PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 04/2016

A Publish-Subscribe based approach for Testing Multi-Agent Systems

Nathalia Moraes do Nascimento
Carlos Juliano Moura Viana
Arndt von Staa
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 04/2016 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Dezembro, 2016

A Publish-Subscribe based approach for Testing Multi-Agent Systems

Nathalia Moraes do Nascimento, Carlos Juliano Moura Viana, Arndt von Staa, Carlos José Pereira de Lucena

nnascimento@inf.puc-rio.br, cviana@inf.puc-rio.br,
arndt@inf.puc-rio.br, lucena@inf.puc-rio.br

Abstract. Multi-agent systems (MASs) have been applied to several application domains, such as e-commerce, unmanned vehicles, and many others. In addition, a set of different techniques has been integrated into multi-agent applications. However, few of these applications have been commercially deployed and few of these techniques have been fully exploited by industrial applications. One reason is the lack of procedures guaranteeing that multi-agent systems would behave as desired. Most of the existing test approaches only test agents as single individuals and do not provide ways of inspecting the behavior of an agent as part of a group, and the behavior of the whole group of agents. Accordingly, we modeled and developed a publish-subscribe-based architecture to facilitate the implementation of systems to test MASs at the agent and group levels. To illustrate and evaluate the use of the proposed architecture, we developed an MAS-based application and performed functional and performance ad-hoc tests.

Keywords: integration test, group test, agent test, system test, multi-agent system, test architecture, publish-subscribe, RabbitMQ

Resumo. Sistemas Multiagentes (SMAs) vêm sendo utilizados em diferentes domínios, a exemplo de comércio eletrônico e veículo não tripulado. Além disso, diferentes técnicas vêm sendo integradas à aplicações multiagentes. Entretanto, poucas dessas aplicações foram comercialmente implantadas e poucas técnicas foram, de fato, exploradas por aplicações industriais. Um dos motivos é a falta de processos que possam garantir que sistemas multiagentes funcionem de acordo com o esperado. A maioria das abordagens existentes para testes apenas testam agentes como indivíduos independentes, e não provêem formas de inspecionar o comportamento de um agente como parte de um grupo ou o comportamento de todo o sistema multiagente. Dentro deste cenário, este trabalho propõe o uso de uma arquitetura baseada na tecnologia *publish-subscribe* para facilitar o desenvolvimento de sistemas que permitam testar não só indivíduos, como também, grupos de agentes. Com o intuito de ilustrar e avaliar o uso da arquitetura proposta para testar SMAs, foram realizados testes funcionais e de desempenho em uma aplicação multiagente.

Palavras-chave: teste de integração, teste de grupo, teste de agente, teste de sistema, sistema multiagente, arquitetura de teste, publish-subscribe, RabbitMQ

Responsável por publicações:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

Contents

1 Introduction **1**

2 TEST APPROACH: METHODS AND BACKGROUND **2**

2.1 Failure Diagnosis with Logs Containing Meta-Information Annotations 2

2.2 RabbitMQ: Publish-Subscribe Platform 3

3 APPLICATION SCENARIO **3**

3.1 Sellers and Clients 4

3.1.1 Sequence Diagrams 5

4 TEST APPROACH: THE SOLUTION ARCHITECTURE **6**

5 TESTS AND RESULTS **9**

5.1 Agent and Integration Tests 9

5.1.1 Test Results 13

5.2 System Testings 13

6 Conclusions and Open Challenges **15**

6.1 Testing Non-Deterministic Applications 15

6.2 Automated Testing Systems 15

6.3 Predictive Analysis 16

6.4 Deploying Agents in a Distributed Environment 16

1 Introduction

Multi-agent systems have been applied to a wide range of application types, including e-commerce, human-computer interfaces, network control, air traffic control and diagnosis [27] [36]. However, few of them have been commercially deployed [36]. According to Pěchouček and Mařík [36], one reason is the lack of procedures guaranteeing that the distributed systems would behave as desired. In addition, agent-based systems involve different characteristics, such as autonomy, asynchronous and social features, which makes these systems more difficult to understand. Thus, more elaborate methods of verification and testing of multi-agent operations should be provided [36].

Testing is the activity of evaluating software by observing its execution [2]. To perform this activity, it is necessary to create a test set, which is a set of test cases. For each test case, the tester specifies specific conditions to execute a system or component. Thus, he must raise the input values that are necessary to execute this system under test and the expected result. After executing a test case, the tester will evaluate the system, taking note of the observed or recorded results and compare them against specifications or expected results [29].

According to Nguyen et al. (2009) [32] and Moreno et al. (2009) [28], a full testing process of a multi-agent system consists of five levels: unit, agent, integration (or group), system (or society) and acceptance. In this paper, our goal is to address three of these testing levels: agent, integration and system. Agent test tests the capability of a specific agent to fulfill its goal and to sense and affect the environment. Integration test tests the interaction of agents and the interaction of agents with the environment, ensuring that a group of agents and environmental resources work correctly together [32] [22]. System test tests the quality properties that the intended system must reach, such as performance [32].

In the last years, several approaches have been proposed to test multi-agent systems at the unit and single agent levels [30] [7] [9] [35] [11] [31] [25] [1] [10] [26]. Otherwise, there are very few studies that address the issue of testing a MAS at group [30] [5] [18] [33] [37] [40] [20] and/or system levels [34]. In addition, to perform group tests, all approaches have focused on capturing and visualizing messages exchanged among agents. They do not provide ways of also tracking the behaviors of two or more agents in the same view and finding a correlation between their behaviors. For example, Serrano et al. (2012) [40], which is one of the most recent papers published about testing MASs at the group level, uses ACLAnalyser [5], a tool for debugging MAS through the analysis of ACL [15] messages. Thus, by using these current test approaches, if an agent exhibits unexpected behavior (failure), a developer has to inspect this failed agent or messages exchanged between agents to find the fault that caused that failure. However, if an agent fails, its failure may be related to a previous and an unexpected behavior of another agent in the environment.

In the general context of distributed systems, Araújo and Staa (2014) [12] also faced the problem of testing a group of asynchronous components. They realized that most approaches to detect error and diagnose a failure in distributed systems rely on distributed log files over various machines, which makes the comprehension of the interaction among the machines more difficult. Thus, Araújo and Staa (2014) [12] proposed the use of a central architecture to receive, store and inspect timestamp-based logs from distributed machines, enabling a developer to further diagnose failures in a single machine and in

the whole system during the software development cycle. However, the authors discuss some limitations in their approach, such as the query response time that grows as the database size grows, which impacts the inspection interface usage, and the use of an inefficient solution for creating discarding rules. Therefore, it is necessary to improve their architecture by integrating it with techniques to automatically classify the logs and discard the irrelevant ones.

In this paper, we present an architecture that was implemented[1] to make feasible the implementation of agent, integration and system test activities within the MAS software development process. Our approach is based on the architecture to test distributed systems proposed by Araújo and Staa [12]. Nonetheless, MASs involve some characteristics that are not addressed by non-agent-based systems, such as autonomy and social behaviors. Thus, our goal is to adapt the architecture proposed by Araújo and Staa [12] to create one for testing MASs. In addition, our architecture makes use of a publish-subscribe [19] technology in order to solve the problems of data volume and discarding rules described by Araújo and Staa [12]. Through a publish-subscribe based approach, it is also possible to develop decoupled and different tests that select logs that are useful for their purposes and ignore the irrelevant ones.

To illustrate and evaluate the use of the proposed architecture for testing MASs, we developed and tested a simple MAS-based application[2]. This experiment is presented in section 3. The remainder of this paper is organized as follows. Section 2 presents the background. Section 4 introduces the test architecture. Section 5 evaluates the test architecture, presenting the experimental results and evaluation. The paper ends with conclusive remarks in Section 6.

2 TEST APPROACH: METHODS AND BACKGROUND

As we discussed before, our goal is to adapt the architecture proposed by Araújo and Staa [12] to create one for testing MASs at different levels. To solve the problems of data volume and discarding rules presented in their architecture, we decided to use a publish-subscribe technology, instead of a database technology. Thus, subsection 2.2 presents RabbitMQ, which is a messaging broker.

As shown in subsection 2.1, Araújo and Staa [12] represent logs as tags that are key-value pairs. According to these authors, the most difficult step to use their test approach is how to identify the set of properties that must be represented as tags. Therefore, we decided to provide UML design artifacts in section 3 to derive test properties [6] and create tags. As multi-agent systems perform asynchronous behaviors, we used sequence diagrams to provide an overview of agents' interactions and behaviors.

2.1 Failure Diagnosis with Logs Containing Meta-Information Annotations

Araújo and Staa [12] investigated common approaches for testing distributed systems. According to these authors, there are several approaches that perform diagnosis based on

[1] The source of the test system is available at https://nathyecomp@bitbucket.org/nathyecomp/testingmasapps.git

[2] The source of the MAS application system is available at https://nathyecomp@bitbucket.org/nathyecomp/masapplications.git

log collection. Nonetheless, they have some limitations, such as the need of (i) organizing logs in a centralized architecture and in an adequate time order, (ii) providing visualization tools to assist manual inspection, and (iii) increasing the log details in order to enable the tool to also diagnose the application's logic. Therefore, they presented a diagnosing mechanism based on logs of events annotated with contextual information, allowing a specialized visualization tool to filter them according to the maintainer's needs.

In their approach, each logged event records a set of properties, represented as tags. A tag is a key-value pair where the value is optional. Every event must contain a basic set of tags which are: 1) *timestamp*: used to sort all events into a single timeline; 2)*message*: a description of the event; 3)*request id*: used to identify the type of event; 4) *device*: used to identify the device that originated the event; 5) *module*: the module that triggered the notification; and 6)*line*: the line of code where the notification command was inserted.

2.2 RabbitMQ: Publish-Subscribe Platform

RabbitMQ [38] is an intermediary for messaging, which generates asynchronous, decoupling applications by separating sending and receiving data through a client and scalable server architecture. It can be easily integrated into an application to operate as a common platform to send and receive messages, maintaining messages in a safe place to live until received. RabbitMQ is a multi-platform that may be deployed in Java, C, Python, and many other programming languages. It can also be deployed in a cloud infrastructure.

By using RabbitMQ, it is possible to build a logging system based on publish-subscribe architecture. The publisher is able to distribute log messages to many receivers, while the consumers have the possibility of selectively receiving the logs. Publisher and consumers communicate through queues. Each queue has a particular routing key that is a list of words, delimited by dots. There can be as many words in the routing key as you like, up to the limit of 255 bytes. These words can be anything, but usually they specify some features connected to the message. If the developer specify that a log message must meet the pattern "(month).(day).(deviceId).(typeLog)", valid routing keys would be "november.11.device01.error" and "november. 15.device01.info" [38].

Therefore, a message sent with a particular routing key will be delivered to all the queues that are bound with a matching binding key. However there are two important special cases for binding keys [38]:

* *(star)* can substitute for exactly one word.

\# *(hash)* can substitute for zero or more words.

By using queues, the publisher generates a set of information without the need of knowing which applications will consume them. In addition, more than one application can consume the same data, but giving them different treatments. To understand more about the characteristics of RabbitMQ that we used in our approach, see `https://www.rabbitmq.com/tutorials/tutorial-five-java.html` (Accessed in 11/2016).

3 APPLICATION SCENARIO

In order to evaluate our proposed approach to test multi-agent systems, we developed a simple multi-agent application. This application is based on a scenario commonly used in the MAS literature [8] [39] [3] - a simple marketplace to buy and sell books on-line. We

believe this experiment will assist one to understand our approach and facilitate further comparisons and analysis. We developed this application by using the JAVA Agent Development Framework (JADE) that is a Java software framework implemented to facilitate the development of multi-agent systems [42]. According to Pěchouček and Mařík [36], JADE is a leading open-source agent development environment on the market and some of the existing MAS applications and prototype systems use it. In addition, JADE implements the Foundation for Intelligent Physical Agents (FIPA) specifications that represent a collection of standards for the development of agent-based systems [15].

3.1 Sellers and Clients

This application implements a simple marketplace where users create autonomous agents to sell and buy books for them, as described in the JADE Guide [3]. Therefore, this scenario contains two kinds of agents: Seller and Client. As part of the JADE platform, there is also a Directory Facilitator Agent (DF) that provides a Yellow Pages service by means of which an agent can find other agents providing the services he requires [3]. This illustrative scenario is depicted in Figure 1.

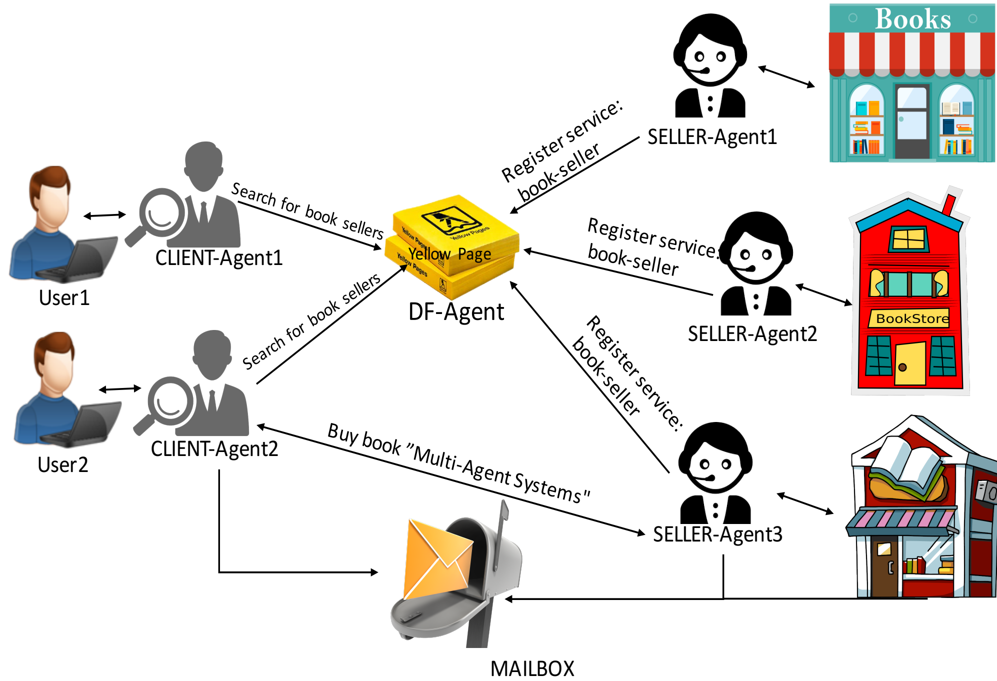

Figure 1: Scenario1: Overview of the general system architecture.

When a user creates a new selling agent, this agent registers itself in the Yellow Page by offering the service of book-seller. A selling agent manages a book catalog for a book store. Each user increments its own catalog at runtime by adding new books for sale. To add a book for sale, the user informs the name of the book and the price that he would like to receive for the book. A client agent is responsible for seeking and buying the book that a buyer user is looking for. Once created, the client agent is released into the marketplace, where it investigates which selling agents have the desired book and it buys the book from

the seller that has the best price.

We also added a mailbox to the application. Our goal is to simulate interactions between agents that are different from message communication. In such case, this interaction is performed by sharing a common resource among agents, that is, the mailbox. After selling the book, the seller agent sends a virtual copy of the book to the mailbox, while the client agent verifies if the book has delivered. If the client agent buys a book and it does not find the book in the mailbox after a time, the client agent will fail.

3.1.1 Sequence Diagrams

As shown in Figures 3 and 2, sending and receiving messages are activities represented as events in the sequence diagrams. For example, a SellerAgent (Figure 2) waits for two message types: "askPrice" and "buy." If it receives the first message type and finds the book in its catalogue, it will send a new message to the client to inform the price of the book. If it receives the "buy" message, the seller removes the book from its catalogue and sends it to the mailbox.

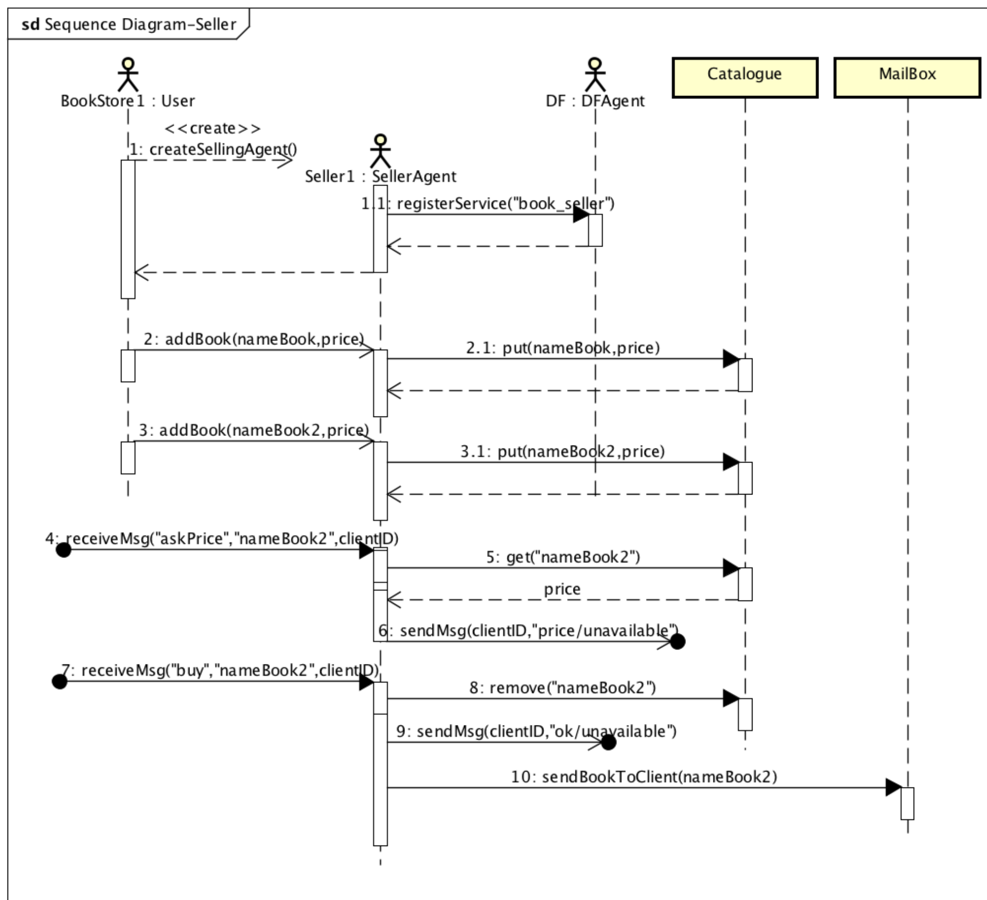

Figure 2: SellerAgent's interactions.

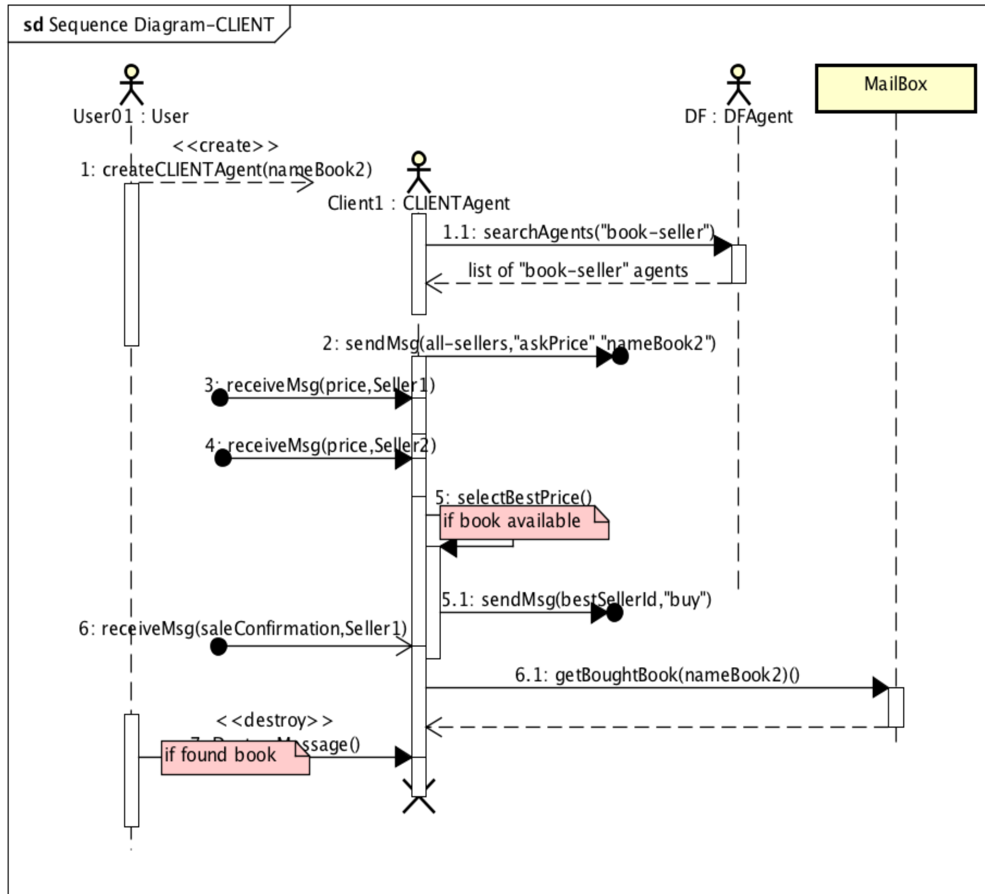

Figure 3: ClientAgent's interactions.

4 TEST APPROACH: THE SOLUTION ARCHITECTURE

We developed a publish-subscribe based architecture as a foundation for generating different kinds of test applications for MASs. Our goal is to provide mechanisms that capture and process logs generated by agents automatically. As depicted in Figure 4, our architecture consists of three layers: MAS Application (L1), Publish-Subscribe Communication (L2), and Test Applications (L3). The Publish-Subscribe Communication layer uses the RabbitMQ platform for delivering logs from agents (publishers) to be consumed by test applications (subscribers).

Each agent publishes logs with annotations that are composed of the following tags:

- *agentType*: the type of the agent (e.g CLIENT,

  SELLER, VEHICLE). In JADE, it refers to the name of the container where this agent lives;

- *agentName*: the name provided for the agent by the system developer/user (e.g client01, client02, seller01);

- *action*: the event that caused the log generation (e.g connectToSystem, searchBookInCatalogue,

  beDestroyed);

- *typeLog*: types of logs (e.g error, info, warning);

- *className*, *methodName*, *codeLine*: necessary information to identify the part of the code that generated the event;

- *resource*: the main resource that has been manipulated or requested by an agent during an event execution (e.g book1, book3, memory). It may be used to investigate all events that are related to a specific resource;

- *timestamp*: time that the log was created. Used to sort all events into a single timeline [12];

- *message*: a description of the event.

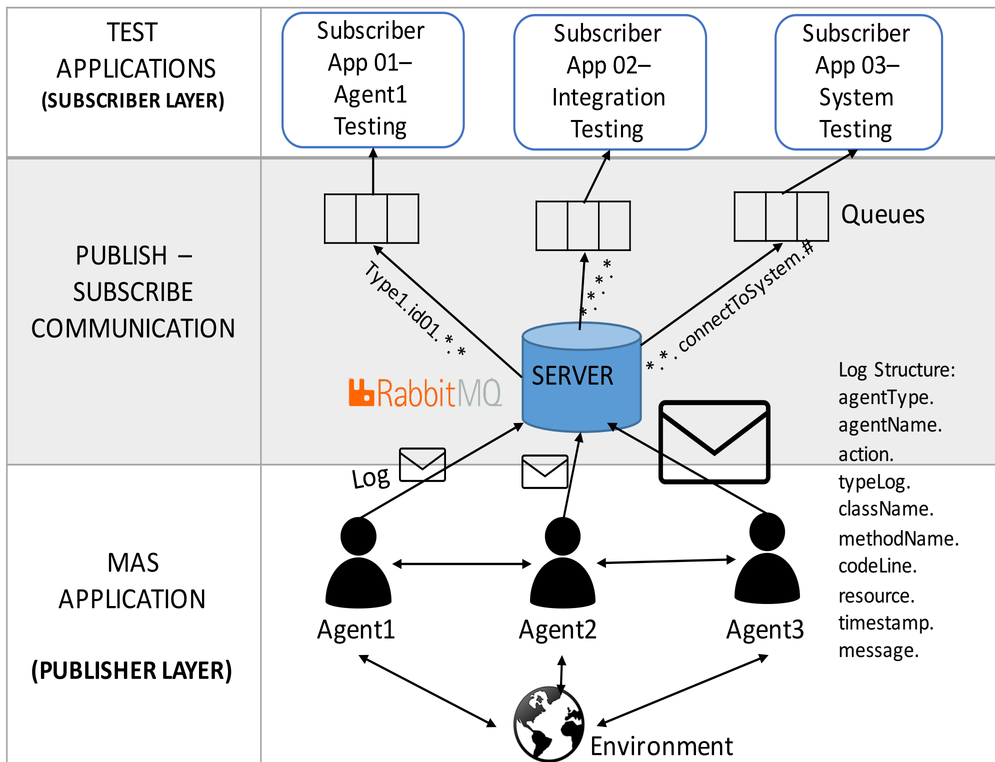

Figure 4: A Publish-Subscribe-based architecture to test MASs.

Thus, a log message must meet the pattern "(agentType).(agentName).(action).(typeLog). (className).(methodName).(codeLine).(resource).(timestamp).(message)." Each application will have a set of values that each tag may assume, excepting the message tag that is an open field.

As depicted in Figure 5, all agents in the MAS application layer are also a TestableAgent type. As a Testable agent extends the JADE agent, it complies FIPA specifications. A Testable agent uses the RabbitMQ properties to send logs with annotations as messages. The Testable agents will only publish logs if the variable testMode is true. These logs can be published from any part of the agent's code. Via the TestableAgent class and JADE properties, some tags have their values attributed autonomously, such as agentType, agentName and timestamp.

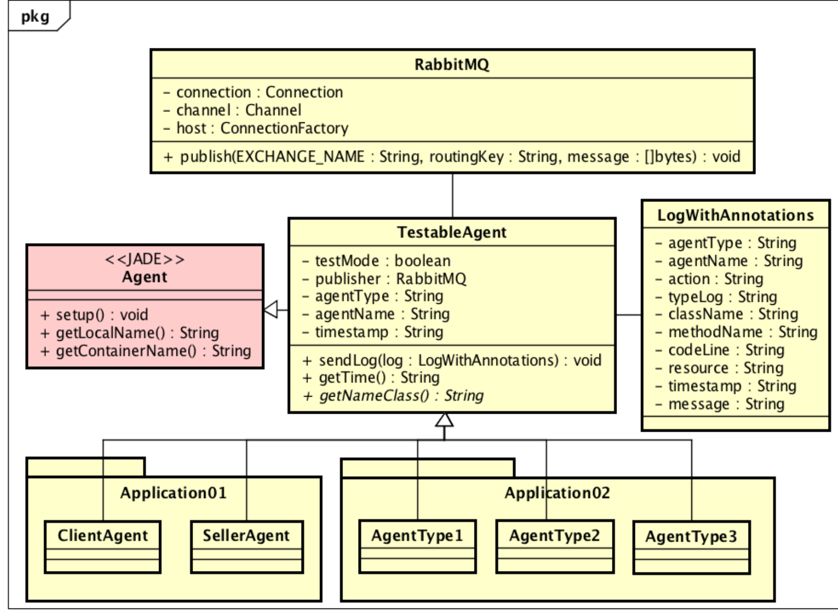

Figure 5: Simplified class diagram for creating testable MASs.

The RabbitMQ autonomously deliver log messages to queues according to their tags' values. Thus, each test application defines a binding key in order to subscribe itself to consume messages from a specific queue. For example, a test application that monitors only error logs from SELLER agents must have the binding key "SELLER.*.*.error.#." Therefore, this application will consume any log with the tuples (agentType,SELLER) and (typeLog,error). It is also possible to create applications that use multiple bindings. For example, if a performance test relates the number of agents in the system to the time that a DF agent spends to retrieve requisitions from CLIENT agents, this application will have to consume logs with different action values. First, this performance test needs to calculate the number of SELLER and CLIENT agents in execution. Thus, it needs to consume logs with the tuples (action,connectToSystem) and (action,beDestroyed). To calculate the time between requisition of CLIENT agents and retrieval from the DF agent, this test also has to consume the tuples (action,searchBook) and (action,receiveListFromDF) and extract timestamp information.

Test applications do not interfere on the execution of each other. As shown in Figure 6, each test class extends the class RabbitMQConsumer that starts an independent process to consume messages from a specific queue. We used the Template Method Pattern [16] to model the consumeMessage method. Thus, to consume and process particular log

messages, a test class must overwrite and customize the methods getListBindingKey() and processData().

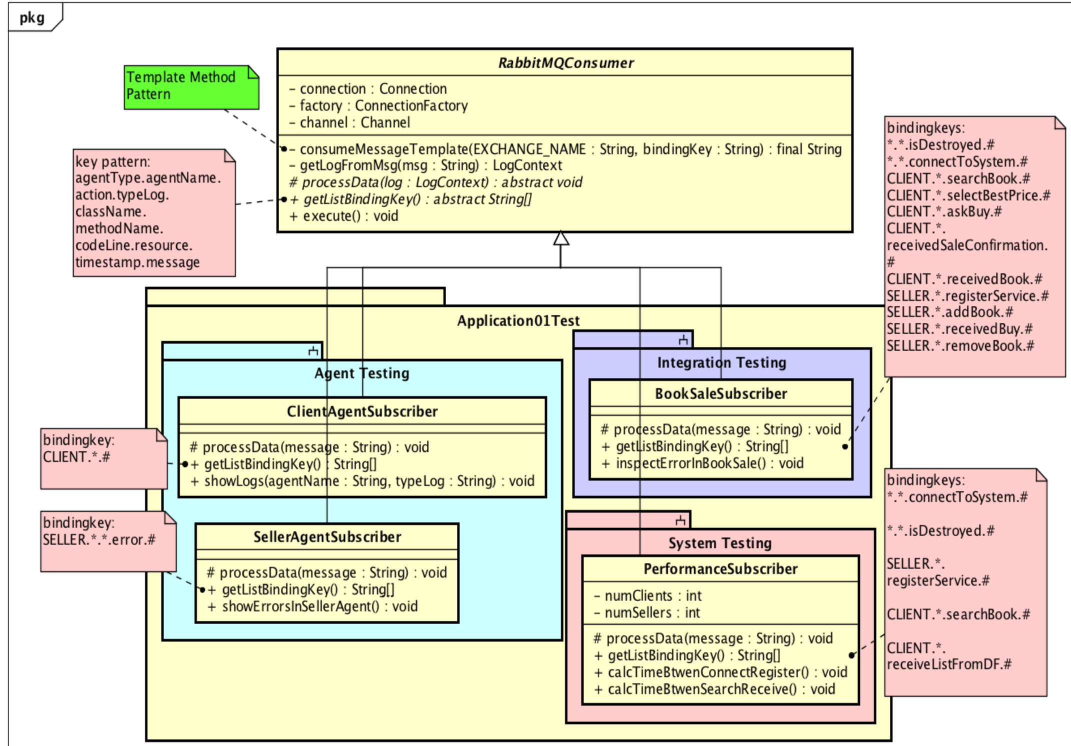

Figure 6: Simplified class diagram for creating applications for testing a MAS application at agent, integration and system levels.

5 TESTS AND RESULTS

By using our proposed architecture, we executed several functional ad-hoc tests at agent and integration levels. In addition, we also performed system tests, by evaluating performance characteristics of a MAS application. Thus, this section presents part of the test plan that we created and performed for testing the application presented in section 3.

5.1 Agent and Integration Tests

Table 1 presents functional tests at agent and integration (group) levels that we performed to test the book-sale application. More functional tests could be performed, but this list would be so extensive. We executed various test cases, taking eight parameters into account: (i) level (i.e. agent or group); (ii) function (i.e create CLIENT agent, add book, and any other function that was identified in sequence diagrams); (iii) procedure (e.g a general description of the test); (iv) previous condition; (v) input (i.e a resource, a component); (vi) actions (i.e actions to be performed during test execution); (vii) expected value (e.g the result that will be produced when executing the test if the program satisfies

its intended behavior [28]); and (viii) validation method (e.g strategies that a tester performs to evaluate the system, comparing the program execution against expected results). The execution of each test case produced several logs with meta-information annotations, which were consumed by test applications. Then, we used only these logs information as a validation method, as shown in table 1.

The function "buy book" is an example of behavior that would be difficult to inspect if the developer does not have a way of finding a correlation among agents' behaviors. If a client buys a book and does not receive it after a time, this agent will fail. Therefore, to analyze logs and understand what caused that failure, the developer must have the logs from the client and the seller sorted into a single timeline.

```
 [x] Sent 'SELLER-1.seller1.create.INFO.SellerAgent.initAgent.40.agent.2017.1.27.18.3.55.554.
 [x] Sent 'SELLER-1.seller1.connectToSystem.INFO.SellerAgent.setup.56.agent.2017.1.27.18.3.55.553.
Started Application 01
 [x] Sent 'SELLER-1.seller1.createCatalogue.INFO.SellerAgent.setup.55.catalogue.2017.1.27.18.3.55.752.
 [x] Sent 'SELLER-1.seller1.registerService.INFO.SellerAgent.registerService.71.yellowpage.2017.1.27.18.3.55.941.
 [x] Sent 'SELLER-1.seller1.receivedBudget.INFO.SellerAgent.receiveBudgetRequest.133.book.2017.1.27.18.5.52.948.client1
```

Figure 7: Logs generated by Seller1 during the execution of the test case 7 in table 1.

```
 [x] Sent 'SELLER-1.seller2.create.INFO.SellerAgentMutant.initAgent.46.agent.2017.1.27.9.6.42.476.
 [x] Sent 'SELLER-1.seller2.connectToSystem.INFO.SellerAgentMutant.setup.45.agent.2017.1.27.9.6.42.474.
Started Application 01
Adress>>::: 0
http://172.16.1.208:7778/acc
 [x] Sent 'SELLER-1.seller2.createCatalogue.INFO.SellerAgentMutant.setup.55.catalogue.2017.1.27.9.6.42.731.
 [x] Sent 'SELLER-1.seller2.registerService.INFO.SellerAgentMutant.registerService.71.yellowpage.2017.1.27.9.6.44.483.
 [x] Sent 'SELLER-1.seller2.waitClient.INFO.SellerAgentMutant.receiveBudgetRequest.133.book.2017.1.27.9.6.44.524.
 [x] Sent 'SELLER-1.seller2.addBook.INFO.SellerAgentMutant.addBookToCatalogue.109.book.2017.1.27.9.6.57.646.book1: 10
 [x] Sent 'SELLER-1.seller2.receivedBudget.INFO.SellerAgentMutant.receiveBudgetRequest.133.book.2017.1.27.9.8.42.333.client1
 [x] Sent 'SELLER-1.seller2.answeredBudget.INFO.SellerAgentMutant.receiveBudgetRequest.133.book.2017.1.27.9.8.42.349.client: client1 Title: book1 P
 [x] Sent 'SELLER-1.seller2.waitClient.INFO.SellerAgentMutant.receiveBudgetRequest.133.book.2017.1.27.9.8.42.371.
 [x] Sent 'SELLER-1.seller2.receivedBuy.INFO.SellerAgentMutant.receiveMsgToBuy.171.book.2017.1.27.9.8.42.468.client: client1 Title: book1 Price: 10
 [x] Sent 'SELLER-1.seller2.soldBook.INFO.SellerAgentMutant.receiveMsgToBuy.171.book.2017.1.27.9.8.42.482.client: client1 Title: book1 Price: 10
 [x] Sent 'SELLER-1.seller2.waitClient.INFO.SellerAgentMutant.receiveBudgetRequest.133.book.2017.1.27.9.8.42.539.
 [x] Sent 'SELLER-1.seller2.isDestroyed.INFO.SellerAgentMutant.takeDown.98.yellowpage.2017.1.27.9.8.43.493.
 [x] Sent 'SELLER-1.seller2.isDestroyed.ERROR.SellerAgentMutant.takeDown.98.yellowpage.2017.1.27.9.8.43.598.
```

Figure 8: Logs generated by Seller2 during the execution of the test case 7 in table 1.

```
 [x] Sent 'CLIENT.client1.create.INFO.ClientAgent.initAgent.54.agent.2017.1.27.9.7.41.954.
 [x] Sent 'CLIENT.client1.connectToSystem.INFO.ClientAgent.setup.41.agent.2017.1.27.9.7.41.952.
Started Application 01
 [x] Sent 'CLIENT.client1.searchBook.INFO.ClientAgent.setup.53.book.2017.1.27.9.8.42.218.book: book1
 [x] Sent 'CLIENT.client1.receiveListFromDF.INFO.ClientAgent.setup.73.yellowpage.2017.1.27.9.8.42.289.
 [x] Sent 'CLIENT.client1.askPrice.INFO.ClientAgent.buyBook.145.book.2017.1.27.9.8.42.329.ask book-price to 2sellers
 [x] Sent 'CLIENT.client1.receivedBudget.INFO.ClientAgent.buyBook.163.agent.2017.1.27.9.8.42.381.seller: seller2 price: 10
 [x] Sent 'CLIENT.client1.selectBestPrice.INFO.ClientAgent.buyBook.182.agent.2017.1.27.9.8.42.424.seller: seller2 price: 10
 [x] Sent 'CLIENT.client1.askBuy.INFO.ClientAgent.buyBook.145.agent.2017.1.27.9.8.42.468.seller: seller2 price: 10
 [x] Sent 'CLIENT.client1.receivedSaleConfirmation.INFO.ClientAgent.buyBook.180.book.2017.1.27.9.8.42.535.seller: seller2 price: 10
 [x] Sent 'CLIENT.client1.receivedBook.ERROR.ClientAgent.getBookInPostOffice.229.book.2017.1.27.9.8.46.627.book: book1
```

Figure 9: Logs generated by Client1 during the execution of the test case 7 in table 1.

In order to force test failure, we created a test case at the integration level by using a mutation based procedure. Adding program mutants is a common testing technique [24]. The goal of mutation test is to force certain classes to act incorrectly during the execution

Table 1: Functional tests at agent and integration (group) levels.

Level	Func.	Procedure	Previous Condition	Input	Actions	Expected Value	Validation Method (Logs sorted into a timeline)
Agent	create Selling Agent	User creates a new agent	DF is running	1.Type: SELLER 2. Name: seller1	1.User inits new agent and 2.DF registers agent in Yellow Page	seller1 agent is registered as book-seller	1)SELLER.seller1. connectToSystem.info.# 2)SELLER.seller1. registerService.info.#
		User tries to create an agent with the same name	seller1 is already running	1.Type: SELLER 2.Name: seller1	1.User inits new agent	Error: System refuses to create new agent	1)SELLER.seller1. connectToSystem. error.#
	add Book	User adds a new book to seller1	seller1 is already running	1.Book's name: book1 2.Book's price:10	1.User adds new book and 2.seller1 inserts book in its catalogue	book1 is in seller1's catalogue	1)SELLER.seller1. addBook.info.*.*.*.*. .'name:book1 and price:10'
	search Book	User searches book1	agent client1 is already running	1.Book's name: book1	1.User searches book1 and 2.client1 asks DF the list of book-sellers	DF returns the list of book-seller agents	1)CLIENT.client1. searchBook.info.# 2)CLIENT.client1. receiveListFromDF. info.#
		client1 waits DF's answer, but DF is dead	client1 has asked DF the book-sellers list	1.agent client1	1.client1 waits DF's answer and 2.DF dies	Error: client1 also dies	1)CLIENT.client1. searchBook.info.# 2)CLIENT.client1. receiveListFromDF. error.#
	buy Book	client1 asks seller1 to buy book1	seller1 has the book1	1.client1 2.seller1 3.book1	1.client1 asks book1 to seller1	client1 receives book1 after 2 seconds	1)CLIENT.client1. askBuy.info.# 2)CLIENT.client1. receivedBook.info.#
Group		client1 has asked sellerM to buy book1, but sellerM is dead now	sellerM has the book1	1.client1 2.mutant agent: sellerM 3.book1	1.client1 asks book1 to sellerM 2.sellerM receives the request 3. sellerM dies	Error: client1 does not receive book1 after two seconds	1)CLIENT.client1. askBuy.info.# 2)SELLER.sellerM. receivedBuy.# 3)CLIENT.client1. receivedSaleConf.# 4)SELLER.sellerM. isDestroyed.# 5)CLIENT.client1. receivedBook.error.#
		client1 and client2 have asked seller1 to buy book1	seller1 has the book1	1.client1 2.client2 3.seller1 4.book1	1.client1 asks book1 to seller1 2.client2 asks book1 to seller1 3.seller1 sells book1 to client1	Warning: client2 receives message of unavailable book	1)CLIENT.client1. askBuy.info.# 2)CLIENT.client2. askBuy.info.# 3)SELLER.seller1. receivedBuy.# 4)SELLER.seller1. removedBook.# 5)SELLER.seller1. sellBook.# 5)SELLER.seller1. receivedBuy.# 6)SELLER.seller1. refuseSell.# 7)CLIENT.client1. receivedBook.# 8)CLIENT.client2. receivedBook.warning.#
		client1 asks seller1 to buy book2, but it refuses the sale	seller1 has the book2, but it does not have price	1.client1 2.seller1 3.book2	1.client1 asks book2 to seller1 2.seller1 receives the request 3.seller1 refuses sale	Warning: seller1 does not sell an existing book and client1 receives msg of unavailable book	1)CLIENT.client1. askBuy.info.# 2)SELLER.seller1. receivedBuy.# 3)SELLER.seller1. refuseSell.# 4)SELLER.client1. receivedBook.warning.#

of the program over some tests in order to verify if the test application is able to identify faults [23]. Thus, we added a Seller agent mutant that dies after accepting a book sale. Therefore, a client agent that buys a book from the seller mutant will not find this book in the mailbox. Figures 7-9 depict the logs that were generated by agents while the test case that uses a Seller agent mutant (test case 7) was executing.

As shown in log messages, Client 1 received a sale confirmation from Seller 2, but he did not receive the book. This situation generated an error. As each agent is running separately, to identify and understand what caused this error, the tester would have to inspect the log file from each one of the agents. This work could be so difficult if the number of agents or the number of log messages was higher. Thus, by using our proposed solution, we can automatically select those logs from different agents that are probably to be essential for a specific test case and show them in a single interface.

In order to specify which characteristics need to be monitored from logs during the execution of a test cases set, the developer must establish a list of binding keys and override the method getListBindingKey() from the RabbitMQConsumer class. The list of binding keys will determine which test cases can be covered by the test application. For example, to cover the test cases "create Selling Agent" (line 1), "add Book" (line 3) and "buy Book" (line 7), which are listed in Table 1, it is necessary to establish binding keys that will make the test application able to receive all logs that are described in the "Validation Method" column from the line 1, 3 or 7.

The code below shows the method getListBindingKey() that was used by a test application to cover these three test cases. If the developer wants this test application covering more test cases, he needs to add more binding keys to allow the test application to consume different logs. For example, to cover test cases of the functionality "search Book", the developer must add a new binding key that contains the "receiveListFromDF" action.

```
@Override
public String[] getListBindingKey() {
BindingKey bd = new BindingKey();
String[] listKey = new String[11];
listKey[0] = bd.createBindingKey(LogValue.Action.isDestroyed);
listKey[1] = bd.createBindingKey(LogValue.Action.connectToSystem);
listKey[2] = bd.createBindingKey(LogValue.AgentType.CLIENT, LogValue.Action.searchBook);
listKey[3] = bd.createBindingKey(LogValue.AgentType.CLIENT, LogValue.Action.selectBestPrice);
listKey[4] = bd.createBindingKey(LogValue.AgentType.CLIENT, LogValue.Action.askBuy);
listKey[5] = bd.createBindingKey(LogValue.AgentType.CLIENT, LogValue.Action.receivedSaleConfirmation);
listKey[6] = bd.createBindingKey(LogValue.AgentType.CLIENT, LogValue.Action.receivedBook);
listKey[7] = bd.createBindingKey(LogValue.AgentType.SELLER, LogValue.Action.registerService);
listKey[8] = bd.createBindingKey(LogValue.AgentType.SELLER, LogValue.Action.addBook);
listKey[9] = bd.createBindingKey(LogValue.AgentType.SELLER, LogValue.Action.receivedBuy);
listKey[10] = bd.createBindingKey(LogValue.AgentType.SELLER, LogValue.Action.removeBook);
return listKey;
}
```

As a result, the interface depicted in Figure 10 shows all logs that were selected according to this binding key list. As shown, not all logs depicted in Figures 7-9 were presented in this interface, but only the relevant logs to validate the test cases 1, 3 and 7. In addition, all logs are organized in a single timeline. Therefore, to affirm that the expected value for the execution of each one of these test cases is satisfied, it will be necessary to verify if logs are appearing in this interface as described in the "Validation Method" column from Table 1. For example, to validate the test case 1, it is necessary to verify if its expected value was reached, which is "seller1 agent is registered as book-seller." According to Table 1, to validate it, we need to find the following logs: "1) SELLER.seller1. connectToSystem.info.#" and 2)"SELLER.seller1.registerService.info.#."

In general, Figure 10 illustrates the main interface of agent and integration test ap-

plications. By using this interface, it is possible to inspect logs that were consumed by a test application and verify if these logs match the logs listed in the validation method column. As shown in application view, the client agent failed after buying a book from a seller agent mutant, which also satisfies the test specification set in Table 1 for the “buy book” function.

INTEGRATION LOGS

Agent Type	Agent Name	Class	Log Type	Action	Resource	Timestamp(ms)	Message
SELLER	seller1	SellerAgent	INFO	connectToSystem	agent	7.9882357E7	
SELLER	seller2	SellerAgentMutant	INFO	connectToSystem	agent	7.9882632E7	
CLIENT	client1	ClientAgent	INFO	connectToSystem	agent	7.988274E7	
SELLER	seller1	SellerAgent	INFO	registerService	yellowpage	7.9882874E7	
SELLER	seller2	SellerAgentMutant	INFO	registerService	yellowpage	7.9884354E7	
SELLER	seller2	SellerAgentMutant	INFO	addBook	book	7.9912699E7	book1: 10
CLIENT	client1	ClientAgent	INFO	searchBook	book	7.9942776E7	book: book1
CLIENT	client1	ClientAgent	INFO	selectBestPrice	agent	7.994286E7	seller: seller2 price: 10
CLIENT	client1	ClientAgent	INFO	askBuy	agent	7.9942873E7	seller: seller2 price: 10
SELLER	seller2	SellerAgentMutant	INFO	receivedBuy	book	7.9942881E7	client: client1 Title: book1 Price: 10
CLIENT	client1	ClientAgent	INFO	receivedSaleConfir...	book	7.9942946E7	seller: seller2 price: 10
SELLER	seller2	SellerAgentMutant	INFO	isDestroyed	yellowpage	7.9945716E7	
SELLER	seller2	SellerAgentMutant	ERROR	isDestroyed	yellowpage	7.9945808E7	
CLIENT	client1	ClientAgent	ERROR	receivedBook	book	7.9946989E7	book: book1

Figure 10: Application View - Agent and Integration tests. Fault detection in test case 7 execution.

5.1.1 Test Results

As shown in Table 1, we executed some functional tests at agent and group levels. By using an interface (Figure 10), we were able to validate these test cases by comparing the logs observed in this main interface against the logs listed in the “Validation Method” column. In addition, we also conducted some tests by inserting software failures and verifying if our test software could be useful for detecting faults. As shown in Figure 10, we were able to identify an error and track the behaviors that were executed by agents before this error.

5.2 System Testings

There are different tests that can be performed to evaluate quality properties of a system, such as fault tolerance and performance. In this paper, we covered only performance tests. Figure 11 depicts some load tests that were executed in order to give out the response time of the most important operation: buy a book. To perform these tests, the MAS and test applications were running at the following operating environment:

- MacBook Pro (Retina, 13-inch, Early 2015)
- Processor 2.7 GHz Intel Core i5
- Memory 8 GB 1867 MHz DDR3

Our goal is to verify the maximum number of selling agents that the book sale application supports. In addition, we aim to answer the following question: If a customer specifies the response time limit of book sale, which will be the limit of seller agents that can be registered in the system? For this purpose, we created an application that implements

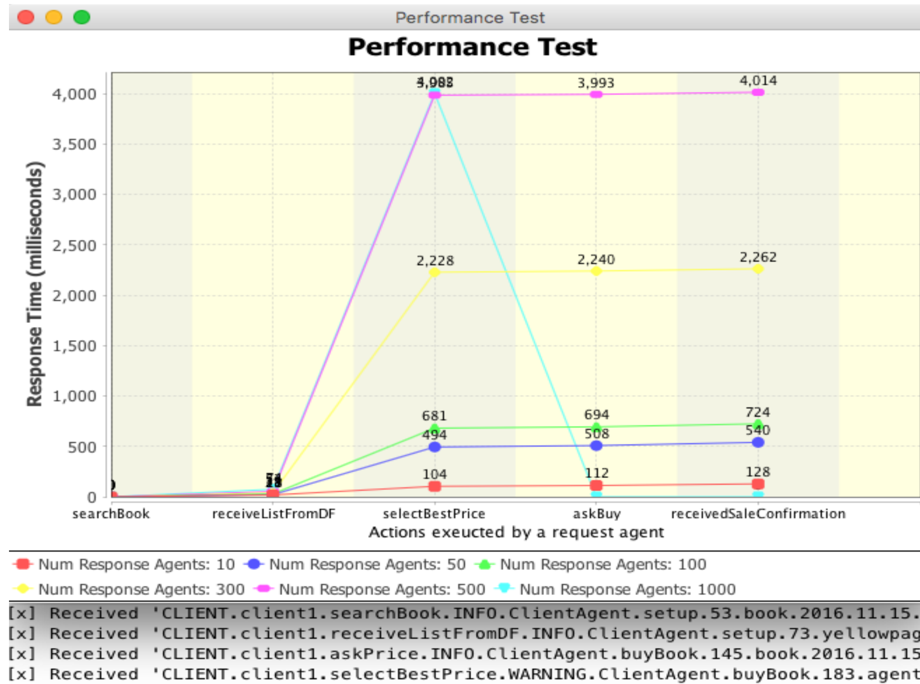

Figure 11: Application View - Performance test.

the getListBindingKey method to consume messages from a client agent and to verify the number of agents that are connected to the system, as follows:

```
@Override
public String[] getListBindingKey() {
BindingKey bd = new BindingKey();
String[] listKey = new String[4];
listKey[0] = bd.createBindingKey(LogSystem.Action.connectToSystem, LogSystem.TypeLog.INFO);
listKey[1] = bd.createBindingKey(LogSystem.Action.isDestroyed);
listKey[2] = bd.createBindingKey(LogSystem.AgentType.CLIENT, LogSystem.TypeLog.INFO);
listKey[3] = bd.createBindingKey(LogSystem.Action.selectBestPrice, LogSystem.TypeLog.WARNING);
return listKey;
}
```

While running this test application, we executed MAS applications in sequence with different numbers of selling agents, as shown in Figure 11. All applications have only one client agent, which is looking for the book1. In addition, only the selling agent that was the last one to start has the book1.

According to the results shown in Figure 11, we can affirm that if a customer specifies that 540 milliseconds is the limit time for a client receives a sale confirmation after starts to search a registered book, this system cannot allow more than 50 selling agents to register.

We also observed that when the application has more than 100 selling agents, it starts to be very unstable. The reason is the limit of agents that can be indexed by the DF-Service.search [41] from JADE. Thus, the DF agent randomly selects the selling agents to receive the book-search request from the client agent. However, it is possible to change this limit. According to our operating environment, we got to change this limit to 500 agents. We executed a MAS application with a number of selling agents higher than the limit of 500 agents, as shown in Figure 11. For instance, the client agent went into a loop state, which took the test to be aborted. Therefore, we can conclude that if the system

operates at this same environment, the maximum number of selling agents will be 500.

6 Conclusions and Open Challenges

We believe these preliminary results are promising. We presented a decoupled architecture that allows a developer to execute tests simultaneously and independently while running a MAS. In addition, we provided evidence of the usability of our proposal, using it to test a known MAS application. We showed that it is possible to develop different tests for a multi-agent system at different levels by using logs containing meta-information annotations and a publish-subscribe technology.

6.1 Testing Non-Deterministic Applications

However, MASs usually are much more complex than the experiment that was used in this work. Current approaches modeled by using a MAS may involve non-deterministic characteristics that were not addressed by this paper, such as *learning* [14], *self-adaptation* and *self-organization* (SASO) [13]. In fact, there is a gap in the literature regarding the test of systems with these features. There are few approaches to inspect the emergence process in a self-organizing MAS system [17] [4], and all of them do MAS design based only on simulation techniques. One reason is the difficulty of specifying expected results for non-deterministic applications, especially in actual environments. Nonetheless, we believe our approach opens the way for more experiments in testing Multi-Agent Systems, since it provides ways for testing a MAS at different levels. For example, as a self-organizing MAS system enables the emergence of social features based on the behavior of individual agents, to test this kind of system it is necessary to perform tests at single and group levels. Therefore, another goal is to evaluate the use of this publish-subscribe architecture to test more complex MAS applications, such as self-organizing and self-adaptive systems.

Nascimento and Lucena (2016) [13] presented a framework to create self-organized and self-adaptive agent-based applications. They evaluate their architecture by deriving two instances from their framework (a simulated and a realistic one). However, since they do not provide test beds in small, medium and large scales, it is not possible to verify the quality of the applications generated by using their framework. Our next contribution is to generate a SASO application by using their framework and show that these kinds of applications can be tested by using our test approach.

6.2 Automated Testing Systems

We aim to improve our architecture to allow the tester to also interact with the application. For this purpose, each layer to be publisher and subscriber needs to be modeled. For example, the MAS application would receive messages from the performance test application in order to initiate new agents and execute other actions. Thus, the performance test app may execute and evaluate the system autonomously. We can further consider relating the TestableAgent class with a singleton class [16] that contains a set of variables to be set by test applications. For example, the boolean variable “testMode” in the class TestableAgent, which enables the system to publish logs, should be placed in this common class. Thus, test applications could manage this variable in order to begin or pause

tests. As all agents extend the TestableAgent in our architecture, all agents will access this common class and will be accessed by test applications.

6.3 Predictive Analysis

For each application, there is a set of predetermined values that can be used in tags. Thus, we can codify and normalize these values to use them as inputs of a temporal neural network [21], which is a known structure of predictive analysis. By consuming temporal logs, a test application may use a temporal neural network to process log information in order to predict errors.

6.4 Deploying Agents in a Distributed Environment

For instance, all agents are running on a single machine and the proposed approach assumes that there is a central clock so that the timestamp in each message can be used to sort events and measure throughput. If the tester needs to evaluate the MAS application in a distributed environment, he must adapt the RabbitMQ class to use the Remote Procedure Call (RPC) function, which is a technology supported by RabbitMQ to build a scalable RPC server [38]. Thus, he can create a common publisher that publishes logs from agents localized on different machines. However, new challenges will arise. In the general case of distributed agents, possibly running on machines in different continents, a synchronization mechanism is required, especially for performance test.

Acknowledgments

This work has been supported by the Laboratory of Software Engineering (LES) at PUC-Rio. Our thanks to CNPq, CAPES, FAPERJ and PUC-Rio for their support through scholarships and fellowships.

References

[1] Y. Abushark, J. Thangarajah, T. Miller, J. Harland, and M. Winikoff. Early detection of design faults relative to requirement specifications in agent-based models. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 1071–1079. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[2] P. Ammann and J. Offutt. *Introduction to software testing.* Cambridge University Press, 2008.

[3] F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, and R. Mungenast. Jade administrator's guide. *TILab (February 2006)*, 2003.

[4] C. Bernon, M.-P. Gleizes, and G. Picard. Enhancing self-organising emergent systems design with simulation. In *International Workshop on Engineering Societies in the Agents World*, pages 284–299. Springer, 2006.

[5] J. Botía, A. Lopez-Acosta, and A. Skarmeta. Aclanalyser: A tool for debugging multi-agent systems. 2004.

[6] L. Briand and Y. Labiche. A uml-based approach to system testing. In *International Conference on the Unified Modeling Language*, pages 194–208. Springer, 2001.

[7] G. Caire, M. Cossentino, A. Negri, A. Poggi, and P. Turci. *Multi-agent systems implementation and testing.* na, 2004.

[8] A. Chavez and P. Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proceedings of the first international conference on the practical application of intelligent agents and multi-agent technology*, volume 31, page 40. London, UK, 1996.

[9] R. Coelho, E. Cirilo, U. Kulesza, A. von Staa, A. Rashid, and C. Lucena. Jat: A test automation framework for multi-agent systems. In *2007 IEEE International Conference on Software Maintenance*, pages 425–434. IEEE, 2007.

[10] F. Cunha, A. D. da Costa, M. Viana, and C. J. P. de Lucena. Jat4bdi: An aspect-based approach for testing bdi agents. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2015 IEEE/WIC/ACM International Conference on*, volume 2, pages 186–189. IEEE, 2015.

[11] A. D. da Costa, C. Nunes, V. T. da Silva, B. Fonseca, and C. J. de Lucena. Jaaf+ t: a framework to implement self-adaptive agents that apply self-test. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 928–935. ACM, 2010.

[12] T. P. de Araújo and A. von Staa. Supporting failure diagnosis with logs containing meta-information annotations. *Technical Reports in Computer Science. ISSN 0103-9741*, 14:21, 2014.

[13] N. M. do Nascimento and C. J. P. de Lucena. Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things. *Information Sciences*, 2016.

[14] N. M. do Nascimento, C. Jos, P. de Lucena, and H. Fuks. Modeling quantified things using a multi-agent system. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 1, pages 26–32. IEEE, 2015.

[15] FIPA. The foundation for intelligent physical agents. http://www.fipa.org/, 10 2016.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

[17] L. Gardelli, M. Viroli, and A. Omicini. On the role of simulation in the engineering of self-organising systems: Detecting abnormal behaviour in mas. 2005.

[18] J. J. Gomez-Sanz, J. Botía, E. Serrano, and J. Pavón. Testing and debugging of mas interactions with ingenias. In *International Workshop on Agent-Oriented Software Engineering*, pages 199–212. Springer, 2008.

[19] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273. Springer-Verlag New York, Inc., 2004.

[20] C. Gutiérrez, I. García-Magariño, E. Serrano, and J. A. Botía. Robust design of multi-agent system interactions: A testing approach based on pattern matching. *Engineering Applications of Artificial Intelligence*, 26(9):2093–2104, 2013.

[21] S. Haykin. *Neural Networks: A Comprehensive Foundation.* Macmillan, 1994.

[22] Z. Houhamdi. Multi-agent system testing: A survey. *International Journal of Advanced Computer*, 2011.

[23] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, (4):371–379, 1982.

[24] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[25] V. J. Koeman and K. V. Hindriks. Designing a source-level debugger for cognitive agent programs. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 335–350. Springer, 2015.

[26] V. J. Koeman, K. V. Hindriks, and C. M. Jonker. Automating failure detection in cognitive agent programs. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 1237–1246. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

[27] C. Lucena. *Software engineering for multi-agent systems II: research issues and practical applications*, volume 2. Springer Science & Business Media, 2004.

[28] M. Moreno, J. Pavón, and A. Rosete. Testing in agent oriented methodologies. In *International Work-Conference on Artificial Neural Networks*, pages 138–145. Springer, 2009.

[29] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[30] D. T. Ndumu, H. S. Nwana, L. C. Lee, and J. C. Collis. Visualising and debugging distributed multi-agent systems. In *Proceedings of the third annual conference on Autonomous Agents*, pages 326–333. ACM, 1999.

[31] C. D. Nguyen, S. Miles, A. Perini, P. Tonella, M. Harman, and M. Luck. Evolutionary testing of autonomous software agents. *Autonomous Agents and Multi-Agent Systems*, 25(2):260–283, 2012.

[32] C. D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah. Testing in multi-agent systems. In *International Workshop on Agent-Oriented Software Engineering*, pages 180–190. Springer, 2009.

[33] C. D. Nguyen, A. Perini, and P. Tonella. Ontology-based test generation for multiagent systems. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pages 1315–1320. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[34] C. D. Nguyen, A. Perini, and P. Tonella. Goal-oriented testing for mass. *International Journal of Agent-Oriented Software Engineering*, 4(1):79–109, 2009.

[35] D. C. Nguyen, A. Perini, and P. Tonella. A goal-oriented software testing methodology. In *International Workshop on Agent-Oriented Software Engineering*, pages 58–72. Springer, 2007.

[36] M. Pěchouček and V. Mařík. Industrial deployment of multi-agent technologies: review and selected case studies. *Autonomous Agents and Multi-Agent Systems*, 17(3):397–431, 2008.

[37] W. Peng, W. Krueger, A. Grushin, P. Carlos, V. Manikonda, and M. Santos. Graph-based methods for the analysis of large-scale multiagent systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 545–552. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[38] RabbitMQ. Rabbitmq. Available in https://www.rabbitmq.com/, 10 2016.

[39] J. Sabater and C. Sierra. Reputation and social network analysis in multi-agent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 475–482. ACM, 2002.

[40] E. Serrano, A. Muñoz, and J. Botia. An approach to debug interactions in multi-agent system software tests. *Information Sciences*, 205:38–57, 2012.

[41] Telecom. Class dfservice. Available in http://jade.tilab.com/doc/api/jade/domain/DFService.html, 11 2016.

[42] Telecom. Java agent development framework. Available in http://jade.tilab.com/, 10 2016.