. It receives a WSDL specification, WS-agreement specification of QoS properties, and an automata model based specification of service functionality, and generates a service stub accordingly.*

## 4. Monitoring Proxy
## Also known as Testing Broker, Message Interceptor




**Use a monitoring proxy to intercept messages exchanged between remote services, allowing the validation of messages sent between the two systems.**

The goal, in this scenario, is to validate the messages exchanged between the system under test and the remote system. A proxy is set up between the system under test and the remote system so that all messages exchanged by them goes through the proxy. It, thus, also requires reconfiguration of the system or network channel to make the system under test communicate via the proxy. This proxy should provide to the test means to retrieve the content of the intercepted messages.

The **Monitoring Proxy** does not replace the remote system, which is included in the test. Proxies receive messages, store, and forward them. The remote system should be up to receive the message and provide a response. For the system interaction, the proxy existence is

transparent. After the interaction, tests can access the proxy information to verify if it happens as expected.

Usually the proxy does not interfere in the communication between the distributed systems. However, in some cases, it may increase the request payload to inform that the messages are for testing purposes. This information is usually added in the message header. The proxy could not forward the message to the other remote system, but this way it is acting like a **Remote Mock**. This strategy can be used, for instance, to simulate a timeout or error scenario.

The use of this pattern is suitable for integration tests, which should include all the nodes in the distributed system. The Monitoring Proxy is transparent to the system and it is not intrusive, in a sense that it does not interfere in the communication.

* * *

**Monitoring Proxy** applies the pattern **Broker** [Buschmann et al. 1996] for testing purposes. In this scenario, the service provided by the **Broker** is the storage of the messages to allow a further verification by the test. Similarly, the **Proxy** pattern [Gamma et al. 1994] can also be used to add a proxy inside one of the the applications to monitor the messages.

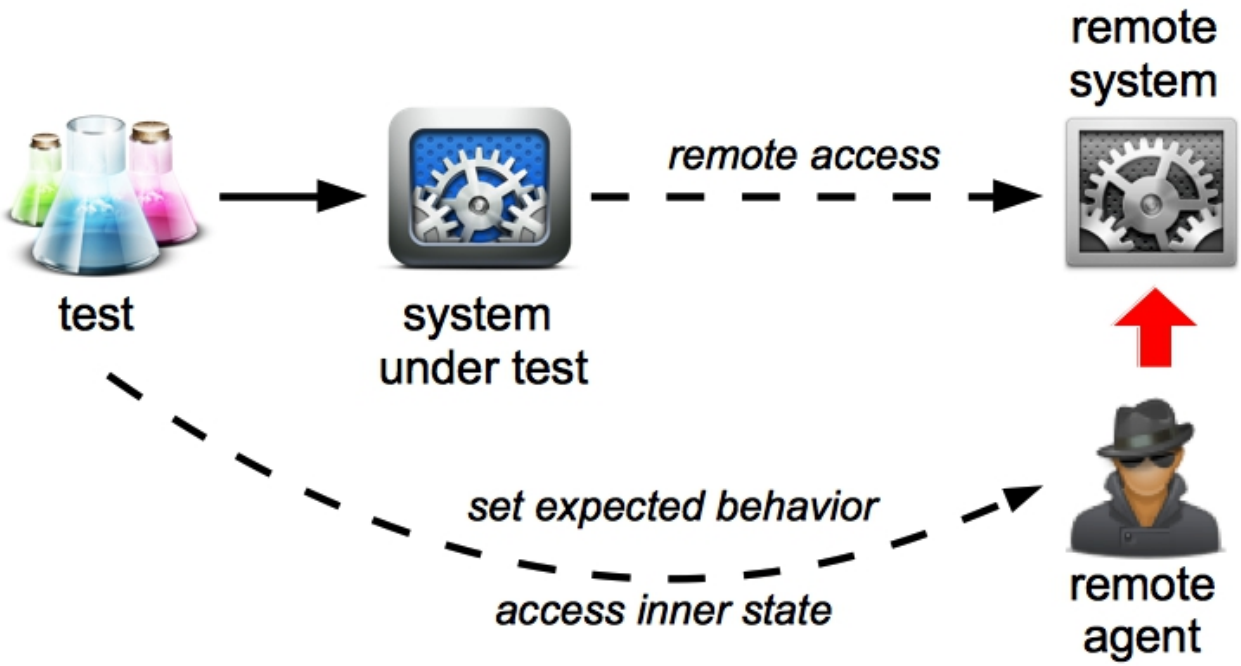
In a SOA architecture, the pattern **Messaging Metadata** can be used by the Monitoring Proxy to add information about the test on the messages [Erl 2009]. Additionally, the pattern **Endpoint Redirection** can be used to redirect the message to the **Monitoring Proxy**, instead of the regular remote system.

*Rehearsal tool provides a message interceptor feature which consists of a proxy that provides the same WSDL interface of the intercepted service. First, the proxy intercepts and stores the messages, for a later validation. Then, these messages are forwarded to the intercepted web service.*

*Intern is a toolkit that supports creating test cases for Javascript. The user can set the URL for a reverse proxy, allowing message interception to test AJAX calls [Intern 2013]. Thus, the test runner will load from that URL. The proxy should be set up with a regular Web server, such as nginx, Apache, or IIS.*

*A method has been proposed to use proxies for testing CORBA applications [Hoffmann et al. 2003]. Proxies are created from IDL specifications, with the support of QEDO toolset, with an additional step to include the code that implements proxy's logic. Tests are written with TTCN-3 to validate the interactions intercepted by the proxies.*

## 5. Remote Testing Agent




**Connect a testing agent to the remote system allowing the test to access and interfere in its internal state.**

Following this pattern an agent will be attached to the remote application and will be able to provide information about its internal state. Additionally, the agent could also be able to influence the remote system behavior allowing the configuration of a test scenario. However, the usual application is only to access its state. Then, the test can access the remote system through the agent to configure test scenarios and to perform verifications in its internal state.

Differently from the other two patterns in which the test focuses on the communication between the nodes, this one focuses on the expected result on the remote system, independent of the exchanged messages. This characteristic makes this approach suitable for tests that need to verify the behavior of a whole distributed system, without concerning about how the nodes interact. For instance, even if some services of the remote system are only accessed asynchronously, the remote agent can provide synchronous operations that testers can invoke to access the inner state of the system and perform the necessary verifications.

This approach is very intrusive and affects directly the remote system. To enable its use, the developer should have access to the remote system execution environment in order to plug-in the agent. The remote system should also accept agents to be plugged on it, which can be supported by the virtual machine or server where it is being executed, or by the application itself.

* * *

To Implement the support for agents that can be plugged in the remote system, some creational patterns that can detect the presence of an agent and create it in case of its existence can be applied. Examples of such patterns are **Service Locator** [Fowler 2004] and **Dynamic Factory** [Yoder et al. 2008]. The structure of an **Observer** [Gamma et al 1994] can be used to notify to the agents changes in the application internal state.

**Distributed Test Agents** [Farias et al. 2012] is a related pattern focused on techniques for the development of automatic system tests for distributed applications. It defines some test components that must be offered in the SUT to the testers. One of these components is the "Test Agent", which is responsible for performing some actions of the test scenarios in the remote test machines. Another component is the "component access point", which allows testers to exercise and verify the state of a system component. The "Test Agent" should know the access point of each component deployed in the remote machine where it is running.

*The Java Virtual Machine allows agents to be plugged-in. These agents can act when the system classes are being loaded to manipulate their bytecode allowing the addition of instrumentation to the classes. This mechanism is used for profiling tools used for performance tests in order to collect data from systems executing remotely.*

*JMX is Java Management eXtension API, and can be used to report the state of an application to the external world. A report from a developer in the Java.Net blog [Walend 2008] documents an usage of this API for testing in a TDD style of development. The JMX is used for the test to access remotely the application state.*

*OurGrid [Cirne et al. 2006] and the BeeFS [Souza et al. 2010] have used remote testing agents to abstract the way testers retrieve information about remote system components for their assertions. OurGrid is a middleware for open grid computing environments, it was developed in Java and it is in production since 2004. BeeFS is a distributed file system which aggregates the lazy disk spaces of workstations in a local network.*

*Project LEONA, Transient Luminous Event and Thunderstorm High Energy Emission Collaborative Network in Latin America, has some remote components that should return values based on hardware interactions. To test this feature, the server allows an agent to provide a service which allows to set results and retrieve information about the hardware interaction.*

## 6. Discussion

This paper presents three patterns for testing the interaction between remote systems. Each one has its own advantages, fragilities, and scenarios in which they are more suitable. This section presents a comparative analysis between the techniques.

A **Remote Mock** is more suitable when the remote component cannot be included in the test. Reasons for that can be because it is not accessible or because it has not been developed yet. By using this pattern, the test focus is on a single node of the system, verifying if it interacts correctly in a remote collaboration.

The **Monitoring Proxy** is the less intrusive technique, does not interfering in the nodes and does not replacing a node by a fake one for testing purposes. That characteristic makes this technique suitable for scenarios in which the test should verify the behavior of an entire distributed system. The verifications should focus on the messages exchanged by the nodes to inspect if the behavior is the expected.

Finally, the **Remote Testing Agent** is the most intrusive technique and it directly interferes in one of the nodes. To enable the usage of this technique, the system should provide an API to plug-in agents or should be in control of the developer to include this capability. This technique is suitable for asynchronous interactions, since they do not return a value to the caller. It also should be used when a function can be invoked by using several protocols, or the message exchanging is not the key point, but the expected effect in the remote node.

## Acknowledgements

We thank Jason Yip for being our shepherd. His help was fundamental during the development of this paper. We also thanks for the essential support of FAPESP through the project "*LEONA - Transient Luminous Event and Thunderstorm High Energy Emission Collaborative Network in Latin America*", nr 2012/20366-7.

## REFERENCES

BECK, K. (2003). Test-driven development: by example. Addison-Wesley, Boston

BERTOLINO, A., ANGELIS, G., FRANTZEN, L., POLINI, A., 2008. In: Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems: 8th International Workshop, Berlin

BESSON, F., MOURA, P., KON F., MILOJICIC D., 2012. Rehearsal: A framework for automated testing of web service choreographies. In: Proceedings of 3rd Brazilian Conference on Software: Theory and Practice (CBSoft), São Paulo

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M. , 1996, Pattern-Oriented Software Architecture Volume 1: A System of Patterns, Wiley.

CIRNE, W., BRASILEIRO, F., ANDRADE, N., COSTA, L., ANDRADE, A., NOVAES, R., AND MOWBRAY, M. 2006. Labs of the World, Unite!!! Journal of Grid Computing 4, 3, 225–246.

ERL, T., 2009, SOA Design Patterns, The Prentice Hall.

FARIAS, G., DANTAS, A., LOPES AND GUERRERO, D. 2012. Distributed Test Agents: A Pattern for the Development of Automatic System Tests for Distributed Applications. In: Proceedings of the 9th Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP), Natal, 2012.

FLORIANO, M. ; CHAMA, D. ; GUERRA, E. M. ; SILVEIRA, F. . The Annotated Test Step Pattern. In: 18TH CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, 2011, PORTLAND.

FLORIANO, M. ; CHAMA, D. ; GUERRA, E. M. ; SILVEIRA, F. . MakeATest: Um Framework para Construção de Anotações de Validação e Inicialização de Fatores Externos em Testes Automatizados. In: Brasilian Workshop on Systematic and Automated Software Testing, 2011, São Paulo.

FOWLER, M. 2004. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html.

FOWLER, M. 2011. IntegrationContractTest. http://martinfowler.com/bliki/IntegrationContractTest.html. last update at January 12th 2011

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J., 1994, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional.

HOFFMANN, A., Rennoch, A., Vouffo-Feudjio, A, Skljarski, S. 2003. Proxy-Based Component Testing with TTCN-3. In: International Conference Software & Systems Engineering and their Applications, 2003, Paris.

Intern, 2013, Using Intern to unit test Ajax calls. availabe at https://github.com/theintern/intern/wiki/Using-Intern-to-unit-test-Ajax-calls, last update at July 2013

JUnit, 2014, JUnit - A programmer-oriented testing framework for Java. available at http://junit.org

MakeATest RMI, available at https://github.com/FelipeOlivers/makeatest-rmi, last update at Sept 10 2011.

MESZAROS, G. 2007. xUnit Test Patterns: Refactoring Test Code, Addison-Wesley.

ROBISON, I. 2008. Consumer-Driven Contracts: A Service Evolution Pattern, at The Thoughtworks Anthology: Essays on Software Technology and Innovation, Pragmatic Bookshelf, 1 edition.

SmartBear, 2014. SoapUI - Automated testing of web service. Available at: http://www.soapui.org/

SOUZA, C. A., LACERDA, A. C., SILVA, J. W., PEREIRA, T. E., SOARES, A., AND BRASILEIRO, F. 2010. Beefs: Um sistema de arquivos distribuído posix barato e eficiente para redes locais. Simpósio Brasileiro de Redes de Compoutadores 2010 - Tools Session.

WALEND, D. 2008. JMX and Test-Driven Development. Available at https://weblogs.java.net/blog/2008/06/08/jmx-and-test-driven-development

YODER, J., WELICKI, L., WIRFS-BROCK, R. 2008.The dynamic factory pattern. Proceedings of the 15th conference on pattern languages and programming.