Reliable Multi-agent System for a large scale distributed energy trading network

Marcel Koster

Faculty of Mathematics and Natural sciences

Rijksuniversiteit Groningen

Master Degree Thesis

2010/2011

Contents

1 Introduction

Electric energy is a vital part of our lives nowadays, in contrast to a few hundred years back. It had no uses back then and was only produced by nature. Today, energy is used for many applications, such as lights, factories and household appliances. The production of energy by man made machines has been around since 1831, when the first electric generator was invented by Michael Faraday. Since then a lot of things have changed, the generators now have a much larger capacity, there are many more around and they are often placed in energy production facilities, which are scattered over a country. The amount of people that use this energy has also increased dramatically. This has resulted in an electricity grid being built and still being expanded, in order to connect the people to the energy production facilities. Because the basic electricity grid and the first electricity production facilities have been pondered and built during the previous century, the whole system is still suffering from an outdated energy distribution model.

This outdated model is based on a monopolistic view of the energy distribution and is still widely used by most countries. In this view, every consumer is restricted to a very limited amount of energy production companies, called Gencos, that are present in his country. This amount of Gencos is limited by the government. Because of this limited amount of Gencos, the competition between these Gencos is very low, resulting in limited or no innovation towards a more liberal or decentralized system and often high prices for the consumers. This has changed in the last decade, as the market has been opened by several governments and has allowed an expansion of the amount of Gencos in a country. The market is still heavily regulated by the government, but soft market competition is possible and more common. This has resulted in a more open market where consumers can choose a Genco that suits their needs.

Another change from the last two decades, is the decoupling of the production sector (electricity production facilities), controlled by the Gencos, from the distribution sector (the electricity grid), controlled by the new transmission service operators, or TSOs.

In the outdated model there was also little concern over the environmental issues and energy sources. The energy production was very dependent on raw energy materials, such as oil, gas and coal. This has proven to be problematic, as the materials got more expensive, due to political or economical issues in some of the material exporting countries. Fortunately, in the last decade this has slightly changed due to scientific studies and new government rules, and Gencos are now more keen on “greener” energy and are using different renewable energy sources, such as wind and solar power.

As has been shown, the outdated model had no stimulus for innovation, was expensive, left pollution issues untreated and even left the consumer economically dependent on other countries, due to the dependence on raw materials. This has changed in the last decades as shown, and has resulted in a slightly more open energy market. However, the biggest change is coming from the consumers themselves, who are now installing their own solar panels and wind turbines since the last decade. They are now capable of producing their own energy from renewable resources and can even sell back their surplus energy to their Genco via the electricity grid.

Despite these recent changes the current model still needs a lot of improvements. As more consumers are selling back their surplus energy to their Genco, the grid lines may become overloaded. This requires more intensive energy checks and possibly an expanded grid. Also, the energy production is still damaging the environment and is still largely dependent on the economical issues of the raw material distributing countries. Finally the amount of consumers that get their energy from “greener” and renewable sources is still very limited.

This can all be improved by extending the functionalities of the TSOs, by providing their services to consumers that have their own renewable energy devices, such as solar panels or wind turbines. These consumers can then become producers themselves and form a new actor on the field of the energy market, called “Prosumers”. This will give the regular consumers the possibility to get their energy directly from these Prosumers without any intervention from a Genco. In this case the Prosumers can supply their own energy needs, as well as the energy needs of a few other consumers. This will create a new free energy market with Prosumers and Gencos that offer the same service, which is much more decentralized and where the consumers can decide where they get their energy and at which price. Prosumers can use weather forecasts to manage their solar panels and/or wind turbines production and adjust prices according to the market trends and raw material prices.

1.1 Research context

The research group of this master thesis is Distributed Systems. The Distributed Systems group performs research and delivers education in all aspects of distributed information systems with particular emphasis on Service-Oriented Computing, Pervasive middle-ware and Energy distribution infrastructures. This master thesis contributes to a project from this research group, named SPARC.

SPARC stands for: Smart metering Peer-to-peer ARChitectures. Prof. Dr. Ir. Marco Aiello is the Principal Investigator for this project. The SPARC project focuses on the energy grid. Currently this is a monopolistic system, with only a few large suppliers. The idea is to change this to a democratic energy exchange. The aim of the project is to study the topology of the current power grid and to identify necessary changes in order to enable a peer-to-peer energy exchange. The vision is that of future residential units that not only consume energy, but also store and generate energy which they can sell to neighboring nodes.

1.2 Problem statement

To move to a more open and democratic energy market, certain changes have to made to the current market. There is very little knowledge of how to properly design a retail electricity market and how to effectively incorporate other services. The energy distribution service requires quality improvements for the new market to function correctly, because of the higher granularity of the energy contracts. This work can not be done by humans, because of this increased granularity they would need to handle a huge amount of operations in the system. Also, searching for the best sellers in the new market with millions of suppliers should be done autonomous.

To be able to identify the necessary changes, a flexible simulation platform should be created of the new democratic energy market. This simulation can then be used to give indications on electricity grid bottlenecks and topological problems and identify market problems, such as an excess amount of Consumers, Gencos, or Prosumers. It can also be used to test new market designs before their implementation. The simulation platform has to meet some demands, in order to be able to simulate the new energy market effectively. It must be able to test different market designs, structural conditions of the energy grid and adaptive behaviors of the Consumers and their impact on short-term and long-term market performance. It should also be flexible and able to simulate Consumers, Gencos and Prosumers, which can sell and buy energy from each other. And it should be possible to expand the system later on.

A simple system, such as a web service or a web application, is not capable of handling this large and complex problem. This problem requires a more autonomous, adaptive and independent system

as a basis. This will prevent the requirement for human operators and leave only a few human controlled adjustments in the system, such as initial variable values and other preferences. A Multi-agent system is an ideal candidate for this kind of simulation, as they are often used for auction, market and economic simulations and other complex adaptive systems. A Multi-agent system consists of multiple interacting intelligent agents. These agents are autonomous, proactive, reactive and can therefore emulate the behaviors of heterogeneous market participants, such as the Consumers, Gencos and Prosumers. A Multi-agent implementation of this peer-to-peer energy exchange was already created by N. Capodieci and is used as a basis for this research. However, the reliability and scalability factors of this system have not yet been taken into account [17], and require improvement to support future expansions, fault tolerant and more realistic simulations.

1.3 Research focus and questions

This research contributes to the system by N. Capodieci, by focusing on the scalability and reliability factors that require improvement. The system must be scalable, because of the amount of peers in an energy network, which could range to a few million. To make sure that a realistic simulation can be performed, the system must therefore support at least a 1000 agents and support future expansions. The reliability is also very important, because the system needs to be able to run successful simulations. This is especially important if the simulations take more than a few minutes, in which case the chances of failure increase. The base system needs to be reliable, in order to support the future expansions of the system. Reliability includes fault-tolerance here, as faults may still occur in an expanded system, but the core system itself needs to stay functional.

If all these problems can be solved, the system can form a solid basis for future expansions on strategy learning algorithms, more agent types, realistic weather forecasts and even real world topology settings. The platform used for this system is JADE, which is one of the few frequently updated and highly used platforms for Multi-agent systems. This will ensure a system that is still usable in the future.

From the problem statement and the focus of this research, some essential research questions are created. These questions are used as a guideline throughout the research and are answered in section 7. Four main questions, each having two sub-questions are devised:

- How to support the large scale of this system?
    - Which techniques have already been applied to other large scale MAS, to increase scalability?
    - What are the advantages and disadvantages of these techniques and are they suitable for our MAS?
- How to make sure that the system is reliable?
    - Which techniques have already been applied to other large scale MAS, to increase reliability and fault-tolerance?
    - What are the advantages and disadvantages of these techniques and are they suitable for our MAS?
- How to implement this system?
    - How have other MAS been implemented?
    - How to implement the techniques which are suitable for our MAS?
- How to evaluate such a system?

- How have other MAS been evaluated?
- How to evaluate the techniques and ensure correct functionality of the system?

The research involved a literature study and an implementation. The literature study focuses on research that has already been done in the field of agent systems in relation to scalability and reliability. The implementation of this Multi-agent system uses N. Capodieci's work and expands it by adding scalability, reliability and fault-tolerance solutions.

1.4 Thesis layout

The next chapter is the State of the art, the current state of research in the field of Multi-agent system and energy trading simulations is explained in this chapter. Chapter 3 gives background knowledge on JADE and on the methods that are used to improve the scalability and reliability of the system. Chapter 4 explains the improvements and the architecture of the MAS in detail. This includes the system structure, agent organization, agent behavior and interaction. The 5th chapter focuses on the implementation of the MAS. Each of the classes in the system is discussed. Chapter 6 discusses the test cases and results, as well as the GUI and evaluation strategies. The last chapter is a discussion on the results of this research and on future work.

2 State of the art

The focus of this research is on Multi-agent systems for energy market simulations. This is a highly active field of research, where a lot has already been written on the subject. A short explanation on the state of the art in Multi-agent systems is provided in the next section. Specific systems for energy market simulations are discussed in the following section. The MAS created by N. Capodieci is discussed last.

2.1 Multi-agent systems

A Multi-agent system is a system that consists of several agents that interact with each other. These interactions are often handled by messages that are sent between the agents. These agents simulate intelligence by using methodical, functional, procedural or algorithmic search, find and processing approaches. Each of the agents can have different goals and behaviors, which together combines to a dynamic system. The agents have some critical features according to [18]: they are at least partially autonomous, no agent has a global view of the system or it cannot use this knowledge practically, there is no controlling agent. Multi-agent systems are very useful in solving problems that are difficult or impossible for an individual agent or a monolithic system to solve. This could be problems like modeling social structures or simulating a trading market. A lot of work has already been done in the field of Multi-agent systems, as it is used for a wide variety of applications.

The ease of use has improved, as a standard for communication between the agents that was defined for industrial and commercial Multi-agent systems was released for public use. This formal IEEE standard called FIPA (Foundation for Intelligent Physical Agent), is commonly used today and focuses on facilitating the interoperability of agents and Multi-agent systems across different software platforms.

Another improvement in the field of Multi-agent systems, is the development of Multi-agent platforms and programming languages. This makes the implementation of Multi-agent systems much easier and makes it possible to create Multi-agent systems that are used in actual operations. The agent platforms that are available today for Multi-agent system development include: DESIRE,

Jadex, TuCSoN and JADE among others [12]. According to [16], the most used platform is JADE. These platforms are often combined with agent-oriented programming languages, which are used for the implementation of the agents' behavior within the Multi-agent system. These languages include: FULX, JACK Agent Language, 3APL, Jason.

2.2 Energy market simulations

The problem discussed in the introduction, is very big and current. It greatly involves the everyday lives of people and is linked to multiple disciplines. Because of this, there is a lot of related work. The change from the centralized and outdated model of the energy network to the new open energy network, requires legislation changes to legally and economically work. These legislation's and the required changes are very different for each country, so most simulation systems can only be used for a single country. As the legislation's differ for each country, these are not discussed in this paper, because it would be a very extensive list. The simulation discussed in this paper is therefore also very general and does not apply to country specific cases.

Other researchers have already created systems that focused on distribution, learning strategies and demand and supply balancing. This was often focused on using the laws of a certain country as a basis and simulating how these had to be changed in order to achieve a more deregulated market. Some of this research is based on software agents like this system. For instance there is some research on the Croatian market simulations in [21].There is a large number of electricity market related papers present in the First International Workshop on Agent Technology for Energy Systems (ATES 2010) [20], that focus on the software and hardware architectures required for supporting the new deregulated market. Solutions have been provided on energy markets in countries, such as Croatia, USA, Australia, Germany, Sweden and Spain. In the related papers there is focus on a wide range of solutions all related to this subject, such as strategic learning, demand/supply balancing, technologies for modeling pricing and platforms.

The technologies that other researchers have used to create a market simulation are very different from each other. This also includes non agent-based systems such as the system proposed by [22], where a web based JSP/Servlet solution is used. This system features a Java powered framework that uses the JSP/Servlet pages as resources. A lot more of the systems that are proposed today are agent systems. Since the year 2000, the first agent-based systems for energy market simulations have been created, with nowadays outdated technologies, such as CORBA and ZEUS. A Java toolkit that originated from agent-based modeling in social sciences, called RepastJ, was used in [23] to create an electricity market framework (AMES).

This thesis uses JADE, which is the most commonly-adopted agent-oriented middle-ware that conforms with Foundation for Intelligent Physical Agents (FIPA), as the agent platform for development, which is motivated in chapter 3.1. There are also other researchers that have used JADE for an agent-based system for electricity market simulation. JADE was used in [13] to develop the wholesale electricity market that is modeled as a Multi-agent system.

There are also agent system that do not use a peer based model like JADE, which are SEPIA and MASCEM. MASCEM (a Multi-agent system that simulates competitive electricity markets) was created in [24] by using Open Agent Architecture (OAA), a framework for integrating heterogeneous software agents in a distributed environment.

2.3 MAS by N. Capodieci

The Multi-agent system that has been used as a basis for this research was created by N. Capodieci[17]. This MAS was built in Java using the JADE platform and therefore uses the FIPA

standard for messages. This MAS simulates a peer-to-peer energy exchange and has Gencos, Consumers and Prosumers as agents. The Gencos simulate the big energy production companies. The Consumers simulate the commercial and private energy consumers. The Prosumers simulate the consumers that have their own solar panels or wind turbines and that generate more energy than they consume. The system also has additional mediator agents, that are used to create more realistic situations. These are the Time agent and the Weather agent. The Time agent is used to create a simulation where the time of day influences the weather. The Weather agent is used to generate random weather and give weather forecasts for the Prosumers. A Balancer agent is used to manage the simulation and display results. In this system the Consumers first try to establish contracts with a Prosumer by negotiating on the price. If this fails, the Consumers search for the cheapest Genco and establish a contract with it. This negotiating strategy could be changed if necessary in future versions of the system.

This MAS is limited to one host machine, which limits the scalability of the MAS to the capabilities of a specific machine. The host can only handle a certain amount of agents and tests in section 6.3.3 show that this is limited to about 300. To provide a more realistic simulation, more agents are needed in the system. Providing a way to run the system on multiple host machines can significantly improve the amount of agents. The MAS is also not very reliable, because the system will no longer function if one of the mediators crashes, because they cannot be restarted and there is only one of each mediator agent active. This can be improved by running multiple instances of each mediator and by providing a way to restart a crashed mediator agent. To implement these improvements, some background knowledge is needed. This is explained in the next chapter.

3 Background

This chapter focuses on the background knowledge needed to achieve a scalable and reliable Multi-agent System. First the JADE framework and its possibilities are discussed. Next the different methods that are available to achieve a scalable and reliable MAS are discussed in detail. Finally a summary and discussion on the selected and non selected methods is given.

3.1 JADE

To create a Multi-agent system, the easiest way is to use a specialized agent programming platform. By using a platform, the implementation of the system becomes much smaller, because the communication and several other aspects have already been taken care of. This makes it possible to focus solely on the agent implementation itself. There is a wide range of Multi-agent platforms available on the Internet, some of which have already been mentioned in section 2.1. Each of these platforms differs in their features and their flaws and a lot of them are no longer being updated or supported. From this wide range of platforms, only one can be selected and used. This platform must match some criteria in order to be able to use the agents' possibilities to their full extent and prove to be the 'right' platform for this job. It must provide an easily updatable environment and a standardized multi-platform programming language, supporting libraries or extensions for fault tolerance, security and distribution.

A few other researchers [16] and [19] have already compared some of the more known, updated and used platforms on a list of criteria. These results can be used as a selection basis for this thesis. From these sources it can be concluded that JADE, or Java Agent Development Environment, is the best choice for general purpose uses. The JADE platform supports Multi-agent system development with Java. The choice of this platform for the implementation of the Multi-agent system was based on some advantages and criteria [16][17][19]:

- The MAS in this paper is based on the MAS by N. Capodieci, which also uses JADE for implementation. The advantage of using the same implementation platform is that no recoding is needed, as the same implementation work can be used and adapted for the new system.
- JADE is updated regularly and has a large development crew and community.
- JADE uses Java and each agent is run in a separate thread, which is faster than conventional Java threads.
- JADE works on any platform that supports a Java Virtual Machine, or JVM.
- The methods and architecture proposed in this paper do not conflict with the possibilities of the JADE platform and Java.
- The JADE platform itself already implements and uses some of the methods proposed in this paper, which is shown in section 4.1.
- The JADE platform is free and open source, which makes this a cheaper choice than a paid alternative and allows for customization of the source code.
- There is standard Java API documentation for JADE, as well as numerous other Internet sources containing tutorials, manuals and Q and A. Most of these are largely up to date.
- It has an excellent GUI with a lot of useful features and tools.
- It has already been used in a lot of development and research projects and has a high acceptance rate in the community.
- It supports the FIPA specification standard for Multi-agent system messaging.
- There are very good security features, such as SSL support for inter platform communications, permission grants and added security possibilities.
- The platform is easy to distribute on multiple hosts.
- There is a wide range of different extensions and libraries for additional features, such as added security, web service integration and embedded JADE for small devices.
- It supports multiple communication and transport protocols, such as socket, RMI and IIOP communication.

Not all of the features and advantages mentioned here are used for the MAS discussed in this paper. It is also possible to use a special agent language together with JADE for the implementation of the agents, but this is not used in this implementation as the Java programming language provides enough possibilities in this case for implementing the agent behavior.

3.2 Scalability improvement methods

In this section the methods that can be used to improve the scalability of the MAS are explained in detail. The advantages and disadvantages of each method are discussed after which a summary is given.

3.2.1 Agent-level improvements

These methods focus on the agents implementation and organization, in order to improve scalability. There are two methods described here, changing the agent organizational form and

locating the agents based on caching lists.

Change agent organizational form

Jennings and Turner have defined several organizational forms of MAS in [4], suitable for a trading scenario, comparable to our MAS. The forms are distinguished by the constraints within which the agents interact with each other. They have defined three different forms.

In the first organizational form, each customer can communicate with each supplier and the other way around. But customers are unaware of other customers and suppliers are unaware of other suppliers. Which means that agents of the same type cannot share information, form groups or undertake co-operative behavior. In this case agents are asocial and do not exhibit properties expected of MAS agents.

The second organizational form is the same as the first form, with the exception that it is also possible for costumers to communicate with other customers and for suppliers to communicate with other suppliers. Agents are now able to share information and form groups with other agents, enabling them to co-operate and share tasks. In this case agents are social and represent a standard fully connected peer MAS.

The third organizational form expands on the second form, by adding an intermediary agent that undertakes collective tasks. This agent performs intermediary functions, such as matchmaking, recruitment, facilitation, etc., thus relieving the other agents of this work.

Changing the organizational form, can increase the scalability of the system, because agents can share tasks and intermediaries can reduce the workload on other agents.

Advantages

Choosing the right form for the current MAS can reduce the communication overhead and increase efficiency, by introducing more agent teamwork.

Disadvantages

A disadvantage of this method is that some research has to be done, in order to pick the right organizational form for the current MAS. Also the chosen form might later turn out to be non-efficient for the current MAS. It is also possible that none of the forms matches what is required in this case.

Locate agents based on agents caching list

Each agent in a MAS needs to know where other agents are, in terms of addresses, in order to communicate with them. This process can be time consuming, if the agent does not know what the addresses are. To increase the performance and scalability, agents can use caching lists to store the location of other agents [6]. In this approach, each agent has a list of other agents it knows. This list stores all the relevant information about other agents, such as addresses, names and expertise. This list may not be up to date or correct, and changes dynamically. It can be assumed that with a high message reliability and a slow frequency of change, the agents lists are largely up to date and accurate.

When an agents needs an address of another agent, it checks its caching list. If the address is not there, the agent will contact some, or all agents in its caching list, for the address information. These other agents, will perform the same procedure recursively. To prevent duplicate request handling, a unique request id. is used. To guarantee cooperative behavior, payment schemes can be used. The communication overhead of this method is very low, with an average complexity of O(1), if the contact list is limited to a certain size.

Advantages

An advantage of this method is that it removes the need for middle agents to serve as brokers. The communication overhead is therefore reduced. It is also very suitable for heterogeneous MAS, because there is no dependence on middle agents. The method also has a very low overhead.

Disadvantages

This method may not be very suitable for unstructured MAS, because of the required inter-agent communication, which may result in slow response times.

3.2.2 System-level improvements

These methods focus on the system structure and components, in order to improve scalability. There are five methods described here: hiding communication latencies, component distribution, component replication, agent scheduling and transparent access.

Hiding communication latencies

This method, proposed in [2], focuses on geographical scalability. If a MAS spans a large area network, there may be severe communication latencies. Agents may be waiting very long for responses from other agents. These latencies cannot simply be solved, but it is possible to change the agents. The agents can be adapted to do other useful work, while they wait for responses. In this way the communication latencies can be hidden. It requires that agents can be interrupted when a response is delivered.

Advantages

When possible, a major advantage is that agents can perform other tasks when waiting for a response. The communication latencies can be largely hidden by doing other useful work.

Disadvantages

The agents have to be interrupt-able, to be able to handle the asynchronous requests. The agents must be able to do other useful work, instead of waiting for a response.

Component distribution

Component distribution can be used to partition the MAS over multiple separate servers [1]. Agents are distributed over different physical machines, in order to spread the load. The distribution of components can be manually done by a human programmer. The different components can be hosted in different processes. This requires inter-process communication, which can be realized with Java RMI, simple socket communication or JADE.

If component distribution is used in large-scale networks, this method should be combined with hiding communication latencies, whenever possible, in order to ensure performance increase.

Advantages

An advantage of this method is the easy implementation and the instant increase in scalability of the system. A large amount of machines can be added to support a large scale system.

Disadvantages

This method does have some drawbacks, such as the manual distribution. The designer must decide which components are distributed and how they are distributed. The best distribution strategy is therefore difficult to achieve, because of the varying load situations, which complicates adjustments to the distribution. Another disadvantage is that the distributed components are bound to one

machine and cannot scale beyond the limits of this machine, unless the implementation supports dynamic moving of components to other machines.

Component replication

Component replication can be used to replicate certain components of the MAS across a network [1]. This can improve scalability by reducing communication latencies, by placing components close to where they are needed. Expected performance bottlenecks can be resolved by replicating. In this way it can also spread the load on certain components, by decentralized load balancing. The replicated components can be hosted in different processes and on different machines. This requires inter-process communication, which can be realized with Java RMI, simple socket communication or JADE.

Advantages

One advantage is the reduction in communication latency, by bringing the components close to the agents that use them. Another advantage is the possibility to prevent bottlenecks in the system, by replicating heavily used components.

Disadvantages

A disadvantage of replication is inconsistency problems. These can be overcome, but introduce some amount of overhead. The replicas must be consistent with each other, which can be achieved by global synchronization or by adopting a weaker consistency model. This consistency model depends heavily on the application.

Another disadvantage is that replication increases resource consumption and complexity. By adding more agents, resources are wasted since all the agent specific services are also replicated. In addition, the system becomes more complex, since more components have to be managed. Load balancing is required for this to work.

Agent scheduling

To increase performance of individual hosts and therefore scalability, agent/thread scheduling can be used [1]. This method enables the execution of large numbers of (reactive) agents. With agent scheduling, the agents that are not performing any tasks, are deactivated and only require memory resources. The agents that are active and performing tasks, can use all the resources. This scheduling of agents preserves the resources for the active agents, preventing resource wastes. To make the scheduling most efficient, there should be a large group op deactivated agents and a small group of active agents. To determine which agents should be deactivated and which agents shouldn't, a scheduling policy must be used. This policy must also be able to control each agents' access to system resources. There are multiple scheduling policies, which use ranking of importance, statistics and heuristics.

A common form of scheduling involves messages received by agents. The agents that received messages are moved from the deactivated group to the active group. After the message has been processed, the agent is moved back. A variant of this approach uses events, such as a user logging on or off. The agents that are associated with this event, are moved to the active group. After the event has been processed, the agents are moved back. In both approaches, most of the agents should be reactive to make the methods efficient enough.

Advantages

The major advantage of this method is the large increase in performance that can be achieved, by efficient scheduling of the active and inactive agents.

Disadvantages

Unfortunately this method is not useful for large numbers of pro-active agents, because there is only a small number of inactive agents in this case and the performance can actually decrease in this case, because the scheduling itself is also computationally intensive.

Transparent access

Transparent access provides a possibility to enable a MAS to scale beyond the limitations of underlying physical machines [1]. Scalability can be improved by providing transparent access to the distributed resources available. Transparent access prevents additional complexity of the MAS, by hiding resource locations. This results in simple access and flexible adding or removing of resources. Transparent access can be realized by using a transparent resource management layer to use/create threads and objects within other processes. The transparent access layer allows a host to farm out the execution of agents. Only by distributing the load it becomes possible to ensure that a large number of agents reside in a single agent host.

Agents themselves cannot access the system resources or services directly, but only through an environment object. This environment object is a proxy that keeps the implementation of its public methods hidden. This helps achieve two goals, fine-grained control and location independence. Fine-grained control of the agents, provides a way to distribute resources among the agents according to the importance or vitality of their services and to disconnect troublesome agents. Physical location independence of the agents is achieved by interaction via a proxy and by hiding the location details. Agents can thus be moved freely by the system between processes.

Advantages

The major advantage of this method is that it increases the location independence of the MAS. This makes it possible to use different kinds of physical machines and/or software and add or remove resources. This method also increases the effectiveness of other system-level methods, such as replication and distribution.

Disadvantages

This method can increase the overhead on the system, because an extra layer is added to the system.

3.2.3 Summary

Method	**Advantages**	**Disadvantages**
Change agent organizational form	Can reduce communication overhead and increase efficiency.	Organizational form must fit the problem that the agents are modeling.
Locate agents based on agents caching list	No need for middle agents. Is suitable for heterogeneous MAS.	May not be suitable for unstructured MAS.
Hiding communication latencies	Agents can perform other tasks when waiting for a response.	Agents must be interrupt-able and immediate communication must not be required.
Component distribution	The components are distributed, thus spreading out the workload.	Must often be combined with Hiding of communication latencies, to ensure performance increase. The components must be manually distributed.

		Components are bound to a single machine.
Component replication	Data is close to the agents. Bottlenecks can be prevented, by replicating heavily used components.	Possible data inconsistencies. Load balancing required.
Agent scheduling	Can increase performance, by efficient scheduling of active and inactive agents.	Useless for large numbers of pro-active agents. Computationally expensive.
Transparent access	Increases location independence. Increases effectiveness of other agent-level methods.	Can increase overhead.

3.3 Reliability improvement methods

In this section the methods that can be used to improve the reliability of the MAS are explained in detail. The advantages and disadvantages of each method are discussed after which a summary is given.

3.3.1 Agent-level improvements

These methods focus on the agents implementation and organization, in order to improve reliability. There are four methods described here: using sentinels to check the system, using agent teamwork to handle agent failures, refuse requests ability and increase agent mobility.

Using sentinels to check the system

Sentinels can be used to increase the reliability of the system [8][14]. These sentinels are agents, which can guard specific functions or guard against specific states in a MAS. It is up to the designer to decide which functions are most vital for the systems integrity, because not all of the functionality can be guarded. Sentinels can take several actions to guard the system. They can choose alternative problem solving methods for agents, exclude faulty agents, alter parameters for agents, and report to human operators. They do not take part in the problem solving of other agents, but they can intervene in this process. By using semantic addressing, the sentinels can interact with other agents and monitor their communication and interaction, in order to build models of these agents. Some parts of these models are exact copies of the agent models and are called checkpoints. These points assist in detecting faulty agents and inconsistencies, by providing information of the internal state of an agent and its behavior Timers can be used to detect crashed agents or faulty communication links.

Advantages

The advantage of sentinels is, that they are separable from the system. The sentinels can be added after the whole system has been developed and tested. They can also be modified, without affecting the system. The communication mechanisms used by the sentinels and the relevant checkpoints can also be created and altered when needed.

Disadvantages

A disadvantage is that the freedom of the agents is limited by the sentinels. Also the functions that need to be guarded have to be decided by the designer and the system must have support for fault handling and reporting, in order for the sentinels to work. This method is also not very suitable for

high volume MAS with highly frequent messages, because a lot of sentinels have to be added in this case, and they have to process a lot of messages.

Using agent teamwork to handle critical agent failures

Using agent teamwork to handle critical agent failures can be used as a method to increase the reliability of the system [9]. This method involves the usage of teamwork between the agents in the system as a technique to automatically recover a Multi-agent system from a sudden agent failure. These failures could be caused by a machine crash, network breakdown, or death of the agent process.

Each agent in the system finds other agents in the system and stores their name or address to be able to communicate with them. When a critical agent disconnects from the system, each agent that fails to contact this agent, attempts to inform the other agents in the system of this failure. Only the agents that regularly communicate with the now disconnected critical agent are informed. After successfully contacting an agent in this manner, this agent updates his information and gives up his attempts to contact the disconnected critical agent.

The Multi-agent system has recovered from failure of the disconnected critical agent when all the agents that interact with that agent have been contacted in this manner. The requests that were in progress at the time of the failure, and hence could not be completed, may be sent again by the requesting agent. This can be considered fault tolerant behavior and hence improves the reliability of the system.

Advantages

Results in minimal overhead, as the teamwork is only used in case of an agent failure. Critical agent failures can be solved and cascading effect can be prevented. Also this method is easy to implement, as it only requires a few special messages and some code to read them and to act on them.

Disadvantages

The use of teamwork may interfere with the required autonomy of agents in the MAS.

Refuse requests ability

The ability to refuse requests can increase the reliability of the agents [9]. Agents can refuse requests to stop flooding of messages. This is making the agents more autonomous and less susceptible to the influences of other agents. It can be implemented by using a message queue and refusing messages if the queue exceeds the maximum queue length.

Advantages

This method can prevent agent thrashing and make the system more reliable. Agent thrashing can occur when there are more messages being received by an agent than it can handle. These messages may stack up and consequently slow the entire system down. If messages are refused, this can no longer occur.

Disadvantages

A disadvantage of this method is the discarding of the messages itself. Some MAS models may require that no messages are discarded, or rely on certain messages being received.

Increase agent mobility

Agent mobility is measured in the ability of agents to be moved from one host to another. Agent mobility can be improved by increasing protocol independence and host independence of agents.

Increasing agent mobility can provide a more fault-tolerant system. For example, if a host is experiencing computational problems due to too many agents, the computational intensive agents can be moved to another host.

Advantages

The agents are no longer tied to certain protocols and/or hosts. In case of failures on a certain host, the agents could be moved to another host.

This method can also increase the scalability by providing a way to support load balancing in a distributed environment, by moving agents.

Disadvantages

The moving of the agents itself could be computationally intensive, depending on the number of agents being moved and the size of the agents' data.

3.3.2 System-level improvements

These methods focus on the system structure and components, in order to improve reliability. There are four methods described here: distinct domain independent exception handling service, active replication, passive replication and critical agent/adaptive replication.

Distinct domain independent exception handling service

An exception handling service can be used to provide a way of reducing the exception handling within the agents [10]. This domain-independent service handles all the exceptions that occur within agents and thus reducing the load of the agents. The exception handling can be separated from the agents doing the logic and provide a way of control. The agents become simpler and do not need to know about the exception handling. This is also called the "citizen" approach. It requires at most that agents support three very simple directives (‘‘are you alive?’’, ‘‘resend RFB’’, and ‘‘cancel-task’’). The service can prevent cascading effects of an exception, by informing other agents of the failure. The method enhances the reliability by offloading exception handling from problem solving agents to distinct, domain-independent services.

Advantages

The load on the agents in the MAS is reduced, by moving the exception handling from each agent to a central location. The agent implementation becomes simpler, as the agents do not have to handle the failures themselves. Fault cascading effects can be prevented.

Disadvantages

This method results in a more centralized system, which may conflict with the required autonomy of the MAS. Another disadvantage is the dependency of the service on communication with the agents. If this fails, the service is no longer able to detect faults.

Active replication

Replication can be used for data and/or computation, to make a distributed system more fault-tolerant [7][11]. Active replication is a replication protocol where each component is replicated and all replicas concurrently process all input messages. This increases reliability, because a replica can immediately replace another, in case of a system failure.

Advantages

The advantage of active replication is, that it provides a fast recovery delay and is ideal for real-time

constrained systems.

Disadvantages

Active replication leads to a high overhead, the overhead equals the amount of replicas. Which makes this method more resource intensive than passive replication. This method is not very suitable for large-scale, adaptive replication.

Passive replication

Replication can be used for data and/or computation, to make a distributed system more fault-tolerant [7][11]. Passive replication is a replication protocol where each component is also replicated, but only one of the replicas processes all input messages and periodically transmits its current state to the other replicas in order to maintain consistency. If the active replica is faulty, a new active replica is chosen from the passive replicas and the execution is restarted. This increases reliability, because a mostly up to date backup can be restored, in case of a system failure.

Advantages

This method requires less CPU resources than the active approach, by activating redundant replicas only in case of failures and still provides a reliable backup mechanism. It also has a low overhead under failure free execution, because of the periodic updates.

Disadvantages

This method needs a checkpoint management which is still expensive in processing time and space and does not provide short recovery delays. As well as active replication, this method is not very suitable for large-scale, adaptive replication.

Critical agent/adaptive replication

A different replication protocol is based on the criticality of certain agents [11]. Only those agents that are defined as critical are replicated and the others are not. Furthermore, one must determine the most critical agents and the needed number of replicas of these agents. There are two cases here:

The agent’s criticality is static, in which case, the organization structure of the agents doesn't change, the behavior is static and the number of agents is small. In this case the critical agents can be identified before run time and replicated where needed.

The agent’s criticality is dynamic, in which case, the organization structure of the agents is dynamic, the behavior is dynamic and the number of agents is large. In this case the critical agents cannot be identified before run time and must be based on the dynamic organizational structure. This can be achieved by observing the domain agents and dynamically evaluating their criticality, based on semantic-level information and system-level information.

This approach increases reliability, because the critical agents are replicated and can replace crashed critical agents. Non critical agents are not replaced in this case.

Advantages

Not a very big impact on performance, because not all agents are replicated, only the critical ones. The system is much more reliable, because it can keep functioning despite failures.

Disadvantages

A system where all of the agents are critical is not suitable for this method, because of the performance impact. Replicas may require synchronization for the system to function correctly.

3.3.3 Summary

Method	Advantages	Disadvantages
Using sentinels to check the system	Sentinels can be added later on and can be modified on the fly.	Not very suitable for high volume MAS with highly frequent messages. Agent communication and world model needs to be public.
Using agent teamwork to handle critical agent failures	Can recover from critical agent failures. Prevents cascading effects. Minimal overhead. Simple implementation.	May interfere with the required autonomy of the MAS.
Refuse requests ability	Prevents agent thrashing.	Might not suit all the MAS applications, because important messages might be discarded.
Increase agent mobility	Agents are not tied to certain protocols and/or hosts.	Moving the agents around can be computationally intensive.
Distinct domain independent exception handling service	Reduced load on the agents in the MAS. Simpler agent implementation. Fault cascading effects can be prevented.	Centralized approach. Relies on communication with the agents.
Critical agent/adaptive replication	Not a big impact on performance. System still functions despite agent failures.	Not suitable for systems with large numbers of critical agents. Replicas may require synchronization.
Passive replication	Minimizes processor utilization by using checkpoints to restore faulty agents.	Requires checkpoint management which is expensive in processing time and space.
Active replication	Provides fast recovery.	Lead to a high overhead.

3.4 Selected methods

From all the methods described in the previous sections, some have been selected as usable for this MAS. The next two sections discuss the selected methods for scalability and reliability and explains why these have been selected.

3.4.1 Scalability

Not all of the methods described in the previous section are used. Some of the methods are not suitable for this MAS, or do not provide an increase of scalability in this case. The methods that are used are:

- Locate agents based on caching lists
- Distribution
- Replication

- Agent scheduling
- Transparent access

Locate agents based on caching lists is also used, because it reduces the load on the middle agents/brokers to handle all communication as agents can store agent locations themselves. Especially when the current MAS is distributed, the load on the middle agents/brokers could become very large. This method also increases support for possible future changes, as it is suitable for heterogeneous MAS. The current MAS is not unstructured, so the disadvantage is not a problem.

Distribution is used, because it is an essential method for increasing scalability. Without distribution the whole system is bound to one machine. With this method the agents are still bound to their respective machines, but not to only one. The agents do have to be distributed manually, but by examining the structure of the MAS this should not pose a big problem.

Replication is used, because it can further increase the performance gain of distribution. This is done by replicating the heavily used agents/components. This should make the system more scalable than by having only one of these components. Communication distances/latencies are also decreased by this method. The possible problems with this method can be solved by implementing a data consistency update. Load balancing is partially solved by the use of distribution and by limiting the amount of replicated agents.

Agent scheduling is used, because the current MAS does not have a large number of pro-active agents and thus does not limit this method. At certain times there are a lot of inactive agents in the system, waiting for responses, or when the auctions have ceased. Agent scheduling can make the system more efficient.

Transparent access is used, because it increases the effectiveness of distribution and replication and also makes these methods easier to implement. It also increases the location independence of the agents, thus making it easier to distribute these. The increase in overhead is limited and does not compare to the performance increase gained by using this method in combination with distribution and replication.

The methods that are not used are:

- Change agent organizational form
- Hiding communication latencies

Change agent organizational form is not used, because the alternative organizational forms do not apply to the current MAS. The current MAS uses a scheme where the suppliers communicate with the Consumers and where top-level intermediaries interact with the suppliers and Consumers. One alternative organizational form requires removal of the top-level intermediaries. Removing the top-level intermediaries is not a viable solution, because their functionality has to be separated from the Consumers and suppliers and cannot be incorporated within these agents. The other form requires intercommunication between the Consumers and between the suppliers. This is not useful, because the Consumers have no messages or information which they need to discuss with themselves. This also goes for the suppliers. Therefore changing the form would result in loss of functionality or useless overhead, which is why it is not used.

Hiding communication latencies is not used, because the current MAS is not meant to be run on an Internet-scale network, meaning that communication latencies will be limited. Also the agents of the system do not have many other tasks to perform when waiting for a reply, making this method not efficient enough to implement.

3.4.2 Reliability

Not all of the methods described in the previous section are used. Some of the methods are not suitable for this MAS, or do not provide an increase of reliability in this case. When choosing the methods that are used, the degree of fault tolerance is of great importance. According to [8] there are four main sources of faults:

- Inadequate specification of software
- Software design error
- Processor failure
- Communication error

where the first two are unanticipated in consequences and the last two can be considered in the design of the system. Even if all possible measures are taken to prevent faults, the first two sources above imply the difficulty in building fault-free systems. This emphasizes the need for fault tolerance. If the system cannot handle a fault and shows unexpected behavior, there is a system failure. If, on the other hand, the system can handle the fault situation, it is called fault tolerant. For the MAS to be more reliable, it needs to be fault tolerant as well. The following degrees of fault tolerance are proposed by [8]:

- Full fault tolerance, where the system continues to operate without significant loss of functionality or performance even in the presence of faults.
- Graceful degradation, where the system maintains operation with some loss of functionality or performance.
- Fail-safe, where vital functions are preserved while others may fail.

For this MAS, the aim is to achieve Graceful degradation, because Full fault tolerance is very difficult to achieve in a Multi-agent System and usually results in a performance decrease caused by the required methods, which will conflict with the requirement of a more scalable system. On the other hand, Fail-safe would be too low for fault tolerance, as the aim in this thesis is for a more reliable system. The methods that would be required for a Full fault tolerant system are Active or Passive replication and Using sentinels to check the system. These are not necessary for Graceful degradation, as Critical agent/adaptive replication and the Distinct domain-independent exception handling service provide enough means to maintain system operation and most functionality.

This degree of fault tolerance results in the following methods that are used:

- Using agent teamwork to handle critical agent failures
- Refuse requests ability
- Distinct domain independent exception handling service
- Critical agent/adaptive replication

Using agent teamwork to handle critical agent failures is used, because it provides a way to handle failures or crashes in the critical agents that are used. These agents are critical to the system and the other agents in the system should be informed if these agents fail. By informing other agents, this method also prevents cascading effects of a failure. The overhead of this method is limited, because only in case of a failure additional messages are sent and additional functions are called. Finally the implementation is also simple. The autonomy of the MAS is no problem in this case, as the agents do need to be movable and do not require complete autonomy.

Refuse requests ability is used, because the buyers and sellers can be flooded by the offers sent to

each other as the system is scaled up. This method prevents thrashing of the agents by message floods. It is also easy to implement, by using a message queue with limited length in combination with a garbage message collector. The disadvantage is not a problem in this MAS, because the offers can be re-sent and are not critical for the system.

Distinct domain independent exception handling service is used, because it makes exception handling easier than in the normal case. The implementation of the agents in the MAS becomes simpler, because of the central handling. The agents can perform more useful tasks instead of exception handling. Another big advantage is the preventing of fault cascading effects, which are common in Multi-agent systems. The disadvantages of this method are not problematic, as the required autonomy of the MAS is not violated. The agents can still act independently and negotiate on the prices. The failure of communication is a problem, which also applies to the entire MAS and therefore cannot be considered a major disadvantage of this method.

Critical agent/adaptive replication is used, because it increases the reliability of the system significantly. Failures of agents in the system no longer result in failure of the entire system. The impact on performance is very limited, as only the critical agents in the system are replicated. There is only a small number of critical agents in this MAS. The main portion of agents are the suppliers and Consumers. The suppliers and Consumers are not critical agents, as the system will continue to function despite failure of these agents. This makes one of the disadvantages obsolete. The synchronization of replicas is also limited, because there are not many critical agents and because almost no functionality needs synchronization.

The methods that are not used are:

- Using sentinels to check the system
- Increase agent mobility
- Passive replication
- Active replication

Using sentinels to check the system is not used, because the it is not useful for high volume MAS with highly frequent messages, which resembles our MAS. The method also results in some communication overhead and limits the freedom of the agents in the system. Another point is that the checkpoints have to be decided manually and that the sentinels can be difficult to implement. The advantage of sentinels is not big enough to compensate these problems.

Increase agent mobility is not used, because it is computationally expensive to move agents around. Also the advantage of being able to move agents around does not weigh against the cost of moving in the current MAS. The agents are distributed manually and it is expected that the computational intensity of the agents does not vary much during the experiment, thus removing the need for load balancing by moving agents between workstations. Also the chances of a workstation crashing completely and requiring moving of agents, are not very high and acceptable for the current MAS, because of the limited running time.

Passive replication is not used, because it replicates all the agents and results in a less efficient system. All the information must be updated to the replicas of each agent and requires an inefficient checkpoint management system. The recovery delay is also high and becomes higher as the size of the system increases. This conflicts with the scalability demands of the system, also the system should be real-time. Also a replication of all the agents is not required for the current MAS to continue functioning.

Active replication is not used, because it also replicates all the agents and it results in a high overhead because all the replicas are updated in real-time. The overhead becomes larger as the

system increases, which also conflicts with the scalability demands. The recovery delay is not an issue, as it is designed to be real-time. The replication of all the agents is not required for the current MAS to continue functioning.

4 Architecture

The architecture and the agent topology of the MAS is discussed in this chapter. The JADE architecture is explained first, as it is used as a basis for the MAS. The improvements and changes are explained in the next section. In the final section the current architecture and interactions of the agents are modeled and explained.

4.1 JADE

The JADE platform is a distributed system, because it can be spread over multiple hosts. On these hosts there can be one or more Agent Containers, each one living in a separate Java Virtual Machine, which enables platform independence. Within each of the agent containers there can be one or more agents located. Java RMI is used for communication between the Agent Containers. Inter-platform IIOP communication is used for the outgoing messages to foreign agent platforms. One of the containers is acting as a front-end IIOP server and waits for the incoming messages from other platforms on the official agent platform ACC address[12]. This defined JADE software architecture is shown in Figure 1.

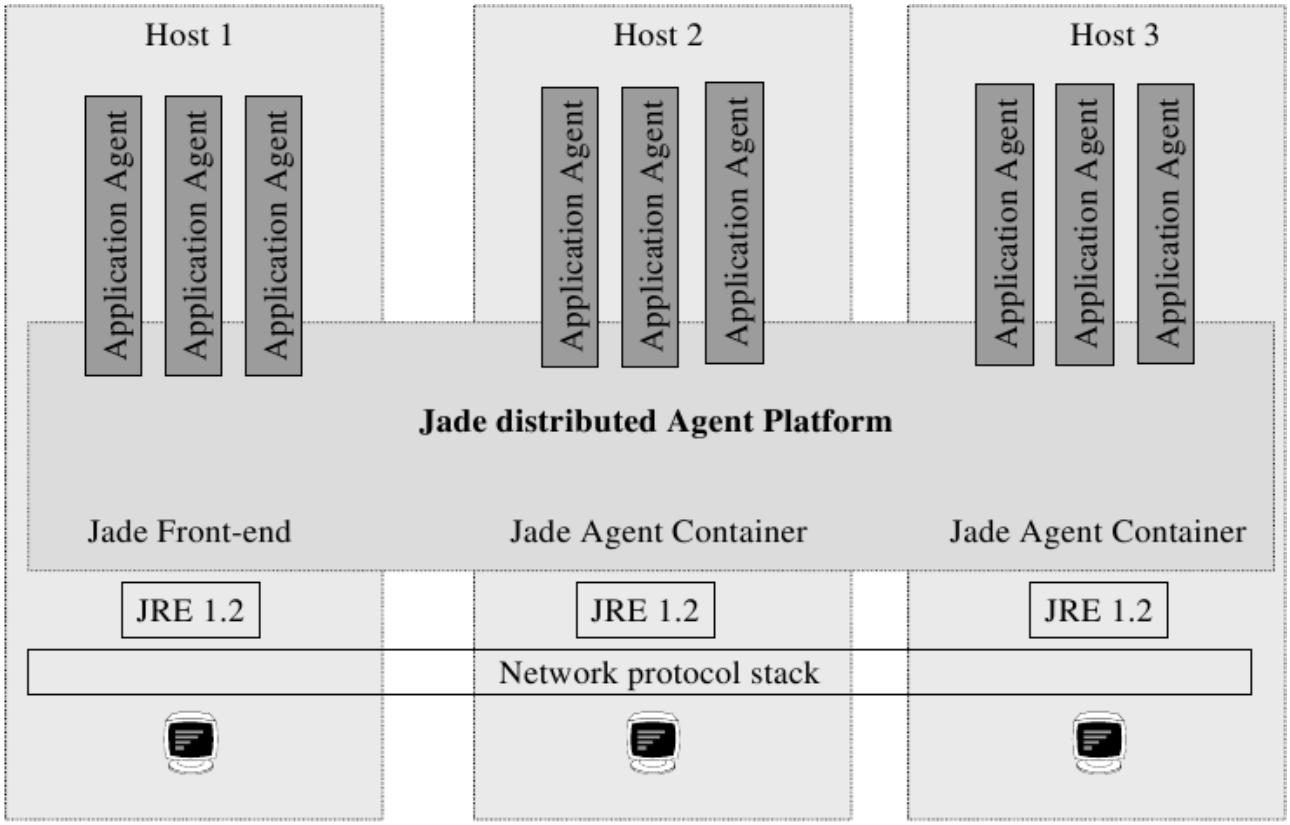

Figure 1: Software architecture of a JADE Agent Platform [12]

The JADE platform includes three system agents that are used to manage it. These are the ACC, the ASM and the DF. The DF is a yellow pages service agent, which enables the agents to register themselves and find other agents. All the agents communicate through message passing, using the

FIPA ACL. To support the agent communication, each agent has a globally-unique identifier (GUID), as specified in the FIPA model, in the form: <agent name> @ <platform address>. This GUID is used to send ACL messages to agents. Each sending agent also send along its GUID, to make replying messages easy. Each time an ACL message is sent via JADE, three choices are possible for the transport mechanism depending on the receivers location[12]:

- If the receiver is located in the same container on the same platform: Java events are used, the ACLMessage is simply cloned.
- If the receiver is located on a different container on the same platform: Java RMI is used, the message is serialized at sender side, a remote method is called and the message is deserialized at receiver side.
- If the receiver is on a different platform: IIOP is used, the ACLMessage is converted into a String and marshalled at sender side, a remote CORBA call is done and an unmarshalling followed by ACL parsing occurs at receiver side.

JADE uses these transport mechanisms in combination with agent caching lists to maximize efficiency. Each container has a table of its local agents, or Local-Agent Descriptor Table (LADT), which is used to translate the RMI object references to the individual agents. The front-end IIOP container also has a Global-Agent Descriptor Table (GADT), which is used to translate every agent into the RMI object reference of its container. To increase the efficiency of this system and prevent continuous querying of the front-end IIOP container for address information, JADE has transparent address caching. This also enables support for agent mobility and new protocols [12]. Together, these methods implement the Locating of agents based on caching lists, mentioned in 3.4.1. This reduces the load on the middle agents/brokers, such as the front-end IIOP container and increase the scalability of the distributed MAS discussed in this paper.

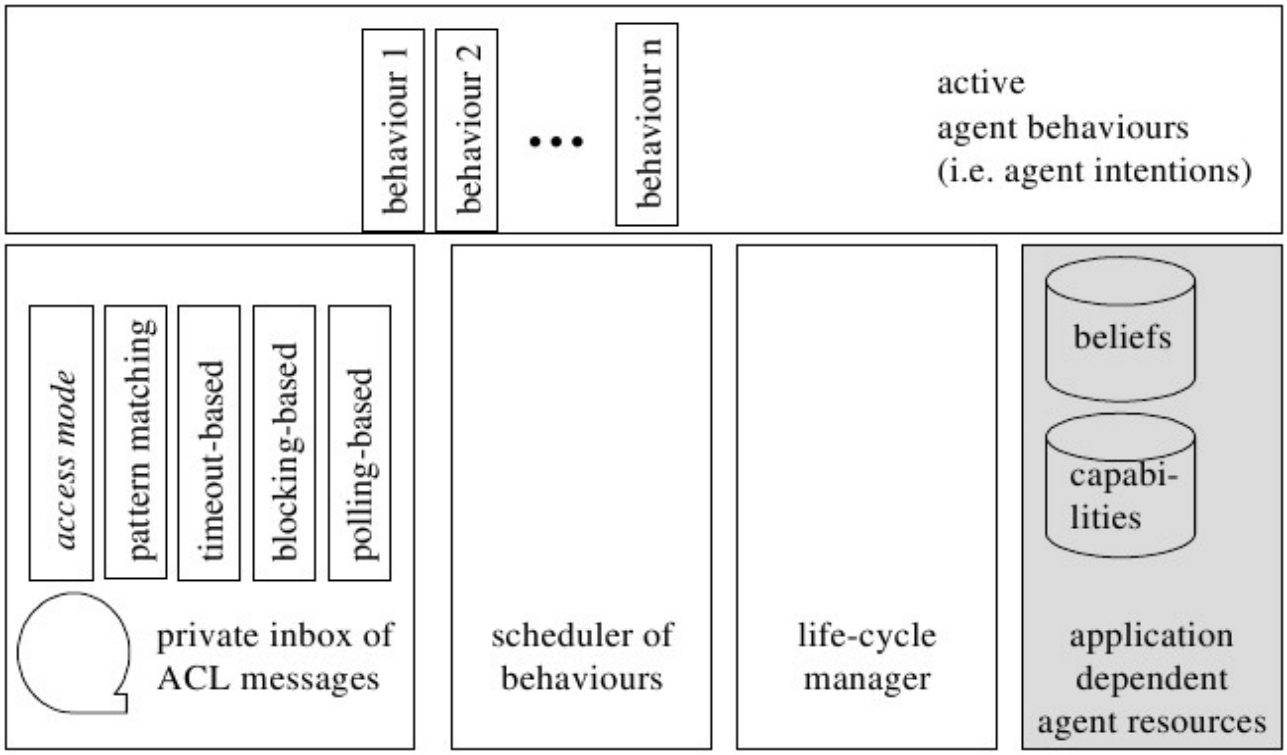

Figure 2: JADE agent architecture [12]

In JADE every agent is run in a separate Java thread, which is very good in satisfying agent autonomy. The tasks that each of these agents performs are put in Behaviours. Each agent can have

multiple behaviors, which are executed in order to perform their tasks. These behaviors represent logical threads of a software agent implementation. To limit the amount of threads in the system, the behaviors of each agent are scheduled and executed cooperatively within the Java thread of this agent. This can be seen in Figure 2, where the agent architecture is shown. In principal JADE therefore uses a thread-per-agent execution model with cooperative intra-agent scheduling. Each agent schedules the different behaviors with cooperative scheduling on top of the stack. This means that all the behaviors are run within a single stack frame created by the agent, that is currently on top of the stack. And that each behavior runs until it returns from its main function and it cannot be preempted by other behaviors. The execution context of each behavior is therefore not saved and this means that each behavior has to start from the beginning, each time they are scheduled for execution. In order to maintain behavior states, variables have to be used within the behavior indicating the current state [12]. The structure of the agent execution model implements Agent scheduling, mentioned in 3.4.1. This makes the system much more efficient, because the inactive behaviors and agents (waiting for incoming messages) are not executed, saving valuable processing time and therefore increase the scalability of the distributed MAS discussed in this paper.

4.2 Improvements and changes

The MAS by N. Capodieci (old MAS) has been improved on a number of points, the architecture related improvements are discussed here. Some of these improvements focus more on the implementation, but these are discussed in section 5.1. Most of the modifications to the old MAS have are related to the methods that have been chosen in section 3.4. The improvements are therefore either related to scalability or reliability.

4.2.1 Scalability improvements

The methods that have been described in section 3.4.1 were implemented in the system. The results from some of these methods are visible in the new architecture and are discussed here. The first method, of which its use is very clear in the system, is Distribution. Distribution provided a way to increase the amount of agents that can be run simultaneously on the system, by increasing the amount of hosts. Therefore the improvements are focused on the expansion of the amount of hosts. Each additional host also means at least one additional container that is running on this host. In each container on each host a certain amount of agents can be running. This is an improvement from the old MAS, where only one host with one container was present. Another possibility that is added by distribution, is efficient distribution of the areas discussed in the previous chapter. A logical choice was linking each area to a container on a host. Each area can now contain more agents than in the old version. By using distribution, each host now uses an Agent creator to start the agents on that particular host. Scalability is improved significantly, because the system is no longer limited to the hosts.

Distribution requires one of the other methods in order to work correctly and efficiently, which is Transparent access. Transparent access solves platform inconsistencies between the different hosts and enables simple communication between these hosts. The agents on each host can communicate with each other, without needing to worry about where they are. Communicating with an agent on the same host is no different from communicating with an agent on a different host. The JADE platform is a transparent access platform, so no additional implementation is required. These improvements are visible in the new architecture.

Critical agent replication is another method that is used to improve the system. Critical agent replication is one of the most clearly visible improvements in the architecture. In this MAS, the critical agents are the top-level intermediaries, because without these agents, the system does not

work. In the old MAS, only one top-level intermediary of each type was running with the system. This has to be changed in the new MAS, because of the distribution. If only one top-level intermediary of each type is running, these agents could become bottlenecks if the system is increased in size. To improve efficiency and resolve bottlenecks, on of each top-level intermediary is running on each host. So instead of only one time agent, Balancer agent and weather agent, the amount of these agents now depends on the amount of hosts. An addition to the replication is the usage of the weather agents. These agents can now be bound to a specific area, as discussed with the distribution. Each area is bound to a container on a host, so one area can correspond to one weather agent. This enables the possibility of simulating local weather. By preventing bottlenecks the scalability of the system is greatly improved, especially when more hosts are added.

4.2.2 Reliability improvements

Some of the methods mentioned in section 3.4.2 that have been implemented and which involve reliability are also visible in the architecture. The first method that involves reliability improvements, is critical agent/adaptive replication. The improvements made by this method that are related to the scalability have already been discussed in the previous section. The reliability of the old MAS was limited, because there was only one agent of each top-level type. If one of these agents would fail, the whole system would fail. To counter this, critical agent replication can be used. The critical agents are in this case the top-level intermediaries. These agents are replicated in the new MAS, which means that if one agent would fail, the system would still be running. To make sure that the all the Balancer agents that are replicated have the same information and state, the messaging in the new MAS is based on a broadcast. In the old MAS the information would only be sent to a single Balancer agent. In the new MAS the information is sent to every Balancer agent. This makes the new MAS much more reliable and able to continue functioning despite certain failures.

Another major improvement of reliability is the usage of a distinct domain independent exception handling service. This service takes the form of an agent, named EH(Exception Handling) agent. This agent was not present in the old MAS. It has been added to handle agent failures and is able to restart failed agents. The agent waits for incoming messages that inform him of a failed agent. This faulty agent is then restarted and resumes normal operation. This makes the system more fault tolerant by improving the ability to handle system faults. This results in a system that can cope with failures and maintain normal operations.

In support of the EH agent, some features of the method using agent teamwork to handle critical agent failures are used. The agents in the system use a certain amount of teamwork when informing the EH agent of agent failures. The agents check for failed message deliveries, indicating a failed agent. The EH agent also informs all the agents in the system if an agent is restarted and for some agents that are restarted, the Balancer agent needs to resend some information to the restarted agent.

4.3 The Multi-agent system

The Multi-agent system that is described in this chapter is based on the system created by N. Capodieci and is comparable to a basic buyer and seller Multi-agent system, with a few additions. The agent topologies of these systems are largely the same and are based around communication between the buyers and the sellers. The interactions between the agents are focused on the negations between the buyers and sellers. The underlying system structure is in this case defined by the JADE platform and its features.

4.3.1 General agent topology

The general agent topology is a schematic view of the general agent interactions. It shows which agent types are communicating with each other. The general agent topology of the MAS is shown in Figure 3.

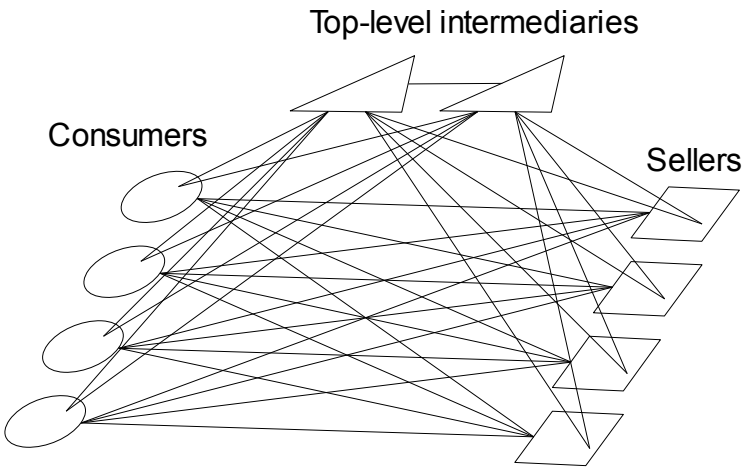

Figure 3: General agent topology of the MAS by N. Capodeici

The general topology consists of costumers, suppliers and top-level intermediaries. All the customer agents can communicate with all the supplier agents. This communication is used to discuss the prices, offers and eventually reach an agreement. The customer agents buy energy in this MAS, the suppliers generate this energy and sell it to the customers. Customer agents do not communicate with each other in this system. The same goes for the supplier agents. The top-level intermediaries are agents that can communicate with all the other agents. These agents are used to handle tasks that are needed for this MAS, but that cannot be done by the customer or supplier agents. The mentioned agent types are explained in the next section.

4.3.2 Agent types

As seen in Figure 3, there are 3 global agent types, Consumers, Suppliers and Top-level intermediaries. One agent does not belong to any of the global types and is listed as Additional agent. Each global agent type consists of a number of different agents, which are listed here.

- Consumers:
  - Consumer: The Consumers are the energy buyers of the system. They buy energy from the suppliers. One or more of these agents can be active during a simulation.
- Suppliers:
  - Prosumer: The Prosumers are agents that produce a small limited amount of energy by wind turbines or solar panels. They also consume their own energy and sell the excess energy to the Consumers. One or more of these agents can be active during a simulation.
  - Genco: The Gencos are agents that produce large amounts of energy and represent the big energy producing companies. They sell this energy to the Consumers. One or more of these agents can be active during a simulation.
- Top-level intermediaries:

    - Time agent: The time agent is used to synchronize and end the bidding rounds and indicates the current time of the day. One or more of these agents can be active during a simulation.
    - Weather agent: The weather agent is used to generate weather forecasts. These are currently generated randomly. The forecasts consist of the temperature, solar power and wind power. These weather forecasts are used by the suppliers and the Balancer agent. One or more of these agents can be active during a simulation.
    - Balancer agent: The Balancer agent is used to guide the rounds and initialize the GUI and keep track of the established contracts. One or more of these agents can be active during a simulation.
    - EH agent: The EH agent is used to handle agent failures. It can kill and restart any agent in the system and informs the other agents if this occurs. Only one of these agents can be active during a simulation.
- Additional agents:
    - Agent Creator: The agent creator is used to create and start all the agents in the MAS. It is terminated after doing so. One or more of these agents can be active during a simulation.

Of all the agents mentioned in this list, at least one of each agent is needed for the system to function. However this does not make the simulation very realistic. More Consumers and suppliers are needed to make the simulation more realistic. Also multiple instances of the Top-level intermediaries and Additional agents can be started during a simulation. The general simulation usually consists of the following agents:

- 2 Agent Creators
- 2 Time agents
- 2 Weather agents
- 2 Balancer agents
- 1 EH agent
- 60 Consumers
- 16 Prosumers
- 6 Gencos

How these agents interact and behave after the system has started, is explained in the next section.

4.3.3 Agent interactions

The different agent types explained in the previous section interact with each other once the system has started. The interactions between all the agent types in the system is visualized in Figure 4.

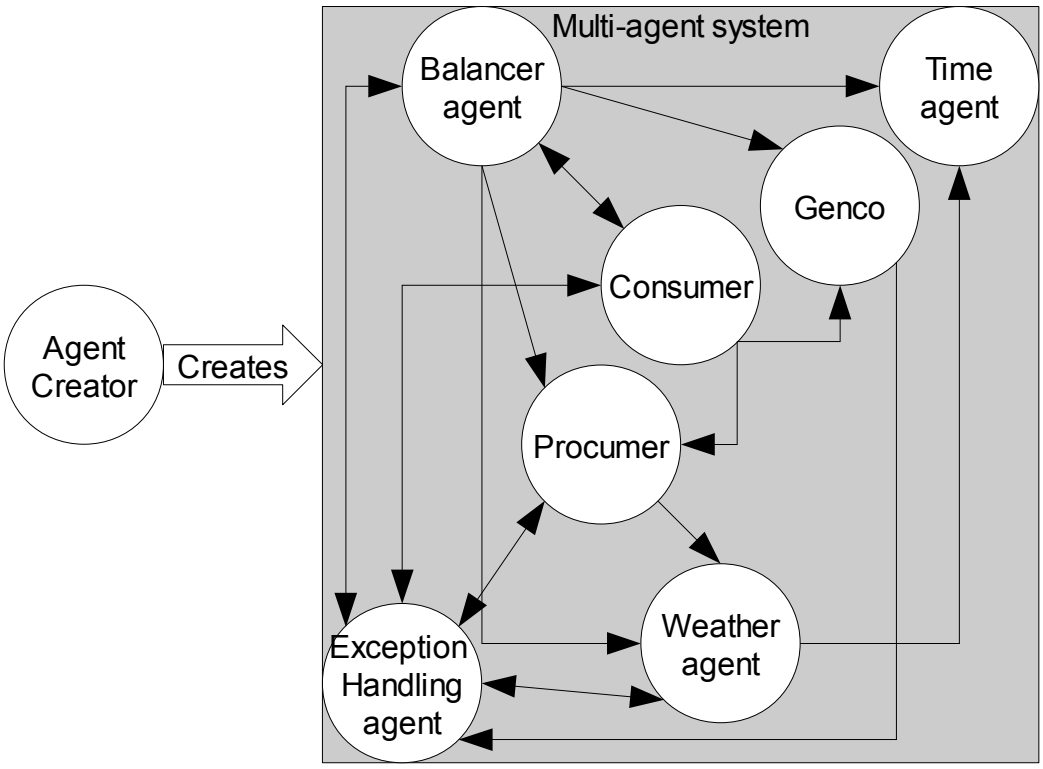

Figure 4: Agent interactions of the different agent types in the MAS

The arrows in this system indicate that an agent contacts another agent. Therefore replies sent by agents are not seen as contacting another agent. In Figure 4 the Balancer agent and the EH are clearly the most important agents of the system, because they interact with most of the other agents in the system.

The contracts that are established between the Consumers and the suppliers at the end of the negotiations, are based on the area that these agents reside in. These areas can be seen as a topological view of the system representing real life geographical areas. The current system supports 6 areas, but as an example 3 areas are used. These areas are bound to host machines, meaning that each area is run on a different host. An example view on these areas and established contracts is shown in Figure 5.

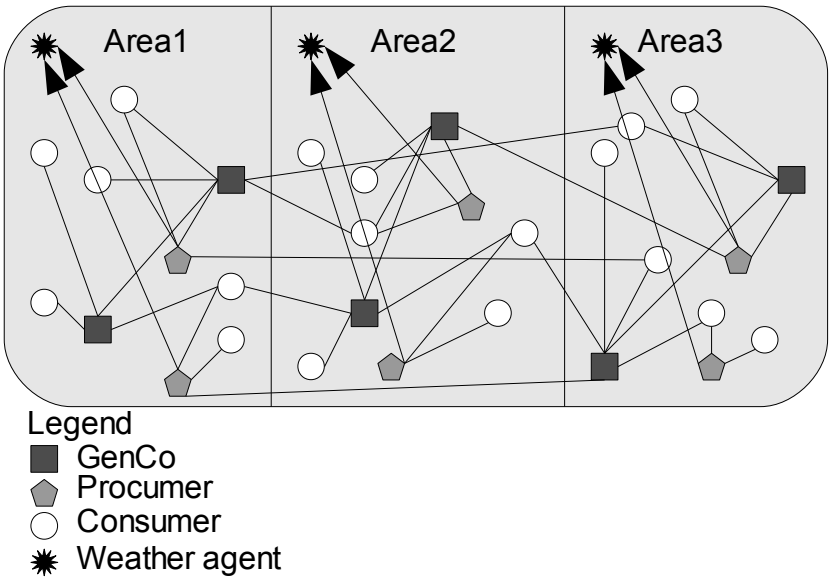

Figure 5: Example of area division and contracts between the different agent types combined with the weather agents for each area.

In Figure 5 there are 3 areas where the Consumers and suppliers reside. The area where the Consumers reside is of importance to the system, because it is taken into account when searching for a supplier. Contracts are established preferably with nearby suppliers, but long distance contracts are possible. The suppliers consist of both Gencos and Prosumers. The energy supply of the Prosumers is based on the weather forecast, which is different for each area. The production capacity of each Prosumer is therefore bound to the weather in the area that the Prosumer resides in, which results in a realistic simulation. The energy supply of the Gencos is based on the expected demand from the Consumers. The behavior of each of the agents is explained in the next section.

4.3.4 Agent behavior

The behavior of each agent is explained in short in this section, a more detailed description is given in chapter 5.1 Class diagram.

Agent Creator

The agent creator is separated from the MAS, because it is only used to start the MAS and its agents. The agent creator is always started first and creates and starts all the other agents in the MAS. The agents are started in the following order:

1. Time agent
2. EH agent
3. Weather agent
4. Gencos
5. Prosumers
6. Consumers
7. Balancer agent

After this the agent creator is done and terminates itself, it does not participate in the MAS.

Time Agent

The time agent only interacts with the other top-level intermediaries, consisting of the Balancer agent and the weather agent. It waits for incoming requests from other agents and replies the current time of day. It does not contact other agents by itself. This time of day is based on the current system time of the host. The day is divided into 6 time slices, ranging 0 to 5.

Weather Agent

The weather agent interacts with most of the other agents, except for the Consumers and the Gencos. The first task of the weather agent is to ask for the current time of day from the time agents. The second task is to wait for incoming requests from other agents, asking for a weather forecast, and reply with the current forecast. The weather forecast calculation is based on the current time of day. The forecast consists of three factors, temperature, solar power and wind power. The weather agent also listens for incoming messages from the EH agent containing the failed agents name and can send messages to the EH agent with the name of a failed agent.

Balancer Agent

The Balancer agent interacts with all the other agents in the system. The GUI is initialized first before any other communication actions are taken. The GUI is only started with the first Balancer agent, the other Balancer agents do not display a GUI, in order to increase performance. After this the first step is to search for all the Suppliers, Genco or Prosumer, and inform them that they can send their name and position and energy production to the Balancer. Once this information is received, the agent is drawn on the GUI. The next step is to search for all the Consumers and inform them that they can send their name, area and energy demand to the Balancer. The third step is to ask each weather agent for the forecast in their area and store it. Each weather forecast is bound to a different area and is displayed in the GUI. Also in this step, the total energy demand is calculated and used to balance the demand and supply, by sending a production threshold to each Genco. This step is concluded by informing all the agents that the negotiating can start. The fourth step is to wait for messages from Consumers that have stipulated a contract with a supplier and update the GUI by showing a direct link between the Consumer and the supplier. Once every Consumer has established a contract, the next step is initiated. The fifth step is to update the GUI with a graph containing selling prices and expectations for each Consumer. After this, restart messages are sent to all the agents if the current time of day is below 6. If this is not the case, the next step is started. The last step is used to send a kill signal to all the other agents, so only the Balancer is still active. The Balancer agent also listens for incoming messages from the EH agent containing a failed agents' name. It can also send a message with an agent name to the EH agent in case of an agent failure. The failed agents are stored and displayed in the GUI. Depending on the agent type, certain additional actions are taken. If the failed agent was a Prosumer, the Balancer sends a special restart message to the newly started Prosumer containing the remaining production capacity. If the agent was a Genco, the Balancer first re-sends the request for information from the Genco and then re-sends the remaining production threshold to the Genco. If the agent was a Consumer, the Balancer re-sends the request for information message and then resend the start message.

Genco

The Gencos interact with most of the other agents in the MAS, except for the time agent, the weather agent and the Prosumers. The first step is to wait for a message from each Balancer, requesting the name, area and energy production and send a reply with this information. The next step is to wait for a message from a Balancer containing the production threshold. The last step is to wait for incoming contracting requests from Consumers, a restart message from a Balancer or a kill message from a Balancer. The contracting requests from Consumers are replied with a proposal price based on the distance between the Genco and the Consumer or a message indicating that the

Genco has sold all available energy if the Genco has reached its production threshold. The Genco can only send messages to the EH agent with the name of a failed agent. It is not able to receive messages from the EH agent, because it does not contact agents by itself and does not need to update the agents.

Prosumer

The Prosumers interact with most of the other agents in the system, except for the time agent and the Gencos. The first step is to ask for a weather forecast from every weather agent, but only the weather forecast of the weather agent that is within the same area as the Prosumer is used. In the second step the Prosumer checks if it has received a special restart message from a Balancer agent, containing the remaining production capacity. If this is the case, this production capacity is used instead of the normal capacity, which is calculated with the weather forecast. After this the Prosumer waits for a message from each Balancer, requesting the name, area and energy production and sends a reply with this information. The next step is to wait for incoming messages from Consumers containing their area and energy demand. If the Prosumer has energy left, it replies with a proposed price, which is based on a starting price and the distance to the Consumer. When a Consumer cancels the negotiations, the Consumers' offer is removed. When an offer from a Consumer is received, the Prosumer proceeds to the next step. The fourth step is to elaborate all the offers that it has received and find the best offer. The other offers are refused and the best offer is only accepted if it is higher than the expected earnings for the Prosumer. If the Prosumer still has energy left, it returns to step three, otherwise it moves on to the final step. The final step is to refuse all offers that are still present and all incoming offers. After this it waits for a restart message from a Balancer or a kill message from a Balancer. The Prosumer also listens for incoming messages from the EH agent containing the failed agents name and can send messages to the EH agent with the name of a failed agent.

Consumer

The Consumers interact with the Gencos, the Prosumers and the Balancer agents. The first step is to wait for a message from each Balancer, requesting the name, area and energy demand and send a reply with this information. The second step is to wait for a start message from one Balancer, after this the negotiations can start. The third step is to search for all the suppliers and contact the Prosumers first. A message is sent to each Prosumer containing the area and the energy demand of the Consumer. It then waits for each Prosumer to reply with a proposal containing a proposed price. When all the Prosumers have replied, the Consumer moves to the next step. The fourth step is to send a message to all the Gencos, containing the area and energy demand of the Consumer. It then waits for each Genco to reply with its area. The nearest Genco is then selected and saved. The next step is to remove the Prosumers that do not produce enough energy for the Consumers' demand. If no Prosumers remain, the Consumer goes to step eight. From the remaining Prosumers, the cheapest Prosumer is selected and the expected price is calculated using the Prosumers' price. The sixth step is to send new offers to the Prosumer until it accepts the offer or the amount of offers exceeds a set limit. The offers are raised with a certain amount each time. If the Prosumer accepts the offer, the Consumer moves to step seven. If the Prosumer cancels the negotiations or if the amount of offers exceeded the limit, the Consumer removes the Prosumer and returns to step five to find a new cheapest Prosumer. If the Consumer receives a message from the Balancer agent, indicating that the auction round time is up, it sends a contracting message to the nearest Genco containing the area and the energy demand and then moves to step seven. The seventh step is to wait for an incoming offer from the nearest Genco containing its price and reply that the offer is accepted with the Consumers area and demand. But only if no contract has been established yet. The contract details are then sent to every Balancer agent containing the Consumers name, the suppliers name, the expected costs and the total price divided by the demand. The Consumer then waits for a restart or a kill message from a Balancer agent. The eighth step is to send a message to every Genco containing

the Consumers area and demand and wait for replies from the Gencos containing their price. The cheapest Genco is then selected and an accept message is sent to that Genco. The contract details are sent to every Balancer agent containing the Consumers name, the suppliers name, the expected costs and the total price divided by the demand and the Consumers goes back to step seven. If the Gencos do not answer or if the auction round time is up, the Consumer contacts the nearest Genco with area and demand and goes back to step seven. The Consumer also listens for incoming messages from the EH agent containing the failed agents name and can send messages to the EH agent with the name of a failed agent.

EH Agent

The EH agent is not very complex, but does communicate with almost all the other agents in the system, except for the time agent. This is because the time agent does not contact any agents by itself, it only waits for requests from other agents. Therefore it cannot detect failed agents and does not need updates on the failed agents. The EH agent has a cyclic behavior, which is repeated until the agent terminates. It first searches for all the Balancer agents, Consumers, Prosumers and weather agents. Then it waits for an incoming message and checks if it is a kill message, in which case it terminates, or if it is a message containing a failed agent. The agent attempts to kill the failed agent if not already dead and restarts the agent with a different name, by adding “-r”. Depending on the failed agents' type, other agents are informed of the failed agent. The following agents are informed in case of these failures:

- Time agent fails: Balancer agents, Consumers, Prosumers and weather agents are informed
- Weather agent fails: Balancer agents and Prosumers are informed
- Balancer agent fails: Consumers and Prosumers are informed
- Prosumer fails: Balancer agents and Consumers are informed
- Genco fails: Balancer agents and Consumers are informed
- Consumer fails: Balancer agents are informed

After this, the process is repeated.

4.3.5 System structure

The system structure of the MAS is highly dependent on the JADE platform and its features. The full system structure is shown in Figure 6.

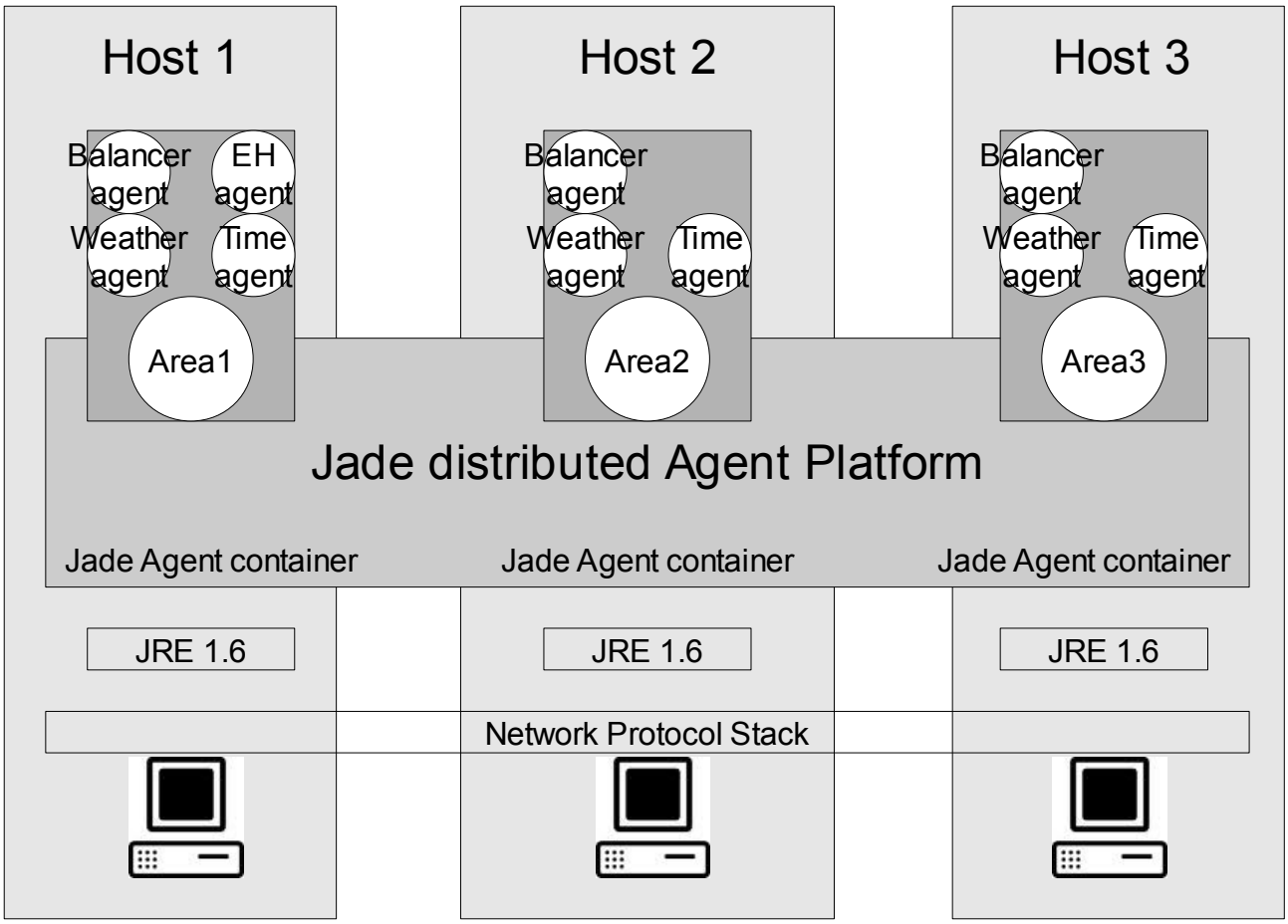

Figure 6: Detailed system structure of the MAS on three hosts

The MAS runs on multiple hosts or workstations. The layered structure of this system on each of the hosts has remained the same. The bottom layer is the Network Protocol Stack, which spans across the network of multiple hosts. On top of this stack is the JRE 1.6 Java library. This contains the basic Java functionality and the Java VM. JADE is positioned on top of the JRE as it is an extension of the basic Java functionality. The JADE platform also spans across the network of multiple hosts, providing a transparent platform, which hides the lower layers. Within the JADE platform there are multiple Agent containers. It is possible to create multiple containers on each host, but in this system only one per host is used. Each container wraps all the agents within, which are parts of the complete MAS discussed in the previous sections. The EH agent is only present on Host 1, because only one instance is needed. Apart from this difference, each host has similar agents. The area division discussed in the previous section, is bound to the system structure. Each area containing the Consumers and Suppliers runs in a container on a different host. Host 1 contains the agents from Area 1, Host 2 from Area 2 and Host 3 from Area 3. A Balancer agent, time agent and weather agent is always present on each host machine.

5 Implementation

For the implementation of the architecture, Java is used in combination with the JADE platform. The latest version of the Java JRE and the JADE platform is used to ensure an up-to-date implementation. To implement and test the system and to make this as easy as possible, the Eclipse development platform is used. This results in the following software being used for the implementation:

- Java JRE 1.6.0.24
- JADE 4.1
- Eclipse version 3.5.2 Java EE IDE for Web Developers

This chapter consists of an explanation of JADE functionality with respect to the implementation of the MAS and contains a detailed description of the classes used in the MAS.

5.1 Class diagram

The different classes that are present in the MAS are explained in this section. First a simplified class diagram is discussed, after which a short description of the basic agent is provided and finally each class is explained separately.

The simplified class diagram of the MAS is shown in Figure 7.

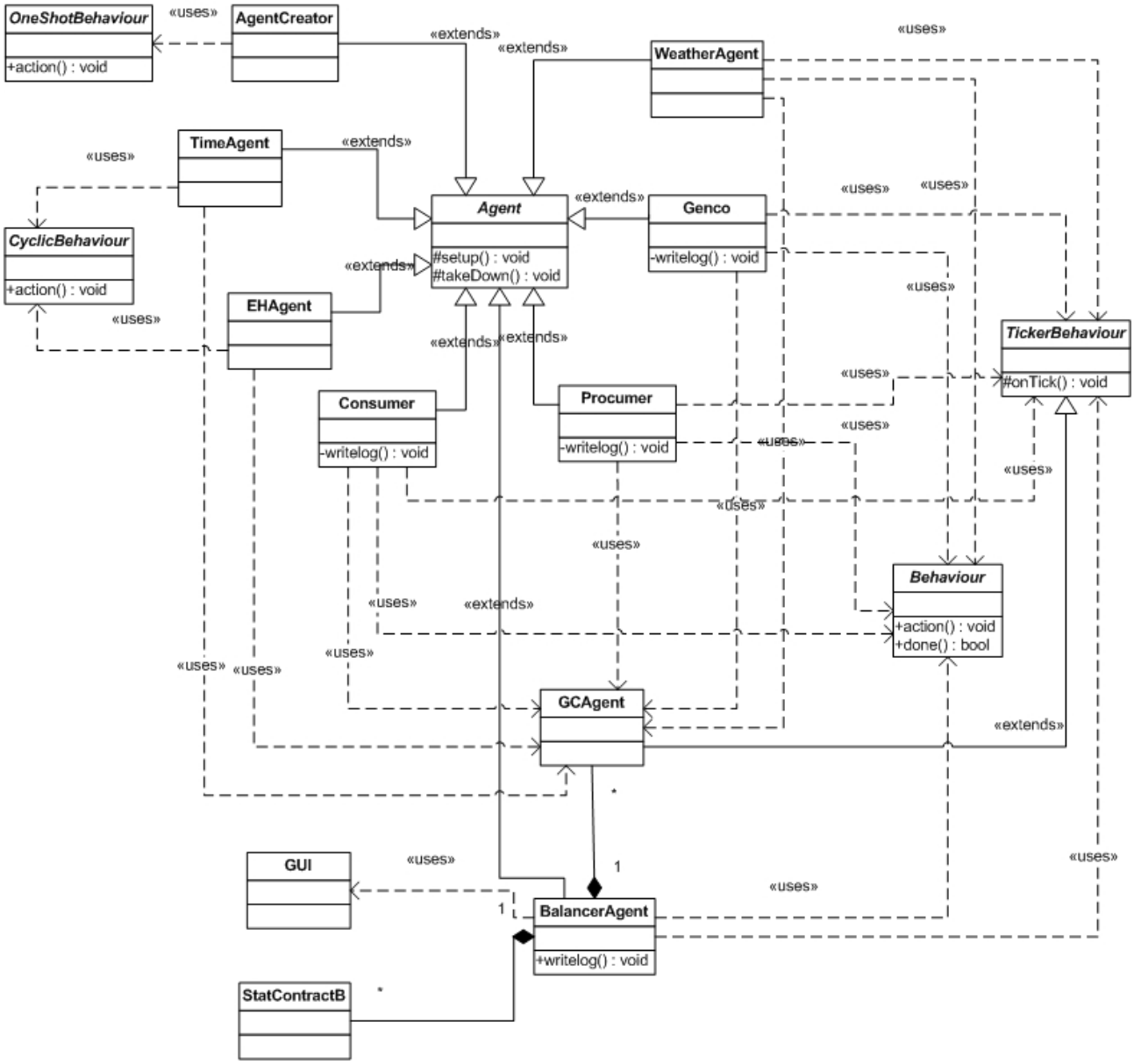

Figure 7: Simplified class diagram of the MAS

This class diagram in Figure 7 has been simplified, in order to focus more on the important characteristics of the MAS and to make it clearer. A very prominent class in this diagram is the Agent class. This is a JADE framework class, which each agent in the system must extend. The default abstract methods of this class are, `setup()` and `takeDown()`. The contents of these methods are explained below.

Other important classes are the behavior classes, which are used by all the agents in the system, these are: Behaviour, CyclicBehaviour, TickerBehaviour and OneShotBehaviour. These classes are also JADE classes and are used to implement a certain type of behavior for an agent. The class Behaviour is a very basic behavior, which constantly runs the code inside and does nothing else. The CyclicBehaviour is a behavior which runs its code constantly in cycles. The TickerBehaviour is similar to the CyclicBehaviour, but uses a timeout between each run. The OneShotBehaviour executes the code within only once.

Another important class that is used by most of the agents is the GCAgent. This is the Garbage collector and its purpose is explained below. Also visible is the Balancer agent, which uses the GUI class to display the information on the screen. The class StatContractB is used by the Balancer to store a contract between a Consumer and supplier. The other classes in the diagram are all agent classes and their implementation are explained in detail below.

5.2 The basic agent

There are some methods that are present in most of the agents in the MAS and are therefore mentioned here. These methods are:

- `protected void setup()`: In the setup method, the agent can register itself to the JADE DF service. Most of the agents create their logging file here as well. The agent behaviors can also be added here, by using `addBehaviour(Behaviour b)`. Within the setup method, there are some other standard methods that are used in most of the agent in the system, these are:
    - `setQueueSize(10000)`: Most of the agents in the system set the maximum message queue size to 10.000. This is done to prevent clogging of the message queue and to make the system more robust and helps implement the refuse requests method mentioned in chapter 3.4.2.
    - `addBehaviour(new GCAgent(this, 10000))`: A behavior that is present in most of the agents is the Garbage collector. 10000 represents 10 seconds in this case. The Garbage Collector is a TickerBehaviour, meaning that, in this case, the code of this behavior is executed every 10 seconds. The Garbage Collector checks the entire message queue of an agent by calling `myAgent.receive()` and removes any messages from that queue that are there for more than 10 seconds. This removes any unused messages and therefore helps implement the refuse requests method mentioned in chapter 3.4.2.
    - `addBehaviour(new TickerBehaviour(this, 2000)`: This behavior is present in most of the agents and is used to check for incoming messages from the EH agent and failures. It is a TickerBehaviour that runs every 2 seconds and searches the message queue for messages with the following performative: `FAILURE`. If the content of the message is a JADE failure message, the agent name is taken from this message and sent to the EH agent. If this is not the case, the agent checks for the agent type by looking at the name of the agent, for instance a Time agent:

`first.startsWith("ta")`, and updates the respective array with the new agent name. This improves the reliability of the system, by supporting the Domain-independent exception handling service mentioned in chapter 3.4.2.

- Each behavior usually consists of the following methods:
  - `public void action()/protected void onTick()`: This is where the main part of the agent code is located and where the behavior of the agent is controlled. The detailed descriptions of the classes explained in this section therefore focus on the content of these methods.
  - `abstract boolean done()`: This method is usually present in a behavior and checks if the behavior is done.

- `protected void takeDown()`: This method is used to terminate the agent itself and de-register itself from the JADE DF service.
- `private void writelog(String s, BufferedWriter out, boolean close)`: This method is used to write information to the logging files. Some of the agents in the system keep their own log.

5.3 Agent Creator

This agent uses a single OneShotBehaviour, because it only runs once. In this behavior the agent first starts a Time agent, by calling: `createNewAgent("ta"+containerNr, "simulation.TimeAgent", null)`. After this a Weather agent and in case the Agent Creator is located in container one, an EH agent. After this an iteration is started, executing 30 times and creating 30 Consumers, 7 Prosumers and 3 Gencos. The name of the Consumers starts with a 'c', the Prosumers with a 'p' and the Gencos with a 'g' each followed by the current iteration number. After this the Balancer agent is started and in case the Agent Creator is located in container one, a "GUI" argument is also sent to the Balancer to enable the GUI on this Balancer. Finally the Agent Creator calls `doDelete()` to kill itself.

5.4 Time Agent

This agent uses the GC behavior and a CyclicBehaviour, because it is constantly checking for incoming messages by calling `myAgent.receive()`. If there is an INFORMATIVE type message, the agent retrieves the system date and time and uses the current seconds and divides this by 10. This results in a number between 0 and 5 indicating the time of day. This number is replied to the sender of the message. If the message was a "DIE"message, the agent calls `doDelete()` to kill itself.

5.5 Weather agent

This agent uses the GC behavior, the EH behavior and a standard Behaviour. In this class there is a switch over 6 cases, based on the current time of day from the Time agent. If the time is 0, the Weather agent first searches the DF for all the Time agents by calling `DFService.search(myAgent, template)`, where `template` has the type `TimeAgent`. After this the Weather agent sends a message to every Time agent requesting the current time of day. The agent then waits for a reply containing the time of day, by calling `myAgent.blockingReceive()`. For each time of day different random weather is generated,

for instance, the temperature and solar power is less in the morning than in the afternoon. The Weather agent checks for incoming INFORMATIVE type messages via `myAgent.blockingReceive()`, containing the "WF" string, requesting the weather and sends back the temperature, solar power and wind power. In case of a “DIE” message, the agent calls `doDelete()` to kill itself.

5.6 EHAgent

This agent uses the GC behavior and has a CyclicBehaviour. The agent first searches the DF for all the Consumers, Prosumers, Gencos, Balancer agents and Weather agents. It then checks for incoming `FAILURE` type messages containing the name of the failed agent, with `myAgent.receive()`. The failed agent is killed via `ac.kill()`, if not already dead and restarted with an added "-r" to its name. The new agent's name is only sent to the agents that need this information, depending on the type of the agent. In case of a "DIE" message, the agent calls `doDelete()` to kill itself.

5.7 Genco agent

This agent uses the GC behavior, the EH behavior and a standard Behaviour. In this class there is a switch over 4 cases, based on the current step as can be seen in Figure 8.

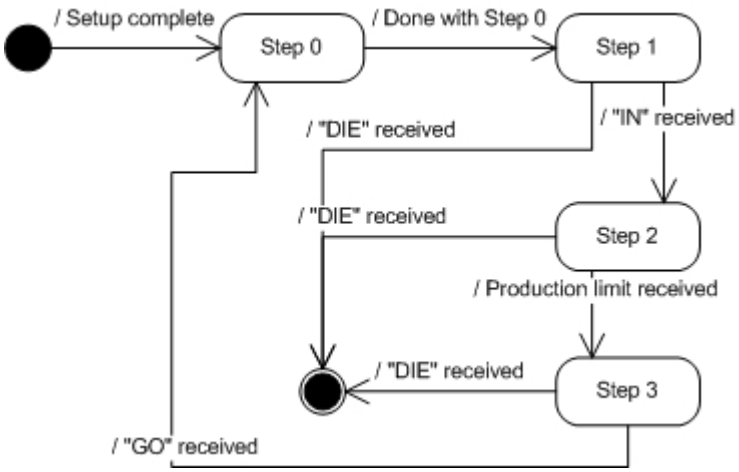

Figure 8: Genco 4 step behaviour

Step 0: The Genco first searches the DF for the EH agent, by calling `DFService.search(myAgent, template)`, where `template` has the type `EHAgent`. After this the agent moves to step 1.

Step 1: The Genco waits for an "IN" signal sent by a Balancer agent. If this has been received, the Genco searches the DF for all the Balancer agents and sends each one a message with content: `msg1.setContent("G" + position + " " + threshold)`, where position is the area the Genco recides in and threshold is the maximum production threshold of the Genco. After this the agent moves to step 2. If a "DIE" signal was received, the agent calls `doDelete()` to kill itself.

Step 2: The Genco waits for a message containing the suggested production limit. If this has been received the Genco adds a random number between 0 and 5 to the production limit and moves to step 3. If a "DIE" signal was received, the agent calls `doDelete()` to kill itself.

Step 3: The Genco waits for different incoming messages. First `INFORM` type messages from Consumers, containing their position and demand can be received. This position is then used in combination with the demand to generate a proposed price. A reply is then sent to the Consumer containing: `reply.setContent(name + " " + proposed_price + " " + n_threshold)`, where name is the name of the Genco, proposed_price is the proposed price and n_threshold is the threshold for the Genco. Also `ACCEPT_PROPOSAL` type messages from Consumers, containing the demand can be received. The demand is subtracted from the production threshold of the Genco. Another message type that can be received, is the `SUBSCRIBE` from Consumers. The Genco replies to this message with its name and position. If a "DIE" signal was received, the agent calls `doDelete()` to kill itself.

5.8 Prosumer agent

This agent uses the GC behavior, the EH behavior and a standard Behaviour. In this class there is a switch over 6 cases, based on the current step as can be seen in Figure 9.

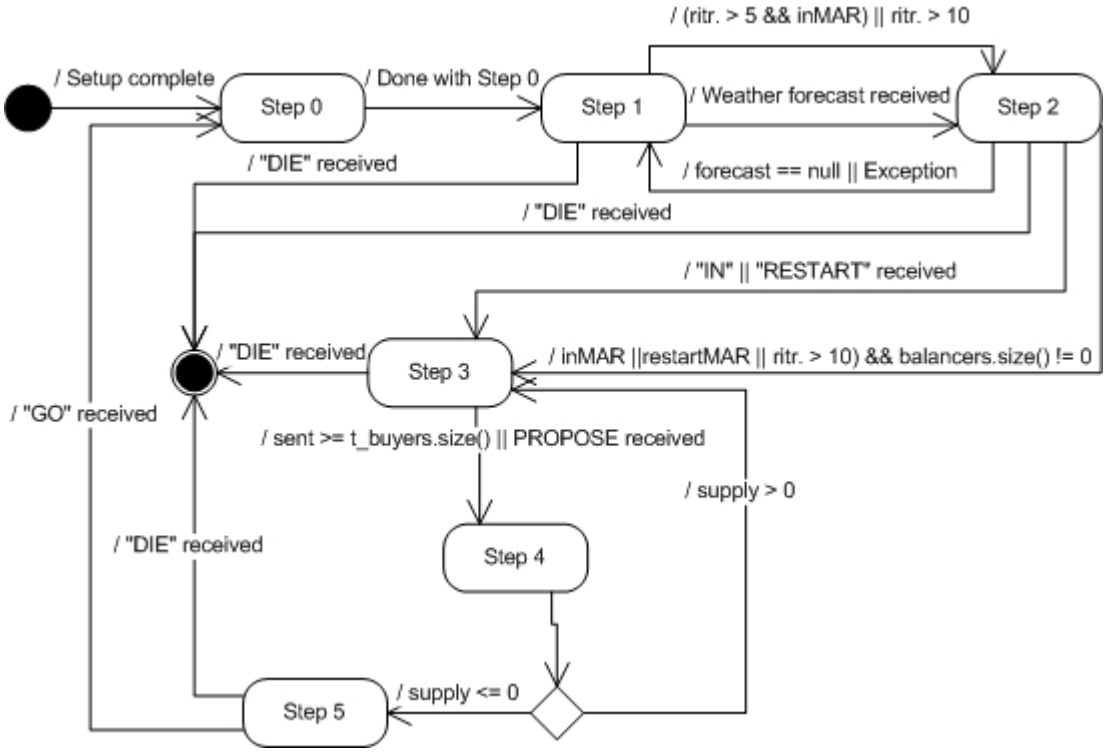

Figure 9: Prosumer 6 Step behaviour

Step 0: The Prosumer first searches the DF for the Weather agents and the EH agent, by calling `DFService.search(myAgent, template)`, where `template` has the type `EHAgent` or `WeatherAgent`. Of the found Weather agents, the agent that has the same number as the container number where the Prosumer resides in, is stored. This represents the area of the agents. After this the agent moves to step 1.

Step 1: The Prosumer sends a message, containing "WF", to the Weather agent requesting a weather forecast. The Prosumer then waits for a few different message types. The first is a reply from the Weather agent containing a temperature, wind power and solar power. After this the Prosumer moves to step 2. The second message type is a message from a Consumer requesting negotiations. The Prosumer replies with an `CANCEL` message containing "NO". The third type is an "IN" or "RESTART" message from a Balancer, indicating that the negotiations have started. A "RESTART" message also contains a remaining supply. The supply is stored, as well as a Boolean

indicating the received message. The last type is a "DIE" message, where the Prosumer will call `doDelete()` to kill itself.

Step 2: The Prosumer will calculate the new supply from the weather forecast with: `supply = (int) (supply * constant) + 1`, where supply is the solar or wind power and constant is a random number between 0 and 1. The agent now waits for an "IN" or "RESTART" message if not already received in step 1. If this message is received, the Prosumer searches the DF for all the Balancer agents. If it was an "IN" message, it sends a message to each Balancer, containing: `msg1.setContent("P" + position + " " + supply)`, where position is the area that the Prosumer resides in and supply is the total production limit of the Prosumer. If it was a "RESTART" message, the Prosumer takes the remaining supply from the message and stores it as `supply`. After this the Prosumer moves to step 3. If a "DIE" message was received, the Prosumer calls `doDelete()` to kill itself. If any other message was received, the Prosumer sends a "MOV" message to every Balancer, to indicate that it is waiting.

Step 3: The Prosumer searches the DF for every Consumer, but only if this is the first run of step 3. Next the Prosumer checks for a few different message types. If an `INFORM` message is received from a Consumer, containing its area, the Prosumer calculates a proposed price with: `float proposed_price = (float) (((Math.abs(b_pos - position) + 1) * c_TSO) + starting_price)`, where b_pos is the area of the Consumer, position is the area of the Prosumer, c_TSO is a random float constant and starting_price is the starting price of the Prosumer. If the Prosumer still has some supply left, it sends an `PROPOSE` reply with the proposed price and its remaining supply, in the other case an `CANCEL` message is sent. If the amount of sent replies is larger than or equal to the amount of Consumers in the system, the Prosumer moves to step 4. If an `ACCEPT_PROPOSAL` message is received from a Consumer, containing its name and energy demand, the Prosumer subtracts this demand from its supply. If an `PROPOSE` message is received, containing an offer from a Consumer, the Prosumer stores the offer and moves to step 4. If an CANCEL message is received, containing an offer, the Prosumer removes that offer. If a "DIE" message is received, the Prosumer calls `doDelete()` to kill itself.

Step 4: The Prosumer first selects the best offer from the list of offers. Then the Prosumer sends every agent from the offers list an `REFUSE` message, containing "NO", except for the best offer. If this offer is less than expected and the number of refusals is smaller than the maximum, the best offer is refused as well. Otherwise, this agents gets an `ACCEPT_PROPOSAL` message containing "YS". If after this the Prosumer has no supply left, it sends an `CANCEL` message, to every Consumer from the offers list and moves to step 5. If it does have supplies left, it moves back to step 3.

Step 5: The Prosumer sends an `CANCEL` message once to every Consumer in the system. It then waits for incoming messages. If a "DIE" message is received, the Prosumer calls `doDelete()` to kill itself. If an offer from a Consumer is received, the Prosumer replies with an `CANCEL` message containing "NO".

5.9 Consumer agent

This agent uses the GC behavior, the EH behavior and a standard Behaviour. In this class there is a switch over 8 cases, based on the current step as can be seen in Figure 10.

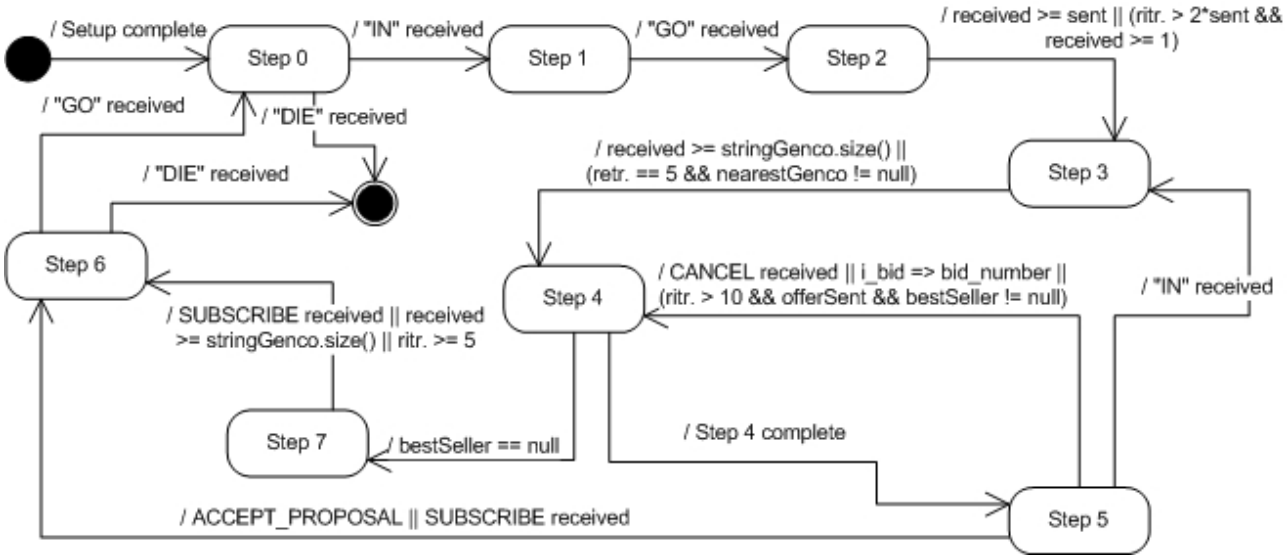

Figure 10: Consumer 8 Step behaviour

Step 0: The Consumer first searches the DF for the EH Agent once. It then waits for incoming messages. If an “IN” message is received from a Balancer agent, the Consumer searches the DF for every Balancer agent and sends a message to each Balancer agent, containing `msg1.setContent("B" + position + " " + p_demand)`, where position is the area that the Consumer resides in and p_demand is the energy demand of the Consumer. It then moves to step 1. If a "DIE" message is received, the Consumer kills itself. If another message is received, the Consumer sends a "MOV" message to every Balancer agent.

Step 1: The Consumer waits for a "GO" message from a Balancer, after which it moves to step 2.

Step 2: The Consumer searches the DF for every "Seller" type agent once, this includes both the Prosumers and Gencos. These agents are stored in `sellerAgents` and a message containing the area and energy demand is sent to every Prosumer once. The Consumer then waits for incoming messages. If an `PROPOSE` message is received, containing an offer from a Prosumer, the offer is stored. If an offer is received from every Prosumer, the agent moves to step 3. If an `CANCEL` message is received from a Prosumer, this agent is removed from the list of Prosumers. If step 2 had been run more than twice the amount of Prosumers and there is at least one offer, the Consumer also moves to step 3.

Step 3: The Consumer first sends a message to every Genco once, containing its area and energy demand. It then listens for incoming messages. If a message is received from a Genco containing its name and area, the Consumer calculates the distance between the Genco and itself. Only the Genco that has the smallest distance is selected as `nearestGenco`. If a message has been received from every Genco, the Consumer moves to step 4. If a message is received from a Prosumer, containing "NO". The Prosumer is removed from the list of Prosumers. If no message is received for 5 times and there is already a `nearestGenco` selected, the Consumer also moves to step 4, otherwise it re-sends a message to every Genco, containing its area and energy demand.

Step 4: If the Consumer is not forced to contact a Genco, the Consumer checks the supply of each Prosumer and removes those that do not have enough supply for the demand of the Consumer. Of the other Prosumers, the one with the lowest price will be selected as `bestSeller`. After this it calculates the expected price to pay for the power demand: `((float) bestPrice + (bestPrice * ((g.nextInt(20) + 1) / 100))) / demand`. After this, or if the Consumer is `forced` to contact a Genco, the Consumer moves to step 5, or if no `bestSeller` is

found, the Consumer moves to step 7.

Step 5: The Consumer sends an `PROPOSE` message to the `bestSeller` once, containing a bid of height `stake` and its energy demand. If after 10 runs no reply has been received from the `bestSeller`, the Consumer deletes the Prosumer from the list and returns to step 4. After this the Consumer checks for incoming messages. If an `ACCEPT_PROPOSAL` message is received from a Prosumer, the Consumer replies with an `ACCEPT_PROPOSAL` message containing its demand and moves to step 6. If an `REFUSE` message is received from a Prosumer, the Consumer replies with a new `PROPOSE` message and a higher bid, increased with a random constant `stake`. If too many refusals have been received, the Consumer removes the `bestSeller` from the list and replies with an `CANCEL` message, containing "NO", and moves back to step 4. If an `CANCEL` message is received from a Prosumer, the Consumer removes the `bestSeller` from the list and returns to step 4. If an `SUBSCRIBE` message is received from a Balancer agent, the Consumer sends an `INFORM` message to the `nearestGenco`, containing its area and demand. After this it moves to step 6. If an "IN" message is received from a Balancer agent, the Consumer replies its area and demand and moves back to step 3.

Step 6: If a contract was established with a Prosumer or Genco, the Consumer sends a message to all the Balancer agents containing the contract details: `name + " " + bestSeller + " " + expected + "X" + ((bestPrice + new_stake) / demand) + "Y" + demand)`. If no contract was established with a Genco and the Consumer is forced, it sends an `INFORM` message to the `nearestGenco`, with its area and energy demand. After this the agent waits for incoming messages. If a "DIE" message is received, the agent kills itself. If an `PROPOSE` message is received from the `nearestGenco` and the demand is still above 0, the Consumer stores the `nearestGenco` Genco in `bestSellerG`. An `ACCEPT_PROPOSAL` message is sent to this Genco with the Consumers area and demand and an INFORM_REF message is sent to every Balancer with the contract details. Demand is set to 0 and `forced` is now false. If an `REQUEST_WHEN` message is received from a Balancer agent, the Consumer waits for 1.5 seconds and then re-sends a reply with the contract details. If an `ACCEPT_PROPOSAL` message is received from a Seller, the Consumer sends back an `CANCEL` message.

Step 7: The Consumer clears the message queue once and if no contract is `forced` and established, sends an `INFORM` message to every Genco once, containing area and demand. If after 5 runs, no Genco has replied, the Consumer sends a message to the `nearestGenco`, containing area and demand and sets `forced` to true and moves to step 6. After this the Consumer waits for incoming messages. If an `PROPOSE` message is received from a Genco, the Consumer checks if this Genco is the cheapest, in which case it is stored. If a proposal has been received from every Genco (`received >= stringGenco.size()`), an `ACCEPT_PROPOSAL` message is sent to the cheapest Genco, containing area and demand. The Balancer agents are informed of the contract details via an `INFORM_REF` message and demand is set to 0. After this the agent moves back to step 6. If an `SUBSCRIBE` message is received from a Balancer agent, the Consumer sends an `INFORM` message to the `nearestGenco` containing area and demand, and moves back to step 6. If an `ACCEPT_PROPOSAL` message is received from a Prosumer, the Consumer replies with an `CANCEL` message. If any other message is received, the Consumer re-sends the contract details to all the Balancer agents.

5.10 Balancer agent

This agent uses the GC behavior, the EH behavior and a standard Behaviour. In the `setup()`

method of the Balancer agent, the agent waits for a while before starting, this is done make sure that all the Balancer agents start at the same time. The agent waits for the next exact half minute before starting: (`Math.ceil((double)oldtime / 30000)*30000)`, where `oldtime` is the current time. After this the Balancer moves to the Behaviour, where there is a switch over 7 cases, based on the current step as can be seen in Figure 11.

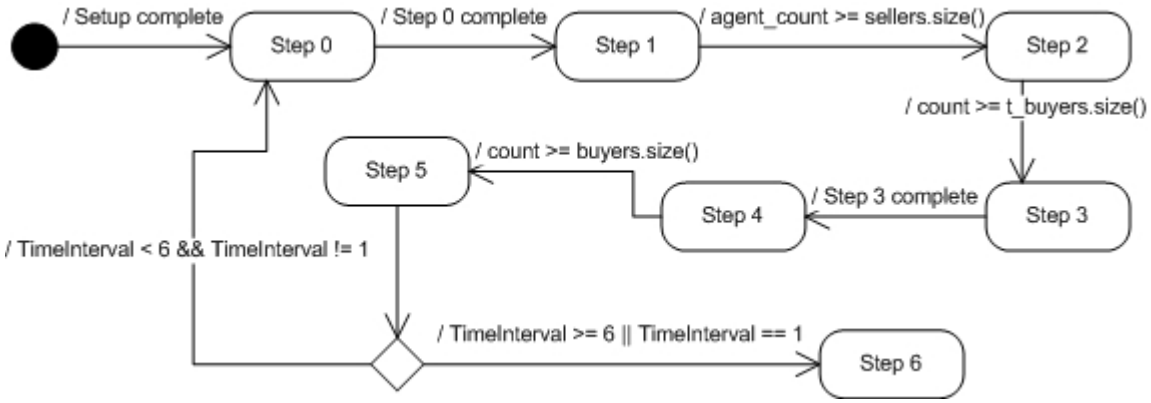

Figure 11: Balancer agent 7 Step behavior

Step 0: The Balancer searches the DF for all the Weather agents, Time agents, the EH agent and other Balancer agents. It then moves to step 1.

Step 1: The Balancer searches the DF for every Seller once, which includes Prosumers and Gencos. Each Seller is sent an `INFORM` message once, containing "IN". The Balancer then waits for incoming messages. If an `INFORM` message from a Genco or a Prosumer is received containing their area and name, the Balancer stores this agent in `gencos` or `prosumers`. Also the supply and name of each Prosumer is stored in `agentSupplies`. After this the agent is drawn in the GUI, by calling: `myGUI.drawAgent(t_position, sa_name, agent_count)`, with area, agent name and agent count. If a reply has been received from each Genco and Prosumer, the Balancer moves to step 2. If any other message is received, the Balancer sends a reply containing "OK". If no messages are received for 9 runs, the Balancer re-sends the messages to the Sellers.

Step 2: The Balancer searches the DF for every Consumer once. Each Consumer is sent an `INFORM` message once, containing "IN". The Balancer then waits for incoming messages. If an `INFORM` from a Consumer containing its area and demand, the Balancer stores this agent in `buyers`. After this the agent is drawn in the GUI, by calling: `myGUI.drawAgent(t_position, sa_name, agent_count)`, with area, agent name and agent count. If a reply has been received from each Consumer, the Balancer moves to step 3. If any other message is received, the Balancer sends a reply containing "IN".

Step 3: The Balancer calculates the total energy demand of all the Consumers by adding their individual demands. The average needed energy production per Genco and per Prosumer is calculated, by: `Sg = (int) ((demand[0] * kg) / (gencos.size())) + 1` and `Sp = (int) ((demand[0] * kp) / (prosumers.size())) + 1`. For each Prosumer the stored energy production and needed production are compared and these differences are summed up over all the Prosumers. This so called `deficit` is divided by the number of Gencos and added up to the needed energy production of the Gencos: `deficit = deficit / gencos.size(); Sg += deficit`. Each Genco is then sent an `INFORM` message, containing this needed energy production, or `Sg`. It is also added to the `agentSupplies` list together with the agent name. After this the Balancer sends a message containing "WF", to every Weather agent. Next every Consumer is sent a message containing "GO". The Balancer now enters a loop, in which it waits for all the

incoming weather forecasts. If a message containing a weather forecast is received, the Balancer uses the weather forecast and the area of the Weather agent and displays it in the GUI: `myGUI.add(myGUI.new DrawTemp(temperature, solar, wind, area))`. If an `INFORM_REF` message is already received from a Consumer containing contract details, the Balancer replies with an `REQUEST_WHEN` message containing "WT". If other or no messages are received for 5 runs, the Balancer re-sends the messages to the Weather agents. After the loop the Balancer moves to step 4.

Step 4: The Balancer waits for incoming messages. If an `INFORM_REF` message is received from a Consumer, containing contract details, the Balancer stores the contract details from the message and replies with an `INFORM_REF` message "OK". The Balancer then checks whether the received contract was already received earlier on, in which case the contract details are discarded. The next check is to make sure that a Consumer only has one contract, if this is not the case, the contract details are also discarded. If the contract is not discarded, the details are stored in an object: `StatContractB`. These objects are added to the GUI and the GUI is refreshed: `myGUI.statsB.addElement(scb); myGUI.refreshGUI(t1, t2)`. The list `agentSupplies` is updated by subtracting the Consumers' demand from the total energy production of the Genco or Prosumer. If all of the Consumers have sent their contract details, the Balancer moves to step 5.

Step 5: The Balancer adds a `MouseListener` and the `drawSkeletonGraph` to the GUI. After this the Balancer waits 2 seconds and then moves to step 6.

Step 6: The Balancer sends a "DIE" message to every agent in the system. It then waits indefinitely, until it is manually killed.

5.11 GUI class

The GUI class contains all the GUI related code. It is created by the Balancer agent, but only on the first host of the MAS. The GUI class contains a number of methods, some of which are called by the Balancer agent. Some of the important methods are listed here together with their purpose:

- `public GUI()`: Called by the Balancer agent. In this constructor the `Jframe` of 1150 by 800 is created and displayed, a `MyCanvas` is created and all the Vectors are initialized.
- `public void clear()`: Called by the Balancer agent. This method is used to fully clear and repaint the GUI, by calling `(gui.getContentPane()).removeAll()` and `gui.repaint()`.
- `public void add(JComponent comp)`: Called by the Balancer agent and the GUI itself. This method can add a new `JComponent` to the GUI, by calling `gui.getContentPane().add(comp)` and `gui.repaint()`.
- `public void refreshGUI(String t1, String t2)`: Called by the Balancer agent. This method is used to add a contract line between agent '`t1`' and agent '`t2`'. The locations of both agents are searched using: `((GUI.Contract) contractNet.get(i)).getPosx(t1)`. The locations are used to create a new `QuadPoints` line: `new QuadPoints(px, py, pxx, pyy)`, which is added to a `lineContract` Vector.
- `public void drawAgent(int pos, String name, int ac)`: Called by the Balancer agent. This method is used to add agent '`name`' in area '`pos`' on the GUI. '`ac`' is

the current agent count and is used to find the correct data in the Vectors. A Consumer is magenta, a Supplier is blue and a Genco is red. A switch is used depending on the area of the agent, to generate a random position within the area: `generator.nextInt(Wstep) + CX + 5`. A new `Contract` object is made using this data: `new Contract(x.get(agent_count), y.get(agent_count), gui_string.get(agent_count))`, and is added to the Vector `contractNet`. After this the GUI is repainted.

There are also some other classes located within the GUI class. Most of these classes are `JComponent` classes and are added to the GUI, using the `add(JComponent comp)` method. These `JComponent` classes each have a `paint(Graphics g)` method, which is called upon repaint and handles the drawing. These classes are listed below:

- `public class DrawTemp`: This class is created by the Balancer agent and is used to draw the information of each Weather agent in the GUI. The Balancer agent calls the constructor, `public DrawTemp(int t, int s, int w, int c)`, where '`t`' is the temperature, '`s`' is the solar power, '`w`' is the wind power and '`c`' is the count of the current area.
- `public class DrawDeadAgents`: This class is created by the Balancer agent and is used to draw the names of the dead agents in the GUI. The Balancer agent calls the constructor, `private DrawDeadAgents()` and also the method `private void updateDeadAgents(Vector<String> failedAgents)`. This method is used to update the Vector `deadAgents`.
- `public class MyCanvas`: This class is created by the GUI and is used to draw the box where the areas are displayed onto the GUI.
- `public class AgentPositioning`: This class is created by the GUI and is used to draw all the agents onto the GUI.
- `public class drawContractLine`: This class is created by the GUI and is used to draw all the contract lines onto the GUI.
- `public class drawSkeletonGraph`: This class is created by the GUI and is used to draw the empty graph and its axis onto the GUI.
- `public class drawLineStat`: This class is created by the GUI and is used to draw the graph of a selected agent, with expected price in red and real price in black.

This final set of classes is used to store information on the agents position and the contract lines and to handle the mouse clicks:

- `public class Contract`: This class is created by the GUI and is used to store the agents in the system. It has a '`posx`', '`posy`' and '`agentName`' as data fields, to store agent position and name.
- `public class QuadPoints`: This class is created by the GUI and is used to store the contract lines in the system. It has a '`posx1`', 'posy1', '`posx2`', '`posy2`' as data fields, to store beginning and end positions.
- `class MyMouseListener extends MouseAdapter`: This class is created by the Balancer agent after the simulation has finished and is used to listen for mouse clicks. If `public void mouseClicked(MouseEvent evt)` is called, the '`x`' and '`y`' position

are used in `((Contract) contractNet.get(i)).retrieveName(x, y)` to retrieve the agent that was selected. After this, `public void drawGraph(String n)` is called with the agent name 'n'. `(StatContractB) statsB.get(i)` is used to get the contract details and `gui.getContentPane().add(new drawLineStat())` is called to add the graph.

6 Evaluation

The evaluation of the new MAS is centered around the two main factors that have been the focus of this research. The scalability and reliability of the MAS. These factors are tested according to predefined test cases. These test cases are defined according to evaluation questions. These questions correspond to the methods that have been implemented in the system and are used to test these methods effectiveness. The evaluation questions are measured by using certain metrics. The metrics are divided into two main categories: [5]

- those related to system parameters
- those related to coordination mechanisms

System metrics include system related performance measurements, such as wall-clock times and CPU and memory usage. Coordination metrics are more related to message performance on the entire MAS, such as time to reach convergence, or the average response time of agents. The different metrics are usually combined, by summing them up or adding weights, in order to obtain a single value which may be compared.

The test cases, as well as the results of these tests are listed in this chapter. Most of the tests are conducted on the old MAS as well as the new MAS, in order to compare performance on both main factors. The test cases for each of the main factors are explained in the following sections.

6.1 Evaluating scalability

To evaluate the scalability of the old MAS and the new MAS and to compare their performance, some test cases can be used. These test cases are based on earlier research on scalability measurements of Multi-agent systems in general. “The scalability of a Multi-agent system depends on whether the worst-case performance of the system (i.e. its overall algorithmic complexity) is bounded by a polynomial function of the load” [3]. The scalability of the system can also be viewed as the ratio between performance and resources. As the available resources increase, the performance should increase [1], which is shown in Figure 12.

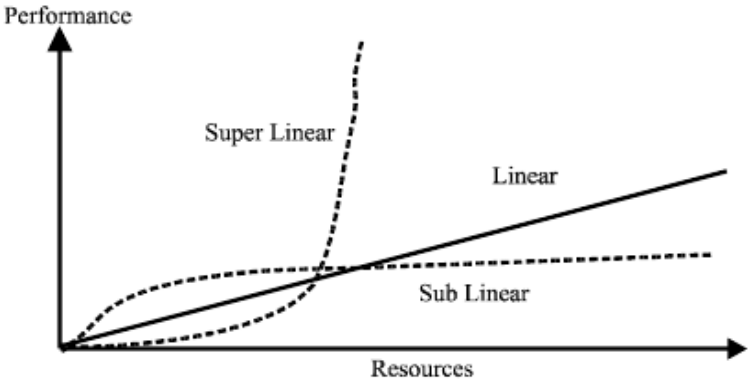

Figure 12: Scalability functions [1]

In this figure three types of scalability can be seen, Sub Linear, Linear and Super Linear, where Sub Linear is the worst performance increase per resource and Super Linear is the best performance increase per resource.

For the performance based on the load of the system, are 4 types of scalability. Exponential, Polynomial, Linear and Sub Linear, where Sub Linear is the best performance per load and Exponential is the worst performance per load.

To measure the scalability and effectiveness of the methods that have been implemented, some evaluation questions have to be answered. The questions are answered by corresponding test cases.

Evaluation questions:

- How many agents can the MAS handle before it becomes unstable? [1][3]
- What is the scalability of the new MAS compared to the old MAS, in terms of performance per load?[3]
- What is the scalability of the new MAS with respect to performance increase per resource? [1]
- What is the performance overhead of critical agent replication? [7]
- What is the performance overhead of restarting an agent in the system through the EH agent?

To answer these evaluation questions, certain metrics are used, these metrics correspond to the categories discussed earlier. The categories are:

Coordination metrics:

- Total number of messages transferred between agents [5]
- Number of agents in the system [4] [3] [5]
- Number of replicas of top level intermediaries [7]
- Number of hosts

System metrics:

- Time to reach convergence [5]/Number of ticks(process/communication cycles) that have passed. [4] (Ticks = milliseconds * 10.000)
- CPU and memory usage [5]

6.1.1 Test cases

The test cases are listed in this section. These test cases are each based on an evaluation question. The old MAS is only used in two test cases, because comparisons cannot be made on the EH agent, the agent replication or the performance increase per resource.

Test Case 1: How many agents can the MAS handle before it becomes unstable?		
How to	Increase the amount of agents on the old and the new MAS until it becomes unstable	
	MAS 1	**MAS 2**
Type	Normal MAS	MAS by N. Capodieci

Test 1	6 hosts	1 host
Metrics	• Number of agents in the system • Number of hosts • Total number of messages transferred between agents • Number of ticks that have passed to reach convergence • CPU and memory usage	

Test Case 2: What is the scalability of the new MAS compared to the MAS by N. Capodieci, in terms of performance per load?		
How to	Increase amount of Consumer and supplier agents on the old and the new MAS	
	MAS 1	**MAS 2**
Type	Normal MAS	MAS by N. Capodieci
Test 1	6 hosts, 58 agents (30c, 7p, 3g, 6b, 6t, 6w)	1 host, 43 agents (30c, 7p, 3g, 1b, 1t, 1w)
Test 2	6 hosts, 98 agents (60c, 14p, 6g, 6b, 6t, 6w)	1 host, 83 agents (60c, 14p, 6g, 1b, 1t, 1w)
Test 3	6 hosts, 178 agents (120c, 28p, 12g, 6b, 6t, 6w)	1 host, 163 agents (120c, 28p, 12g, 1b, 1t, 1w)
Metrics	• Number of agents in the system • Number of hosts • Total number of messages transferred between agents • Number of ticks that have passed to reach convergence • CPU and memory usage	

Test Case 3: What is the scalability of the new MAS with respect to performance increase per resource?	
How to	Increase the amount of hosts, while keeping the amount of agents the same
Type	Normal MAS
Test 1	1 host, 99 agents (60c, 14p, 6g, 6b, 6t, 6w, 1e)
Test 2	3 hosts, same agent amount
Test 3	6 hosts, same agent amount
Metrics	• Number of agents in the system • Number of hosts • Total number of messages transferred between agents • Number of ticks that have passed to reach convergence • CPU and memory usage

Test Case 4: What is the performance overhead of restarting an agent in the system through the EH agent?	
How to	Kill each agent type once, to initiate a restart of that agent

Type	Normal MAS
Test 1	1 host, 84 agents (60c, 14p, 6g, 1b, 1t, 1w, 1e)
Test 2	1 host, same agent amount, kill a Balancer agent
Test 3	1 host, same agent amount, kill a Weather agent
Test 4	1 host, same agent amount, kill a Time agent
Test 5	1 host, same agent amount, kill a Consumer
Test 6	1 host, same agent amount, kill a Procumer
Test 7	1 host, same agent amount, kill a Genco
Metrics	• Number of agents in the system • Number of hosts • Total number of messages transferred between agents • Number of ticks that have passed to reach convergence • CPU and memory usage

Test Case 5: What is the performance overhead of critical agent replication?	
How to	Compare the performance of the MAS with different amounts of replicas on six hosts.
Type	Normal MAS
Test 1	6 hosts, 84 agents (60c, 14p, 6g, 1b, 1t, 1w, 1e), 0 replicas
Test 2	6 hosts, 90 agents (60c, 14p, 6g, 3b, 3t, 3w, 1e), 2 replicas
Test 3	6 hosts, 99 agents (60c, 14p, 6g, 6b, 6t, 6w, 1e), 5 replicas
Metrics	• Number of agents in the system • Number of hosts • Total number of messages transferred between agents • Number of ticks that have passed to reach convergence • CPU and memory usage

6.2 Evaluating reliability

The evaluation of the reliability of the old and new MAS, differs somewhat from the scalability evaluation. The metrics focus less on performance and more on failures and robustness of the system. Test cases have been defined to measure these metrics. These test cases are also based on earlier research on reliability measurements of Multi-agent systems in general. Mentioned earlier in section 3.4.2, the following degrees of fault tolerance are proposed by [8]:

- Full fault tolerance, where the system continues to operate without significant loss of functionality or performance even in the presence of faults.
- Graceful degradation, where the system maintains operation with some loss of functionality or performance.
- Fail-safe, where vital functions are preserved while others may fail.

The goal of the new MAS is to achieve Graceful degradation. To evaluate the reliability of the old and the new MAS, some evaluation questions have been devised.

Evaluation questions:

- How effective is the EH agent in preventing complete system failure, when a certain agent fails? Related to [10]
- How effective is critical agent replication in preventing complete system failure, when a top-level intermediary fails? Related to [11]
- What is the impact of a queue limit and garbage collector on agent thrashing? Related to [9]

To answer these evaluation questions, certain metrics are used, these metrics correspond to the categories discussed earlier. The categories are:

Coordination metrics:

- Number of agents in the system [4] [3] [5]
- Replication Rate = Number Of Extra Replicas/Number Of Agents[11]
- Success Rate(simulations that did not fail) = Number Of Successful Simulations/Number Of Simulations[11]
- Number of hosts

System metrics:

- Time to reach convergence [5]/Number of ticks(process/communication cycles) that have passed. [4] (Ticks = milliseconds * 10.000)
- CPU and memory usage [5]

The test cases are discussed in the next section.

6.2.1 Test cases

The test cases are listed in this section. These test cases are each based on an evaluation question. The old MAS is not used in each test case, because the effectiveness of some methods can be tested on the new MAS. This is done by running the new MAS with the same amount of agents as the old MAS and on only one host.

Test Case 1: How effective is the EH agent in preventing complete system failure, when a certain agent fails?		
How to	Simulate the presence of faults, by randomly stopping a thread of each agent type twice.	
	MAS 1	**MAS 2**
Type	MAS with EH agent	MAS without EH agent
Test 1	1 host, 84 agents (60c, 14p, 6g, 1b, 1t, 1w, 1e)	1 host, 83 agents (60c, 14p, 6g, 1b, 1t, 1w) (Simulates MAS by N. Capodieci)
Test 2	6 hosts, 99 agents (60c, 14p, 6g, 6b, 6t, 6w, 1e)	6 hosts, 98 agents (60c, 14p, 6g, 6b, 6t, 6w)
Metrics	• Number of agents in the system • Number of hosts • Success Rate(simulations that did not fail) = Number Of Successful Simulations/Number Of Simulations	

Test Case 2: How effective is critical agent replication in preventing complete system failure, when a top-level intermediary fails?	
How to	Simulate the presence of faults, by randomly stopping a thread of each top-level intermediary twice.
Type	MAS without EH agent
Test 1	1 replica, 1 host, 83 agents (60c, 14p, 6g, 1b, 1t, 1w) (Simulates MAS by N. Capodieci)
Test 2	3 replicas, 3 hosts, 89 agents (60c, 14p, 6g, 3b, 3t, 3w)
Test 3	6 replicas, 6 hosts, 98 agents (60c, 14p, 6g, 6b, 6t, 6w)
Metrics	• Number of agents in the system • Number of hosts • Replication Rate = Number Of Extra Replicas/Number Of Agents • Success Rate = Number Of Successful Simulations/Number Of Simulations

Test Case 3: What is the impact of a queue limit and garbage collector on agent thrashing?		
How to	Simulate an agent (Spam agent) that aggressively sends requests/messages to the Balancer agent.	
	MAS 1	**MAS 2**
Type	Normal MAS	MAS by N. Capodieci
Test 1	1 host, default amount of agents	1 host, default amount of agents
Test 2	1 host, same amount of agents, Spam agent active	1 host, same amount of agents, Spam agent active
Metrics	• Number of agents in the system • Number of hosts • Number of ticks that have passed to reach convergence • CPU and memory usage	

6.3 Results

The results of the MAS and the test cases that have been performed are listed in this chapter. The systems that were used for testing purposes are listed in the next section. The section after this explains how the GUI works and what it displays. The last two sections deal with the results from the scalability test cases and the reliability test cases.

6.3.1 Testing system

The experiments were conducted on a used 1000 MBPS Ethernet network of the following machines:

HP DC7800 small form factor with:

- Intel Core2 Duo CPU E6550 @ 2.33GHz

- 2Gbyte RAM
- 160Gbyte disk
- MSI 8600GT PCI-EX 256MB DRR low profile

Installed software:

- Debian GNU/Linux 6.0, kernel 2.6.32-5-686
- Eclipse version 3.5.2 Java EE IDE for Web Developers
- JADE version 4.1
- JRE version 1.6.0.24 with just-in-time compiler enabled

No applications were running on the machines, other than the minimal operating system services and the agents used in the experiments. The experiments were conducted over a period of low machine usage by other users. As a result the observed performance measurements are more conservative.

6.3.2 The GUI

The GUI of the MAS is used to display information about the simulation that is running. The GUI is only rendered on one host machine. The Balancer agent creates the GUI and only sends important and relevant information to the GUI. This includes information on the weather and the contracts that have been established. The Balancer updates the GUI multiple times per minute. Figure 13 shows the entire GUI screen.

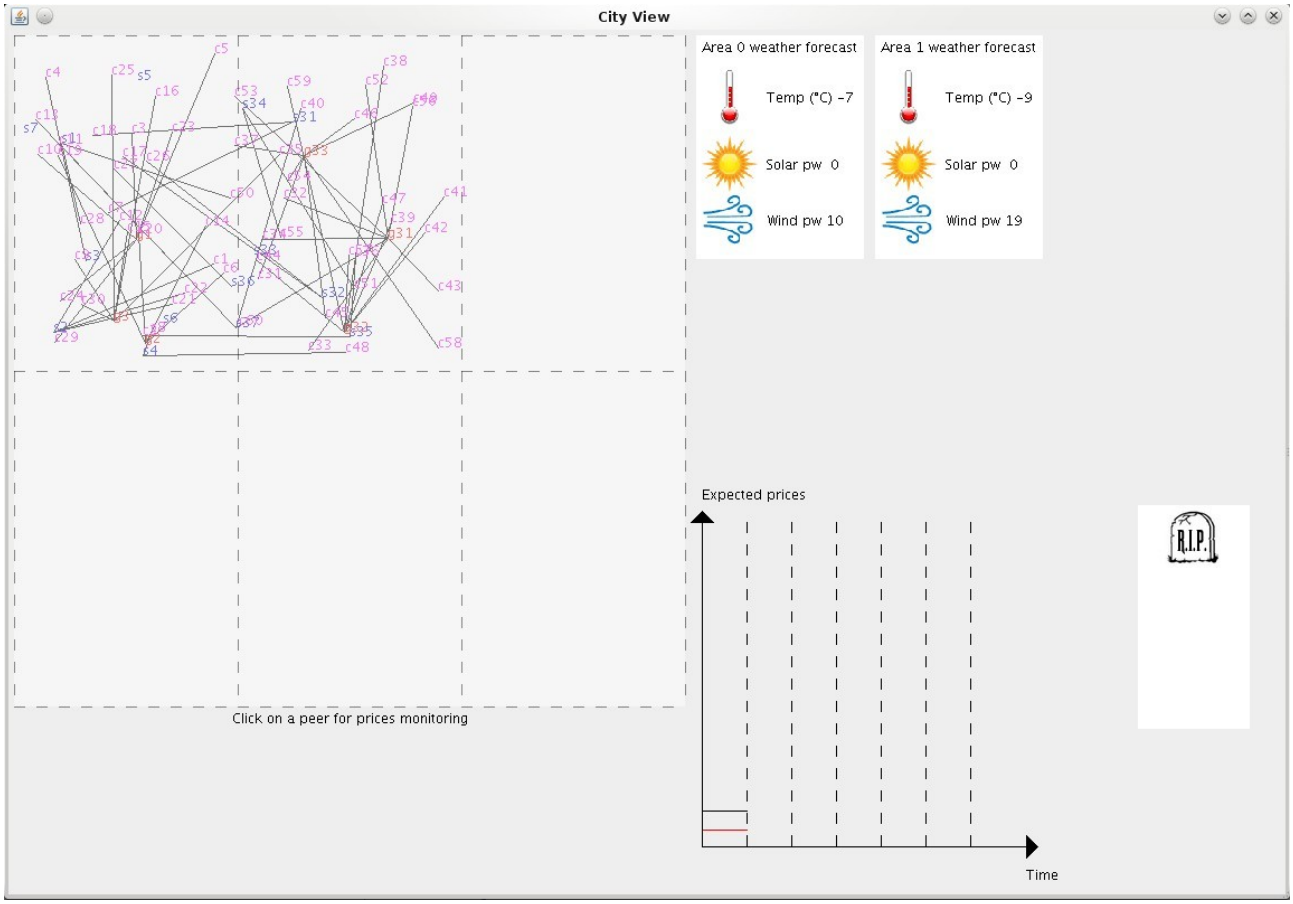

Figure 13: The GUI screen

In the top left corner of the screen, shown in Figure 13, is the area view. All the agents are displayed here within their respective areas. The area view is divided into six areas, but the view can be adjusted as more areas may be needed for future simulations. Within each area the different agent types are displayed. The Consumer agents are pink and start with a 'c', the Gencos are red and start with a 'g' and the Prosumers are blue and start with a 'p'. The lines between the different agents represent the contracts that have been established. These lines are added during the run of the simulations, which means that the contracts are shown as they are established.

After the simulation has finished, you can click on the Consumer agents, to view their price graph. This is shown to the bottom right of the area view. The expected price and the real price are shown compared to the time of day. The expected prices is shown in red and the actual price in black. The weather forecast of each area is shown in the top right of the screen. Each area has a forecast consisting of temperature, solar power and wind power. In the bottom right of the screen is the list of dead agents, the names of the agents that have failed and have been restarted are listed here.

6.3.3 Results of scalability tests

The results from each of the five scalability Test cases are listed in this section. Some interesting results have been obtained during the tests.

Test case 1

The first scalability test case focused on how many agents the MAS can handle before it becomes unstable. The MAS by N. Capodieci (old MAS) and the new MAS are compared in performance. The old MAS is run on one host machine and the new MAS on six hosts. Measurements have been performed on the number of messages transferred, the time to reach convergence and the average CPU and memory usage of the system and JADE. The results from Test case 1 are shown in the following table.

Test case 1: How many agents can the MAS handle before it becomes unstable?		
Metrics	**Test 1, new MAS**	**Test 1, old MAS**
Number of hosts	6	1
Number of agents	818(600c, 140p, 60g, 6b, 6t, 6w)	243(180c, 42p, 18g, 1b, 1t, 1w)
Average number of messages transferred between agents	2865639	654,39
Average time to reach convergence/Number of ticks that have passed	519800 ms	1879993 ms
Average CPU and memory usage	46%, 566MB	39%, 667MB
Average CPU and memory usage by JADE	13%, 230MB	40%, 100MB

In the Test case 1 table, the limit of the new MAS is around 818 agents, consisting of 600 Consumers, 140 Prosumers, 60 Gencos, 6 Balancer agents, 6 Weather agents and 6 Time agents. The simulation only takes about 8 minutes, but it creates a whopping 2865539 messages. This is simply caused by the amount of agents that participate in the system. Because of this amount of messages, the system sometimes loses an important message, which can cause the system to halt. This problem increases as more agent are added, but around 818 agents the system still mostly works correctly. The CPU usage of the MAS is not very high, because of the spread of the agents on the six hosts. The JADE CPU usage is remarkably low, which is probably caused by the fact that the

MAS has to do much more work than JADE. The memory usage of the MAS and JADE is pretty high, because of the amount of agents.

The limit of the old MAS is around 243 agents, consisting of 180 Consumers, 42 Prosumers, 18 Gencos, 1 Balancer, 1 Weather agent and 1 Time agent. With this amount of agents the system crashes after a while, due to the amount of RAM used, which is 667MB for the system alone. The amount of memory exceeds the Java heap space and the system crashes. Also the run-time is very long, around 31 minutes, which is much longer than the new MAS. The amount of messages is lower than for the new MAS, but this is caused by the much smaller amount of agents. The CPU usage of JADE is much higher than the new MAS, because the MAS is doing less work. The memory usage of JADE is much lower, which is also caused by the lower amount of agents.

It is clear that the new MAS is capable of handling about four times as many agents as the old MAS. In this case the amount of hosts was six, but in principle the system can use much more hosts and therefore be able to handle more agents. At the same time, the time to reach convergence is about four times lower on the new MAS than on the old MAS. The limit to the scalability of the system is therefore better on the new MAS than on the old MAS.

Test case 2

The second scalability test case focused on the scalability of the MAS, compared to the scalability of the old MAS. Both MAS are compared in performance. The old MAS is run on one host machine and the new MAS on six hosts. Three tests have been performed with different amounts of agents. The first test uses 30 Consumers, 7 Prosumers and 3 Gencos, the second test uses twice this amount and the third test uses four times this amount. Measurements have been performed on the number of messages transferred, the time to reach convergence and the average CPU and memory usage of the system and JADE. The results from Test case 2 are shown in the following table.

Test case 2: What is the scalability of the new MAS compared to the MAS by N. Capodieci, in terms of performance per load?						
Metrics	**Test 1, new MAS**	**Test 1, old MAS**	**Test 2, new MAS**	**Test 2, old MAS**	**Test 3, new MAS**	**Test 3, old MAS**
Number of hosts	6	1	6	1	6	1
Number of agents	58(30c, 7p, 3g, 6b, 6t, 6w)	43(30c, 7p, 3g, 1b, 1t, 1w)	98(60c, 14p, 6g, 6b, 6t, 6w)	83(60c, 14p, 6g, 1b, 1t, 1w)	178(120c, 28p, 12g, 6b, 6t, 6w)	163(120c, 28p, 12g, 1b, 1t, 1w)
Average number of messages transferred between agents	4578	7124	7408	54578	28189	531730
Average time to reach convergence/Number of ticks that have passed	10871 ms	10486 ms	12148 ms	38076 ms	22583 ms	696450 ms
Average CPU and memory usage	42%, 190MB	51%, 234MB	46%, 143MB	55%, 306MB	42%, 244MB	72%, 606MB
Average CPU and memory usage by JADE	36%, 100MB	25%, 71MB	32%, 113MB	27%, 77MB	25%, 132MB	18%, 95MB

From the Test case 2 table, some graphs can be made to clarify important features. The CPU usage of both MAS is largely constant and no trend is visible in the data, so no graph has been made of this. The CPU usage of the new MAS is consequently 9% lower than the old MAS, indicating that the scalability of the new MAS is considerably better on this point than the old MAS. Two graphs showing the run-time of the system and the amount of messages sent are shown below.

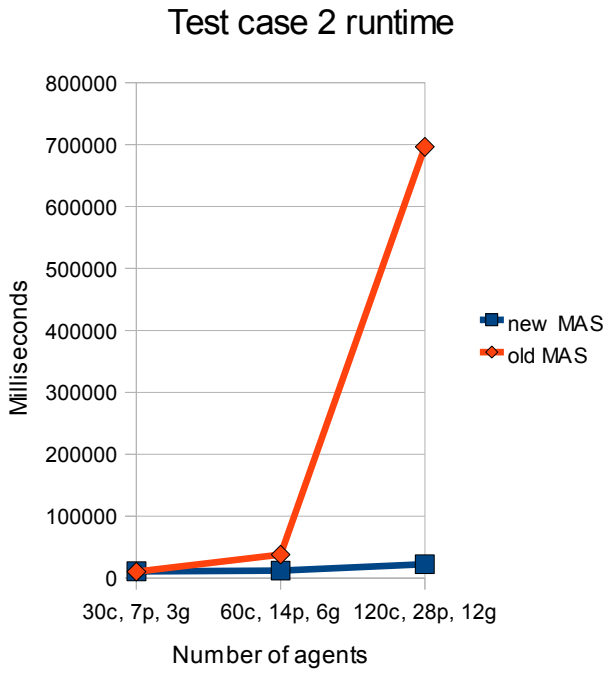

Figure 14: Runtime of the old and new MAS for different amounts of agents

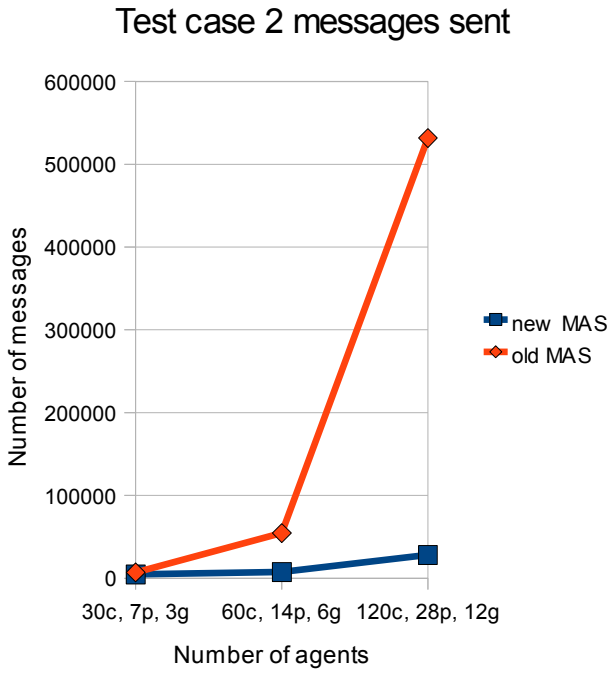

Figure 15: Messages sent by the old and new MAS for different amounts of agents

In Figure 14 the average time to reach convergence is shown in milliseconds, of both the new MAS and the old MAS. Three results for each MAS were obtained, for the different agent amounts. Both MAS performed similar in the first test, with the lowest amount of agents. But the second and especially the third test results are very far apart in run-time. The old MAS matches a Quadratic growth function: y = 62.8327 x^2 + -6850.18x + 183961. The new MAS also matches a Quadratic growth function: y = 0.820937 x^2 + -66.5875x + 12221, but a much lower one, as the coefficients are more than 50 times lower. The new MAS only requires 24 seconds to finish the simulation in the third test, the old MAS requires 11 minutes to do the same.

In Figure 15 the average number of messages being sent during a simulation is shown, of both the new MAS and the old MAS. Three results for each MAS were obtained, for the different agent amounts. Both MAS performed similar in the first test, with the lowest amount of agents. But the second and especially the third test results are very far apart. The old MAS matches a Quadratic growth function: y = 39.8171 x^2 + -3591.7x + 87084.7. The new MAS also matches a Quadratic growth function: y = 1.5751 x^2 + -118.263x + 6788.33, but a much lower one, as the coefficients are more than 30 times lower.

From these results, it can be concluded that the scalability of the new MAS is significantly better than the old MAS. The run-time of the old MAS is much longer than the new MAS, when the amount of agents starts to increase. Also, the amount of messages being sent in the old MAS rises much quicker than the new MAS.

Two more graphs, showing the memory usage of both MAS, are shown below.

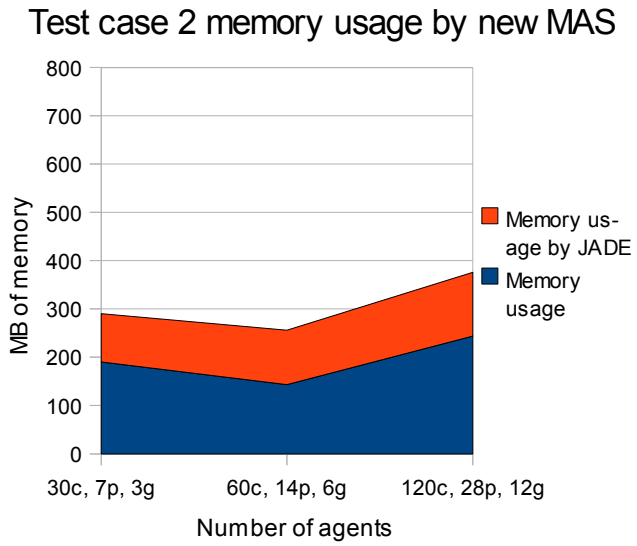

Figure 16: Memory usage of the new MAS and JADE for different amounts of agents

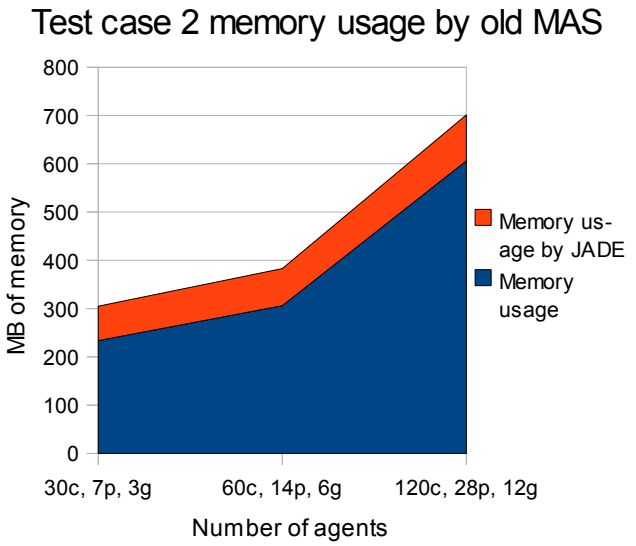

Figure 17: Memory usage of the old MAS and JADE for different amounts of agents

In Figure 16 the average memory usage in MB of the new MAS and JADE are shown in a stacked graph. Three results were obtained, for the different agent amounts. The memory usage of the MAS for the first test is 190 MB and for JADE 100 MB. The second test is lower than the first, which might be caused by an error in the simulation, using only 143 MB for the MAS and 113 MB for JADE. The third test is a bit higher, but still quite low at 244 MB for the MAS and 132 MB for JADE. The new MAS matches a Linear growth function: $y = 0.828571 x + 230$.

In Figure 17 the average memory usage in MB of the Old MAS and JADE are shown in a stacked graph. Three results were obtained, for the different agent amounts. The memory usage of the old MAS for the first test is very similar to the new MAS and is 234 MB and for JADE 71 MB. The second test is higher than the first, using 306 MB for the MAS and 77 MB for JADE. The third test is much higher than the second, at 606 MB for the MAS and 96 MB for JADE. The old MAS matches a Linear growth function: $y = 3.40536 x + 145.5$.

The new MAS uses less memory than the old MAS, the difference is not very large in the first test, but increases quickly as the amount of agents increases. Another important factor is that JADE is only running on one host machine and not on all six, meaning that the memory usage of the other hosts is defined by the blue area shown in Figure 16. The new MAS therefore achieves a much better scalability than the old MAS, as the system can support around 1100 agents according to the growth function. The old MAS can only support around 200 agents according to the growth function.

The next graph displays the CPU usage of JADE of both MAS.

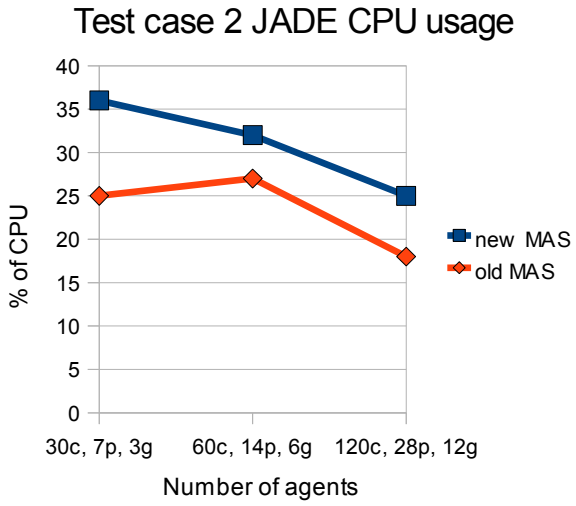

Figure 18: JADE CPU usage for the old and new MAS, for different amounts of agents

In Figure 18 the CPU usage of JADE is shown in percent for both the new MAS and the old MAS. Three results were obtained for each MAS. For the runs of the old MAS, JADE used 25%, 27% and 18% of the CPU for each of the three tests, indicated by the red line. For the runs of the new MAS, JADE used 36%, 32% and 25% of the CPU for each of the three tests, indicated by the blue line. The CPU usage of JADE seems to decrease for both MAS as the amount of agents increases. JADE uses less CPU for the old MAS than the new MAS, which can be explained by the fact that the new MAS works on six hosts and the old MAS on only one host. On a MAS with one host, JADE doesn't need to handle communication between different hosts and thus it uses less CPU. On six hosts JADE has to handle all the communication between these hosts and therefore requires more CPU resources.

This decrease of CPU usage is good for the scalability of the system, because at this rate the CPU usage will not be a bottleneck on the system. Also only one host in the new MAS runs JADE, which means that the other hosts do not suffer from the CPU usage of JADE.

Test case 3

The third scalability test case focused on the scalability of the new MAS, with respect to the performance increase per resource. Three tests have been performed with different amounts of hosts, while keeping the amount of agents the same. The amount of hosts ranges from one to three and to six. The amount of agents is 99, 60 Consumers, 14 Prosumers, 6 Gencos, 6 Balancers, 6 Time agents, 6 Weather agents and 1 EH agent. Measurements have been performed on the number of messages transferred, the time to reach convergence and the average CPU and memory usage of the system and JADE. The results from Test case 3 are shown in the following table.

Test case 3: What is the scalability of the new MAS with respect to performance increase per resource?			
Metrics	**Test 1**	**Test 2**	**Test 3**
Number of hosts	1	3	6
Number of agents	99(60c, 14p, 6g, 6b, 6t, 6w, 1e)		
Average number of messages transferred between agents	7851	5058	5470

Average time to reach convergence/Number of ticks that have passed	17562 ms	10569 ms	17113 ms
Average CPU and memory usage	55%, 323MB	50%, 263MB	40%, 148MB
Average CPU and memory usage by JADE	22%, 111MB	32%, 118MB	39%, 115MB

From the Test case 3 table, some graphs can be made to clarify important features. There is no clear trend in the data on the amount of messages transferred or the time to reach convergence, so no graphs have been made of this data. A reason for this could be that the agents themselves do not change and thus do not require more communication or more time to reach convergence. Two graphs have been made that show the CPU and memory usage of the system and JADE.

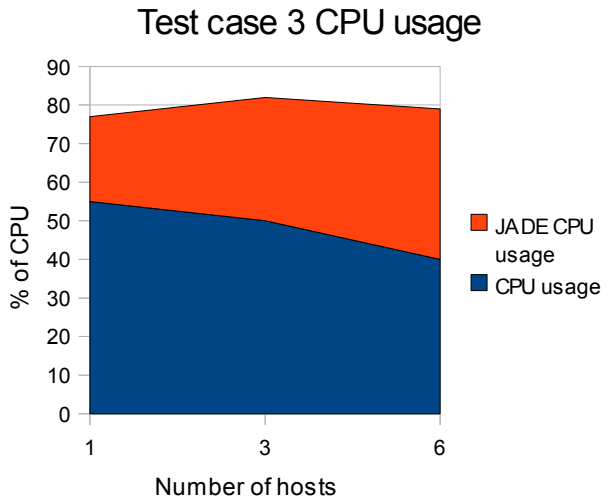

Figure 19: CPU usage of the new MAS and JADE for different amounts of hosts

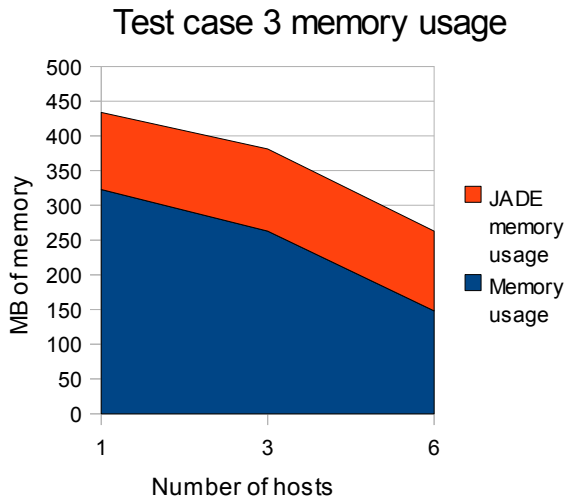

Figure 20: Memory usage of the new MAS and JADE for different amounts of hosts

In Figure 19 the CPU usage of JADE and the new MAS are shown in percentage and are stacked. Three tests were performed, using one, three and six hosts. The CPU usage of the new MAS decreases as the amount of hosts increases, which is shown by the blue area in the graph. The percentage went from 55% to 50% and then to 40% and matches a Linear function: $y = -3.02632 x + 58.4211$. This decrease can be explained by the distribution of the agents over the different hosts. The CPU usage of JADE increases slightly as the amount of hosts increases, visualized by the red area. This percentage went from 22% to 32% and to 39% and matches a Linear function: $y = 3.31579 x + 19.9474$. This increase is caused by the additional hosts in the system, as JADE handles more communication between the hosts. JADE only runs on one host machine and therefore the average CPU usage on the other hosts is defined by the blue area and the corresponding Linear function. The CPU usage on the JADE host will still only slightly rise until about 400 hosts are present, as defined by the two Linear functions.

In Figure 20 the memory usage of JADE and the new MAS are shown in MB of RAM and are stacked in the graph. Three tests can be seen here as well, using one, three and six hosts. The memory usage of the new MAS decreases as the amount of hosts increases, shown by the blue area in the graph. The usage went from 323 MB to 263 MB and then to 148 MB and matches a Linear function: $y = -35.2632 x + 362.211$. This decrease can be explained by the distribution of the agents over the different hosts. The memory usage of JADE does not change much as the amount of hosts increases, visualized by the red area. This usage went from 111 MB to 118 MB and to 115 MB, which makes clear that there is no clear increase or decrease in memory usage.

These results indicate that for most of the hosts the resource usage will decrease significantly as more hosts are added, except for the host that is running JADE. The CPU usage on this host will increase slightly, but as the maximum amount of hosts is around 400, this does not pose a great threat on scalability. This means that the scalability with respect to performance increase per resource is significant enough to make adding hosts useful.

Test case 4

The fourth scalability test case focused on the performance overhead of restarting agents through the EH agent. In total seven tests have been performed, each on one host and using 84 agents, 60 Consumers, 14 Prosumers, 6 Gencos, 1 Balancer, 1 Time agent, 1 Weather agent and 1 EH agent. Measurements have been performed on the number of messages transferred, the time to reach convergence and the average CPU and memory usage of the system and JADE. The results from Test case 4 are shown in the following table.

Test case 4: What is the performance overhead of restarting an agent in the system through the EH agent?							
Metrics	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test 6**	**Test 7**
Number of hosts	1						
Number of agents	84(60c, 14p, 6g, 1b, 1t, 1w, 1e)						
Average number of messages transferred between agents	21047	12432	17730	17650	11189	5837	6606
Average time to reach convergence/Number of ticks that have passed	9756 ms	10923 ms	9989 ms	10269 ms	15157 ms	9005 ms	10129 ms
Average CPU and memory usage	58%, 275MB	54%, 230MB	55%, 226MB	55%, 246MB	53%, 197MB	54%, 202MB	52%, 178MB
Average CPU and memory usage by JADE	21%, 90MB	24%, 93MB	23%, 92MB	21%, 86MB	23%, 96MB	20%, 93MB	23%, 88MB

From the Test case 4 table, a graph is made to clarify the important features. The data on the average CPU and memory usage of the new MAS and JADE is constant, so no graphs have been made of this data. A reason for this could be that the restart of an agent is not big enough to influence the system on the hardware level and only on the software level. One graph has been made to show the number of messages transferred and the run-time of the new MAS.

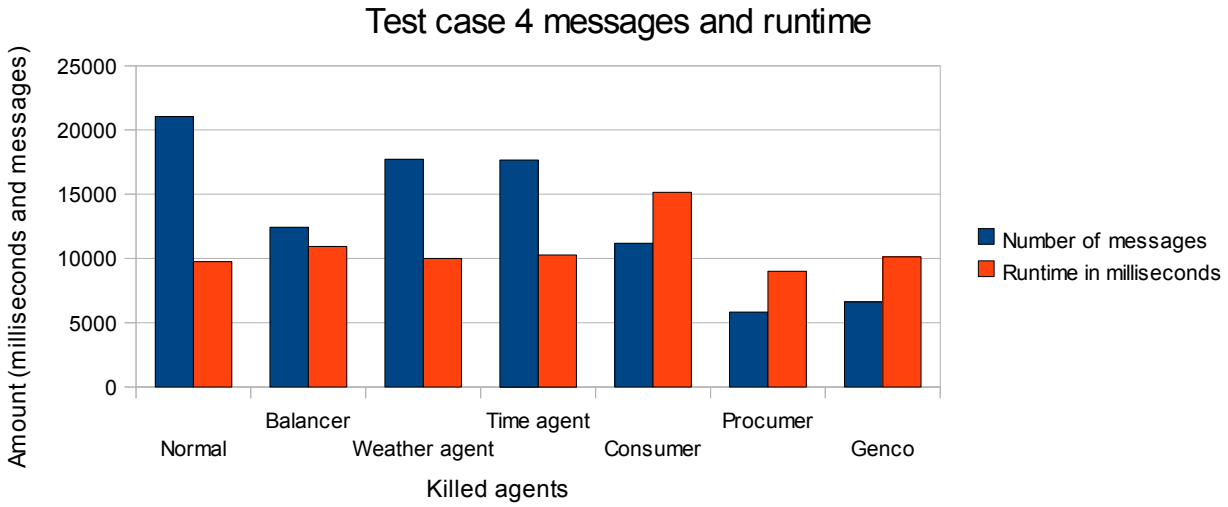

Figure 21: Number of messages sent and Runtime of the new MAS for different killed agents

In Figure 21 the amount of messages being sent in the system, shown in blue, and the run-time in milliseconds, shown in red, is shown for each different test. The first test is a normal run of the system without an agent being killed. The number of messages being sent in this test is higher than in all the other tests. This is caused by the fact that a killed agent cannot receive messages and it takes some time for the agent to be restarted. When a Weather or Time agent is killed, the number of messages is a little lower than in the normal case, as these agents are not contacted very often. A killed Prosumer or Genco results in a very large decrease of messages being sent, as these agents are contacted a lot during the simulation and also send a lot of messages. The run-time of the new MAS in milliseconds is largely the same for most tests, but is more than half as long in the case of a killed Consumer. This is caused by the fact that the simulation is only finished if every Consumer has a contract. As the killed Consumer has to be restarted and restarts its negotiations from the beginning, this takes some time and causes this delay.

The performance overhead of using the EH agent to restart the system is not significant, in many cases the restarting even caused a performance increase in run-time and messages sent. The CPU and memory usage does not differ from the normal case, causing no performance problems. The only case that can result in a longer run-time of the system is killing a Consumer agent. The results on the reliability increase of using the EH agent are shown in the next section.

Test case 5

The fifth scalability test case focused on the performance overhead of critical agent replication. In total three tests have been performed, each on six hosts and each with 60 Consumers, 14 Prosumers, 6 Gencos and 1 EH agent. The first test uses one of each critical agent, the second test uses three and the third test uses six. Measurements have been performed on the number of messages transferred, the time to reach convergence and the average CPU and memory usage of the system and JADE. The results from Test case 5 are shown in the following table.

Test case 5: What is the performance overhead of critical agent replication?			
Metrics	**Test 1**	**Test 2**	**Test 3**
Number of hosts	6		
Number of agents	84(60c, 14p, 6g, 1b, 1t, 1w, 1e)	90(60c, 14p, 6g, 3b, 3t, 3w, 1e)	99(60c, 14p, 6g, 6b, 6t, 6w, 1e)

Average number of messages transferred between agents	3028	3581	5470
Average time to reach convergence/Number of ticks that have passed	6720 ms	7971 ms	17113 ms
Average CPU and memory usage	43%, 171MB	43%, 169MB	40%, 148MB
Average CPU and memory usage by JADE	36%, 112MB	31%, 125MB	39%, 115MB

From the Test case 5 table, a graph is made to clarify the important features. The data on the average CPU and memory usage of the new MAS and JADE has no clear features and is largely similar, so no graphs have been made of this data. A reason for this could be that the extra agents that are running do not have a big enough impact on the hardware level and only on the software level. One graph has been made to show the number of messages transferred and the run-time of the new MAS, according to the number of replicas.

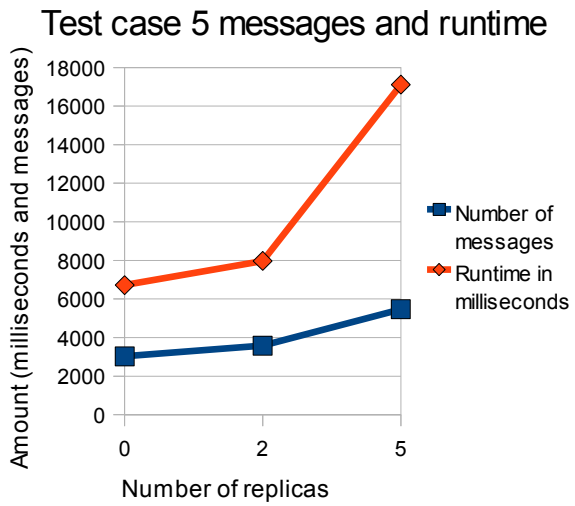

Figure 22: Number of messages sent and Runtime of the new MAS for different amounts of replicas

In Figure 22 the amount of messages being sent, shown in blue, and the total run-time in milliseconds, shown in red, is displayed. Three test results were obtained, for zero replicas, two replicas and five replicas. The number of messages being sent in the new MAS is clearly increasing for each added replica. The data states an increase from 3028 to 3581 till 5470, which corresponds to a Quadratic growth function: $y = 70.6333 x^2 + 135.233x + 3028$. The major cause of this increase is the Balancer agent, because every Consumer sends messages to each Balancer agent and every Balancer agent sends messages to all the other agents in the system. The Time and Weather agent do not send extra messages as they wait for incoming messages and are not contacted by other agents very often. The run-time of the new MAS in milliseconds is increasing as well for each added replica, but even faster than the amount of messages. The data states an increase from 6720 to 7971 till 17113, which corresponds to a Quadratic growth function: $y = 484.367 x^2 + -343.233x + 6720$. This is also largely caused by the Balancer agent, because more message are sent and therefore the run-time of the system increases.

There is a clear performance overhead in using critical agent replication, mainly in the run-time of the system, but this overhead can be contained by limiting the amount of extra replicas. The results from the reliability tests can indicate how many extra replicas are needed for an increase in

reliability.

6.3.4 Results of reliability tests

The results from each of the three reliability Test cases are listed in this section.

Test case 1

The first reliability test case focuses on the effectiveness of the EH agent to prevent complete system failure if a certain agent fails. To test this, the new MAS is run with the EH agent and without the EH agent for each test and the results are compared. Two tests have been performed, one on only one host machine, with 83 agents, 60 Consumers, 14 Prosumers, 6 Gencos, 1 Balancer, 1 Time agent and 1 Weather agent. And a test on six hosts, with 98 agents, 60 Consumers, 14 Prosumers, 6 Gencos, 6 Balancer agents, 6 Time agents and 6 Weather agents. In each test, every agent type was killed twice, resulting in twelve simulations per test. Measurements have been performed on the number of successful simulations, which is combined with the number of simulations in order to calculate a Success rate. The results from Test case 1 are shown in the following table.

Test case 1: How effective is the EH agent in preventing complete system failure, when a certain agent fails?				
Metrics	**Test 1, Without EH agent**	**Test 1, With EH agent**	**Test 2, Without EH agent**	**Test 2, With EH agent**
Number of hosts	1		6	
Number of agents	83(60c, 14p, 6g, 1b, 1t, 1w)	84(60c, 14p, 6g, 1b, 1t, 1w, 1e)	98(60c, 14p, 6g, 6b, 6t, 6w)	99(60c, 14p, 6g, 6b, 6t, 6w, 1e)
Success Rate = Number Of Successful Simulations/ Number Of Simulations	7 / 12 = 0.583	9 / 12 = 0.75	10 / 12 = 0.833	11 / 12 = 0.917

From the Test case 1 table, a graph is made of the different tests and their Success rates, in order to clarify the important features.

Figure 23: Success rates of simulations on the new MAS for different amounts of hosts

In Figure 23 the Success rates of the new MAS on one and six hosts are displayed in a graph. The Success rate of the new MAS with the EH agent is shown in red and without the EH agent is shown in blue. From this graph it is clearly visible that the EH agent results in higher Success rates on both tests. This means that, if an agent fails, the system continues to function more often with the EH agent, than without the EH agent. This indicates that an Exception Handling agent is effective in increasing the reliability of the system. Together with the scalability test results, this makes a very useful and effective method. The Success rate is higher in the test with six hosts, with and without the EH agent. This is probably caused by the replication of the critical agents on six hosts and should be visible in the next Test case 2.

Test case 2

The second reliability test case focuses on the effectiveness of critical agent replication to prevent complete system failure if a top-level intermediary fails. To test this, the new MAS is run without the EH agent for each test, to filter out the effect of the EH agent on the Success rate and the results are compared. Three tests have been performed, with 60 Consumers, 14 Prosumers and 6 Gencos. The first test uses one of each critical agent, the second test uses three and the third test uses six. In each test, every critical agent type was killed twice, resulting in six simulations per test. Measurements have been performed on the number of successful simulations, which is combined with the number of simulations in order to calculate a Success rate. The Replication rate is also calculated according to the number of extra replicas and the total number of agents. The results from Test case 2 are shown in the following table.

Test case 2: How effective is critical agent replication in preventing complete system failure, when a top-level intermediary fails?			
Metrics	**Test 1, Without EH agent**	**Test 2, Without EH agent**	**Test 3, Without EH agent**
Number of hosts	1	3	6
Number of agents	83(60c, 14p, 6g, 1b, 1t, 1w)	89(60c, 14p, 6g, 3b, 3t, 3w)	98(60c, 14p, 6g, 6b, 6t, 6w)
Replication Rate = Number Of Extra Replicas/Number Of Agents	0 / 83 = 0	6 / 83 = 0.07	15 / 83 = 0.18

Success Rate = Number Of Successful Simulations/Number Of Simulations	3 / 6 = 0.5	5 / 6 = 0.833	6 / 6 = 1

From the Test case 2 table, a graph is made of the Replication rate vs the Success rate, in order to clarify the important features.

Figure 24: Replication rate vs Success rate of simulations on the new MAS

In Figure 24 the Success rate of the new MAS is set out against the Replication rate. An increasing line is visible and indicates that as the replication rate increases, the success rate increases as well. With a replication rate of 0.18, the success rate of the system is already at the maximum of 1. This means that whenever a critical agent/ top-level intermediary fails, the system continues to function and will complete the simulation. Without replication, the system only completes the simulation half of the time. This clearly indicates that critical agent replication is very effective in increasing the reliability of the system. Together with the results from the scalability test on critical agent replication, this results in a trade-off between the reliability and the scalability of the new MAS.

Test case 3

The third reliability test case focuses on the impact of the queue limit and garbage collector on agent thrashing. To test this, both MAS are run normally once and once with an added Spam agent, which sends messages to every agent in the system, and the results are compared. Two tests have been performed, with 60 Consumers, 14 Prosumers and 6 Gencos. Measurements have been performed on the time to reach convergence and the average CPU and memory usage of the MAS and JADE. The results from Test case 3 are shown in the following table.

Test case 3: What is the impact of a queue limit and garbage collector on agent thrashing?				
Metrics	**Test 1, new MAS**	**Test 1, old MAS**	**Test 2, new MAS**	**Test 2, old MAS**
Number of hosts	1			
Number of agents	83(60c, 14p, 6g, 1b, 1t, 1w)			
Average time to reach convergence/Number of ticks that have passed	9756 ms	38076 ms	11588 ms	78597 ms
Average CPU and memory usage	58%, 275MB	55%, 306MB	51%, 377MB	57%, 402MB
Average CPU and memory usage by JADE	21%, 90MB	27%, 77MB	17%, 76MB	25%, 84MB

From the results of the Test case 3 table, there is one significant difference noticeable in the second test between both MAS. This is the average time to reach convergence. The new MAS finishes in 12 seconds, while the old MAS takes 79 seconds. Even more interesting was the fact that the new MAS had a Spam agent that sent a message to each agent in the system every 50 milliseconds, where the old MAS could not finish the simulation in the same case and could only work if messages were sent every 500 milliseconds.

Compared to the first test, the new MAS performs slightly less with the Spam agent active in the second test, which was to be expected. The time to reach convergence is about 19% higher. Also the memory usage is about 100 MB higher, but this is caused by the amount of added messages to the system by the Spam agent. The old MAS however, performs much worse in the second test than in the first test. The time to reach convergence is increased by 106%. Here the memory usage is also about 100 MB higher.

As seen from these results, the impact of a queue limit and garbage collector is huge. It can prevent agent thrashing due to a faulty agent which is spamming messages. This is clearly what happened to the old MAS in this test and what was prevented in the new MAS. The reliability of the new system is therefore much better, as the system can cope with faulty agent.

7 Discussion

In this chapter there is a summary of the previous chapters in this thesis in the first section. The research questions that have been formed and discussed in the introduction are answered in the next section according to the research explained in this thesis. The next section explains the answers to the evaluation questions that have been formed in the evaluation. Finally there are some concluding remarks and future work.

7.1 Summary

In this thesis, the main focus was to create a reliable Multi-agent system for a large scale distributed energy trading network. The need for such a system was described in the introduction and is linked to the need for changes in the current energy market. The current model is too monopolistic and favors the big energy production companies, or Gencos. The consumers are therefore bound to high prices and limited innovation, because of low market competition. This model is now slowly undergoing some changes, such as new Gencos being added, which tend to make it a more open market where there is more focus on innovation as well as environmental issues. However, there is

still a lot that has to be improved, as today the so called Prosumers can only sell their energy back to their Genco and not other consumers. The ideal future vision is that of an open energy market where Prosumers and Gencos compete for the consumers. The necessary changes to the current market and energy grid therefore have to be identified and a energy market simulation can provide a platform where these changes can be tested before they are actually used. It can also be used after the actual changes have been made, as a testing and monitoring environment. It has been shown that the ideal candidate for such a simulation is a Multi-agent system, because of the autonomy and ability to model behavior.

In the state of the art different related projects have been discussed as well as Multi-agent systems in general. A lot of different platforms and agent languages exist that can be used to implement a Multi-agent system. Also a lot of different systems have already been implemented in the field of energy market simulations. One of these systems is the MAS by N. Capodieci, which was used a basis for this thesis. This is a basic MAS that simulates the energy market by using Consumer, Prosumer and Genco agents that buy and sell energy in a contracting auction. It also has a time and weather simulation and a GUI.

The scope of this research was to use the existing MAS by N. Capodieci that supports this simulation and to expand it by adding scalability and reliability improvements to make the system more usable. This was needed because these features have not been taken into account when that MAS was created. The system was limited to one host only and has no specific measures to prevent failures. This limited the usage of the system, as only a limited amount of agents could be run and the system could crash on a fault.

JADE was used as the platform for the Multi-agent system implementation, as explained in the background information. The JADE platform provided the best advantages and features for the system that needed to be created. Also, the MAS by N. Capodieci uses JADE, which was another major advantage. A lot of different methods of improving scalability and reliability for Multi-agent systems exist, but not all of these were suitable in this case. The advantages and disadvantages of each method have been compared and a selection has been made on this basis. Some of the more prominent methods were: the use of distribution and replication, which proved invaluable for the scalability and reliability of the system.

The architecture of JADE and the MAS has been clearly explained. JADE uses a container that wraps the agents. This allows the JADE system to distribute on several hosts, each running one or more containers, which results in a system that supports transparent access. JADE also implements an agent behavior scheduler and uses agent address caching, which are in total three of the methods chosen that can increase the scalability of a MAS. The architecture of the MAS itself was focused on the structure of the agent communication and interaction and the structure of the system, which corresponded largely to the JADE structure. Different methods have been realized and are visible in the architecture, such as distribution, replication and the EH agent.

The implementation was focused on the class diagram and class implementation of the agents. Three features that improve the reliability of the MAS have been implemented on this level, the queue limit, the garbage collector and the handling of messages from the EH agent.

A wide range of test cases have been created that focused on testing the selected methods in the field of scalability and reliability and are based on the research questions stated in the introduction. In total eight test cases have been created, five for scalability tests and three for reliability tests. The tests cases focused on different aspects of the scalability and reliability of the system, such as the performance increase per added resource and the agent limit, as well as the impact of the EH agent and replication on the success rates.

The tests have been performed on a reasonably fast system and measures have been taken to ensure

realistic and correct test results, such as running no other programs on the system and using similar systems for testing with multiple hosts. Also each of the tests within each test case has been performed five times, in order to average the values and remove spikes in the data.

7.2 Research questions and answers

The research questions defined in the introduction have been used as a guideline throughout this thesis. They have been used to create test cases that could answer the research questions. The answers to these questions are provided below.

How to support the large scale of this system?:

- Which techniques have already been applied to other large scale MAS, to increase scalability?

  This question is answered in section 3.2. There is a wide range of methods and techniques that can be used to increase scalability. In this thesis only the more widely used and general methods to achieve this, have been taken into account. Some of these methods are not bound to the field of Multi-agent systems alone and are used in other fields of software engineering as well. Some of the techniques that are described in section 3.2 are: distribution, replication, locating agents based on caching lists, changing the agent organizational form, hiding communication latencies, etc.

- What are the advantages and disadvantages of these techniques and are they suitable for our MAS?

  This question is also answered in section 3.2 and partially in section 3.4.1. Of each of the methods described in these sections, the advantages and disadvantages are mentioned. These are then compared and their suitability for our MAS is considered. Some of the methods did not fit our MAS and some had too many disadvantages too be usable. The methods that have failed to suit this MAS are: changing the agent organizational form and hiding communication latencies, as explained in section 3.4.1.

How to make sure that the system is reliable?

- Which techniques have already been applied to other large scale MAS, to increase reliability and fault-tolerance?

  This question is answered in section 3.3. The range of methods and techniques that were used for this is similar to the range of the scalability methods. A lot of the methods that are devised for reliability increase are not used very often and there are less of these methods available. This is caused by the fact that reliability in Multi-agent systems is not a major field of study. Only replication is a well known widely used technique for improving reliability. Some of the methods that are described in section 3.3 are: domain-independent exception handling service, replication, using sentinels to check the system, refuse requests ability, etc.

- What are the advantages and disadvantages of these techniques and are they suitable for our MAS?

  This question is also answered in section 3.3 and partially in section 3.4.2. As for the scalability methods, the reliability methods have also been examined on advantages and disadvantages. These have been compared and the suitability of the methods for our MAS was considered. There were more methods that have failed to suit this MAS than for the scalability methods. This is probably caused by the fact that there a less reliability

improvement methods and that these methods are not used very often. The methods that failed to suit this MAS are: using sentinels to check the system, increasing agent mobility and passive and active replication.

How to implement this system?

- How have other MAS been implemented?

  This question is answered in section 2 and partially in section 3.1. In the first section the characteristics of other MAS have been considered as well as their implementation. For instance the agent languages that are used today in Multi-agent systems are considered, as well as the agent platform that are used. One of the conclusions in section 3.1 is that JADE is one of the most widely used agent platforms for implementation. JADE also has a long list of features and advantages that fulfill the implementation needs for this MAS.

- How to implement the techniques which are suitable for our MAS?

  This question is answered in sections 4 and 5. Because most of the techniques that have been selected as improvement methods for scalability and reliability are research subjects, there is not a lot of code mentioned in the papers. In fact almost no papers use code blocks. Also, as some of the methods are not widely used, there is no implementation to be found on the Internet. Therefore the implementation of each method is based on the theory discussed in the paper where the method is described. The details on how to implement each of the techniques is explained in sections 4 and 5, as these deal with the architecture and implementation of the system.

How to evaluate such a system?

- How have other MAS been evaluated?

  This question is answered in section 6. Other MAS have been evaluated according to the criteria mention in this section. A few different papers focus entirely on evaluating reliability or scalability in Multi-agent systems. Some metrics and test cases have been proposed and used in these papers. These are used as a basis for the evaluations done in this thesis.

- How to evaluate the techniques and ensure correct functionality of the system?

  This question is also answered in section 6. Some of the papers that proposed the methods for improving scalability and reliability have also tested these methods, using some metrics and test cases. These metrics are also used in this thesis.

7.3 Evaluation questions and answers

The evaluation questions that have been formed in section 6 are answered here in short.

Scalability evaluation questions:

- How many agents can the MAS handle before it becomes unstable?

  The MAS can handle 818 agents on a 6 host system, which is four times as many as the old MAS. The MAS requires only 8 minutes to finish this simulation, where the old MAS takes four times as long.

- What is the scalability of the new MAS compared to the old MAS, in terms of performance per load?

  The new MAS was tested on 6 hosts, the old MAS on 1. The CPU usage of the new MAS did not increase as the amount of agents increased, this did happen on the old MAS. The

memory usage of the new MAS is significantly lower than the old MAS and increases slower, but both match a Linear function. The time to reach convergence and the amount of messages is limited in the new MAS and also rises much slower than on the old MAS, but both match a Quadratic function.

- What is the scalability of the new MAS with respect to performance increase per resource?

  As new hosts are added to the MAS, the time to reach convergence and the amount of messages transferred does not change noticeably. The memory and CPU usage decreases significantly for each added host by a Linear function, except for the host that is running JADE, which has a slight increase in CPU usage.

- What is the performance overhead of critical agent replication?

  The memory and CPU usage of the MAS shows no specific changes if more replicas are added. The time to reach convergence and the amount of messages does increase significantly by a Quadratic function.

- What is the performance overhead of restarting an agent in the system through the EH agent?

  The CPU and memory usage for restarting an agent does not differ from the normal case. The time to reach convergence and the amount of messages sent decreases in most of the cases, compared to the normal case. Only the restart of a Consumer has a negative effect on the time to reach convergence, which increases by 50%.

Reliability evaluation questions:

- How effective is the EH agent in preventing complete system failure, when a certain agent fails?

  The MAS with the EH agent results in higher success rates than the MAS without the EH agent. The success rates increase by 10-28%, depending on the amount of hosts.

- How effective is critical agent replication in preventing complete system failure, when a top-level intermediary fails?

  The MAS without critical agent replication has a success rate of 50%, where the MAS with a replication rate of 0.18 has a success rate of 100%.

- What is the impact of a queue limit and garbage collector on agent thrashing?

  The new MAS was tested with a Spam agent sending a message every 50 milliseconds, the old MAS was tested with a Spam agent sending a message every 500 milliseconds. The new MAS finishes in 12 seconds, as the old MAS without queue limit and garbage collection takes 79 seconds. There is no clear distinction in memory usage or CPU usage of both MAS.

7.4 Conclusion and future work

From the answers given to the research questions and evaluation questions, some conclusions can be drawn. This research has proven very useful for the Multi-agent system that was used. The old MAS created by N. Capodieci was a good functioning basis to expand on. The flaws that had to be addressed in that system have been fixed in the new MAS. The different methods that have been selected for improving scalability and reliability have mostly proven to be working as planned. Distribution of the system is one of the main performance increasing methods for the scalability. Also the methods implemented by JADE have proven their usefulness in the new system, as transparent access is required for a good functioning distributed system. The reliability has been

increased significantly by using critical agent replication and by using an exception handling service. From the test results it can also be concluded that the new MAS outperforms the old MAS on a lot of points. The new MAS supports much more agents and also runs faster and uses less messages to do the same. The memory and CPU usage is also lower in general and the number of successful simulations are much higher. This results in a system that is reliable and scalable and which can be used as a solid basis for expanded functionality. The possibilities for this expanded functionality are endless, as the system can be expanded by all kind of agent types, learning strategies, more hosts, improved GUI, realistic weather, real world topology, etc. Some possible expansions that I would personally find useful or would have done if I had more time:

1. Extended bug fixing and performance increases, by removing unnecessary messages and code.
2. Adding realistic weather.
3. Adding some small real world topology.

8 References

[1] Ralph Deters. Scalability & Multi-Agent Systems. 2001.

[2] N.J.E. Wijngaards, B.J. Overeinder, M. van Steen, and F.M.T. Brazier. Supporting Internet-Scale Multi-Agent Systems. In *Data & Knowledge Engineering,* Vol. 41, Issues 2-3, Pages 229-245, June 2002.

[3] L.C. Lee, H.S. Nwana, D.T. Ndumu and P. De Wilde. The stability, scalability and performance of multi-agent systems. In *BT Technol J,* Vol. 16, No. 3, July 1998.

[4] Phillip J. Turner and Nicholas R. Jennings. Improving the scalability of Multi-Agent Systems. In *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems.* Lecture Notes in Computer Science, Vol. 1887/2001, Pages 246-262, 2001.

[5] Omer F. Rana and Kate Stout. What is Scalability in Multi-Agent Systems? In *Proceedings of the fourth international conference on Autonomous agents (AGENTS '00)*, Pages 56-63, 2000.

[6] Onn Shehory. A Scalable Agent Location Mechanism. In *Intelligent Agents VI. Agent Theories Architectures, and Languages*. Lecture Notes in Computer Science, Vol. 1757/2000, Pages 162-172. 2000.

[7] Olivier Marin, Pierre Sens, Jean-Pierre Briot, Zahia Guessoum. Towards Adaptive Fault Tolerance For Distributed Multi-Agent Systems. In *Proceedings of ERSADS'2001*, May 2001.

[8] Staffan Hägg. A Sentinel Approach to Fault Handling in Multi-Agent Systems. In *Proceedings of the Second Australian Workshop on Distributed AI, in conjunction with Fourth Pacific Rim International Conference on Artificial Intelligence (PRICAI'96)*, 1996.

[9] Sanjeev Kumar and Philip R. Cohen. Towards a Fault-Tolerant Multi-Agent System Architecture. In *Proceedings of the fourth international conference on Autonomous agents*, Pages 459-466, 2000.

[10] Mark Klein, Juan-Antonio Rodriguez-Aguilar, Chrysanthos Dellarocas. Using Domain-Independent Exception Handling Services to Enable Robust Open Multi-Agent Systems: The Case of Agent Death. *Autonomous Agents and Multi-Agent Systems*, Volume 7, Pages 179-189, 2003.

[11] Zahia Guessoum, Jean-Pierre Briot, Olivier Marin, Athmane Hamel, and Pierre Sens. Dynamic and Adaptive Replication for Large-Scale Reliable Multi-Agent Systems. In *Software Engineering for Large-Scale Multi-Agent Systems*. Lecture Notes in Computer Science, Vol. 2603/2003, Pages

211-235, 2003.

[12] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing Multi-agent Systems with JADE. In *Intelligent Agents VII*, LNAI 1986, Pages 89-103, 2001.

[13] N. P. Yu. Modeling of Suppliers' Learning Behaviors in an Electricity Market Environment. *M.S. thesis, Dept. EE. Eng, Iowa. State University, Ames*. 2007.

[14] Eric Platon, Shinichi Honiden and Nicolas Sabouret. Challenges in Exception Handling in Multi-Agent Systems. In *Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems (SELMAS '06)*, May 22-23, 2006.

[15] Nanpeng Yu and Chen-Ching Liu. Multi-Agent Systems and Electricity Markets: State-of-the-Art and the Future. In *Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century (IEEE)*, Pages 1-2, July 2008.

[16] R. H. Bordini, L. Braubach, et al. A Survey of Programming Languages and Platforms for Multi-Agent Systems. In *Informatica*, Vol. 30, Pages 33-44, 2006.

[17] N. Capodieci. P2P energy exchange agent platform featuring a game theory related learning negotiation algorithm. *Master's thesis, University of Modena and Reggio Emilia*, 2011.

[18] Michael Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons Ltd, 2002, paperback, 366 pages.

[19] G. Nguyen, T. Dang, L. Hluchy, Z. Balogh, M. Laclavik, and I. Budinska. *Agent platform evaluation and comparison.* Technical report for Pellucid 5FP IST-2001-34519, June 2002.

[20] *First International Workshop on Agent Technology for Energy Systems (ATES 2010)*. 14 October 2010, Toronto, Canada.

[21] K. Fekete, S. Nikolovski, D. Puzak, G. Slipac and H. Keko. Agent-based modelling application possibilities for Croatian electricity market simulation. *5th International Conference on European Electricity Market (EEM)*. 25 July 2008.

[22] Marta Marmiroli and Hiroshi Suzuki. Web-based Framework for Electricity Market. *Power System and Transmission Eng. Center*, Mitsubishi Electric Corporation.

[23] S. Widergren, J. Sun, and L. Tesfatsion. Market Design Test Environments. *Proc. of 2006 IEEE PES General Meeting*. June 2006.

[24] I. Praca, C. Ramos, and Z. Vale. MASCEM: A Multiagent System that Simulates Competitive Electricity Markets. *IEEE Trans. Intelligent Systems.* Vol. 18, Pages. 54-60. Dec. 2003.