POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Thesis

A semi-automatic multi agent intelligent system for production processes

Supervisors

Fulvio Corno
Luigi De Russis

Company Tutor

Rosaria Rossini

Candidate

Nunzio Turco

December, 2019

Contents

1 Introduction **1**
1.1 Objective 1

2 Background **3**
2.1 Multi-Agent System 3
2.1.1 FIPA 5
2.2 Recurrent Neural Network 5
2.2.1 LSTM network 6

3 Related Works **8**
3.1 Current MAS frameworks 8
3.2 "Composition" MAS 10

4 NT-MAS **12**
4.1 Objective 12
4.2 Architecture and list of technologies 13
4.2.1 Agent Management System 15
4.2.2 White Pages 15
4.2.3 Common Agents 16
4.2.4 Server and Dashboard 18
4.3 System Behaviour 19
4.3.1 Registration to AMS 20
4.3.2 Exchange of a message 21
4.3.3 Alarm and faulty machine evaluation 22
4.3.4 Action choice and machine learning 22
4.3.5 Interaction with the user 23

5 Evaluation **26**
5.1 Injection molding 26
5.2 Use case description 26
5.3 Database description 28
5.3.1 Injection molding data 28
5.3.2 Quality check data 30
5.3.3 Error simulation 30
5.4 Network training 34
5.5 System behaviour 37
5.5.1 1 Inspector - 1 Producer 38
5.5.2 1 Inspector - 3 Producers 41
5.6 Results 43

6 Conclusions and Future Work **44**

Chapter 1

Introduction

The rise of **Industry 4.0** has led factories to focus on connectivity and interactions among machines and people. This new technology enables faster, more flexible and more efficient processes to produce higher-quality goods at reduced costs. In this context, the **multi agent system** ability to communicate finds a relevant role. Nowadays, Multi agent systems are used in a wide range of applications, such as in industry optimization, game playing, market simulation, especially in case of complex scenario, where a distributed approach can help to simplify the problem. But in most of these applications, their intelligence is limited only on the communication ability, this is especially true in case of industrial scenario, where due to the low risk policy, a complete automation of the process is venturesome and risky.

1.1 Objective

The objective of the thesis is to design, develop and test a multi-agent intelligent framework in the context of industry 4.0, aiming at improving different aspects of the production process, such as anomaly detection and classification, advise actions and facilitate monitoring of the machines in order to improve operator's work as well as reducing waste.
The developed framework is named NTMAS, it incorporates a real-time data analytics capacity, exploiting machine learning and provides a web based dashboard for data visualization. Nowadays there exists a variety of multi-agent framework, some of them are flexible and powerful, but usually they lack of the intelligent part, the proposed platform instead already integrates a machine learning algorithm that can be used by the agents to take actions based on the data they are perceiving. The NTMAS provide a partial automation of the process exploiting a Human-in-the-loop model, in this way the operator has a complete control on what is happening on the plant. Each agent's action must be validated by the supervisor before it takes

effect and it is also a way to improve agent's decision. The NTMAS was tested in a partially simulated environment with an unsupervised dataset, so as to be able to perturb the data and analyse the consequent agent behaviour. The goal of these tests is to control and monitor the ability of the agents to adapt to the context in which they are deployed. Furthermore, NTMAS comes with a visualization tool that eases the control and the monitoring of the process, the resources and the agent behaviour through a specific dashboard. Lastly, an evaluation of the possible impact of this platform on the production process is done.

Chapter 2

Background

This chapter contains an introduction of what a multi agent intelligent system is, by explaining its property, its advantages and how it has gained importance in recent years with the growing of the Industry 4.0.

2.1 Multi-Agent System

The multi agent intelligent system is an enhanced version of a multi agent system (**MAS**). It represents a powerful model to solve distributed computation tasks, where a single agent fails to complete or requires a lot of complexity. In this organization of agents, entities can have different sub-goals, but they all cooperate to achieve a common one. But what is an agent? There is no universally accepted definition of this term, indeed more than one description exists today. As reported in [6] there are many definitions in the literature, they are:

- A software agent is an autonomous software entity that must be able to perceive its environment through sensors and act upon it with effectors.[2]
- "Agent is a piece of software which performs a given task using information gleaned from its environment to act in a suitable manner so as to complete the task successfully. The software should be able to adapt itself based on changes occurring in its environment, so that a change on circumstances will still yield the intended result." [3]
- "An autonomous agent is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future." [1]
- "It is a hardware or (more usually) software-based computer system that enjoys the following properties: - autonomy: agents operate without the direct

> intervention of humans or others, and have some kind of control over their actions and internal state; - social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language; - reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it; - pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative." [9]

The last definition has been widely accepted since it is the most complete one. MAS

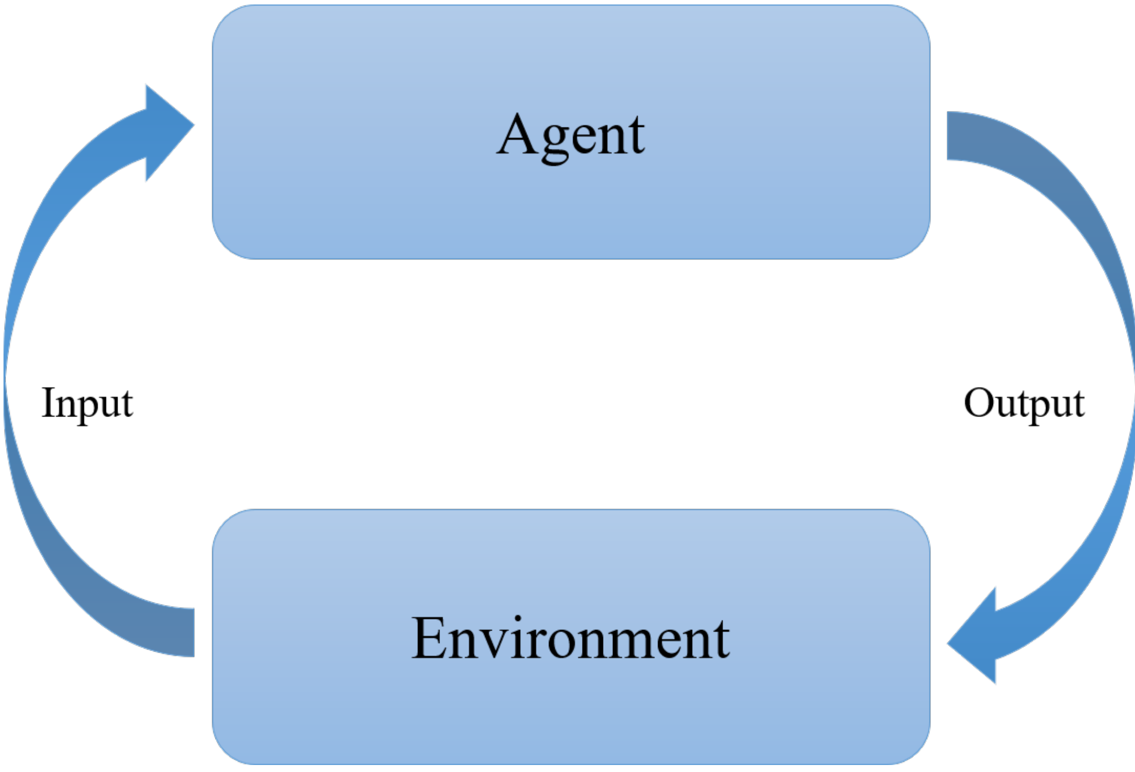

Figure 2.1: Agent cycle

can be view as a collection of distributed autonomous agents, capable of accomplishing complex tasks. Usually, MAS does not incorporate intelligence by default, what gives intelligence is the ability of these agents to interact with each other, in a cooperative way or non-cooperative one and the ability of autonomous decision they have. But these decisions are not always moved by a machine learning method, instead they could be just a simple IF-THEN expression.

2.1.1 FIPA

The Foundation for Intelligent Physical Agents (FIPA) is an organization that has defined a set of standards for agent-base technologies as Multi Agent System. These definitions allows the interoperability between agents and facilitate their development. In particular the FIPA Agent Platform defines the structure of an agent system model, it is composed of:

- Agents, the fundamental actor inside the system
- Agent Management System (AMS), has control over access to the system by offering a white pages service to other agents
- Directory Facilitator (DF), it is an optional component, provides the yellow pages services to other agents
- Message Transport Service (MTS), the communication method exploited by the agents
- Agent Platform (AP), the physical infrastructure in which the agents are deployed

Apart from these components, the FIPA also defines a series of Agent Communication Language, a message schema that each agent has to follow in order to let the system to be extensible and allow the integration of other agents. But, for the scope of the thesis only architectural specifications have been followed.

2.2 Recurrent Neural Network

A recurrent neural network, also known as **RNN**, is a class of artificial neural network which presents a connection between output states and input states. This allows it to exhibit temporal dynamic behaviour. Unlike feedforward neural networks, this network uses its internal state as a memory to process sequences of inputs. It can remember the past and its output is influenced by what it has learnt from the past. Basic feed forward networks can only learn during training, but RNNs can also learn from prior inputs while generating the outputs, because they store some information inside the neuron. Here there is a picture of an RNN:

In this diagram a series of neural network, A, takes some input xt and outputs a value ht. A loop allows this information to be passed from one neuron to the next one. This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to

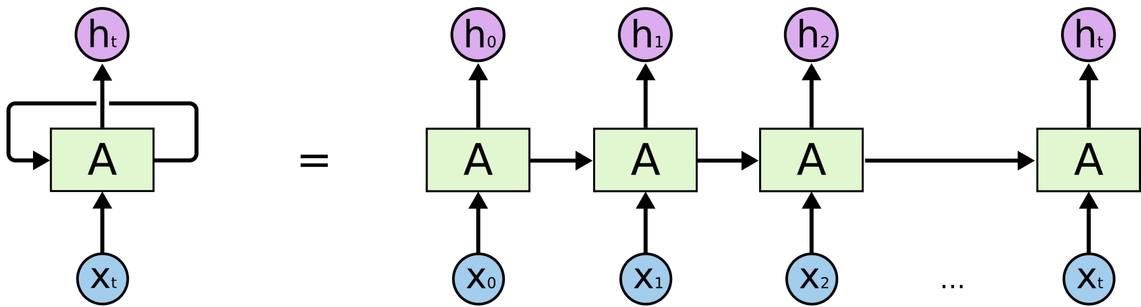

Figure 2.2: RNN-unrolled [8]

use for such data. In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modelling, translation, image captioning and so on. Essential to these successes is the use of “**LSTMs**,” a very special kind of recurrent neural network which works, for many tasks.

2.2.1 LSTM network

Long Short-Term Memory network is a type of recurrent neural network that produces an output based on the previous and the current input. It is explicitly designed to avoid the long-term dependency problem. It is not a variant of RNN, it introduces changes to how the outputs and hidden states are being computed using the inputs. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.
Like RNN, LSTM also have a chain structure, but instead of having a single neural network layer, there are four:

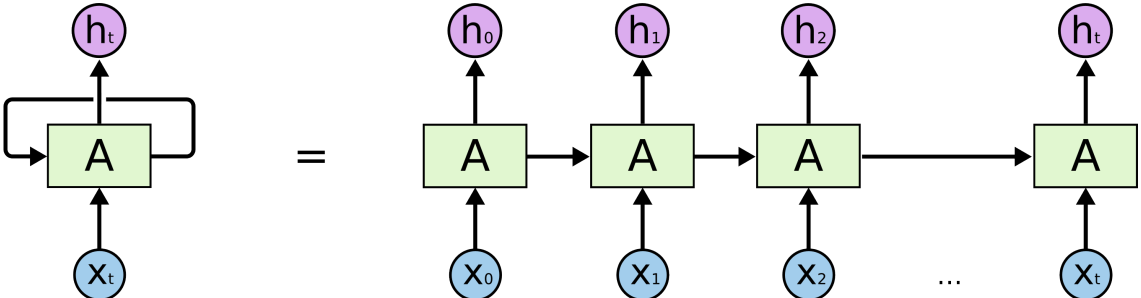

Figure 2.3: LSTM components [8]

A common LSTM unit is composed of:

- Cell state
- Input gate
- Output gate

- Forget gate

The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. LSTM networks are well-suited to classifying, processing and making predictions based on time series data, since there can be lags of unknown duration between important events in a time series. These components inside the LSTM can regulate the flow of information, make it possible to learn which date from a sequence has to be taken and which discarded and by doing that it learns how to use relevant information to make predictions or classifications.

Chapter 3

Related Works

3.1 Current MAS frameworks

Multi-agent systems have been widely studied in literature, today there exists different framework for the implementation of a MAS. An evaluation of these framework was needed before deciding how to implement NT-MAS. Among the many platforms available on the internet, a list of candidates that might have a good set of features and requirements have been selected, they are listed here.

JADE: (Java Agent DEvelopment Framework) is a software Framework fully implemented in the Java language. It simplifies the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications and through a set of graphical tools that support the debugging and deployment phases. A JADE-based system can be distributed across machines (which not even need to share the same OS) and the configuration can be controlled via a remote GUI. The configuration can be even changed at run-time by moving agents from one machine to another, as and when required. JADE is completely implemented in Java language and the minimal system requirement is the version 5 of JAVA (the run time environment or the JDK).[4]

JIACv: (Java-based Intelligent Agent Componentware V) is a Java-based agent architecture and framework that eases the development and the operation of large-scale, distributed applications and services. The framework supports the design, implementation, and deployment of software agent systems. The entire software development process, from conception to deployment of full software systems, is supported by JIAC. It also allows for the possibility of reusing applications and services, and even modifying them during runtime. The focal points of JIAC are distribution, scalability, adaptability and autonomy. JIAC V applications can be developed using extensive functionality that is provided in a library. This library consists of already-prepared services, components, and agents which can be integrated into an

application in order to perform standard tasks. The individual agents are based on a component architecture which already provides the basic functionality for communication and process management. Application-specific functionality can be provided by the developer and be interactively integrated.[5]

SARL: SARL is a general-purpose agent-oriented language. SARL aims at providing the fundamental abstractions for dealing with concurrency, distribution, interaction, decentralization, reactivity, autonomy and dynamic reconfiguration. These high-level features are now considered as the major requirements for an easy and practical implementation of modern complex software applications. Considering the variety of existing approaches and meta-models in the field of agent-oriented engineering and more generally multi-agent systems, this approach remains as generic as possible and highly extensible to easily integrate new concepts and features. The language is platform- and architecture-independent.[7]

SPADE: (Smart Python multi-Agent Development Environment) is a Multiagent and Organizations Platform based on the XMPP/Jabber technology and written in the Python programming language. This technology offers by itself many features and facilities that ease the construction of MAS, such as an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on XML, just like FIPA-ACL.

Composition: A MAS implemented in a European project started at Links Foundation. It is a fully distributed multi-agent system working in a marketplace scenario, designed to support exchanges between involved stakeholders. It is particularly aimed at supporting automatic supply chain formation and negotiation of goods/data. It is FIPA compliant and uses AMQP as message protocol. Even if it is a complete system, not just a framework, its structure allows an easy re-implementation of each agent behaviour.

A set of Feature requirements is listed here:

- Python as programming language, because of the huge support to machine learning.
- Distributed approach, each agent has to be deployed on a different machine.
- Interactive User Interface, that allows humans to change communicate with the agents.
- FIPA compliant is considered as a PLUS. It facilitates the scalability and communication among agents.

Here there is a comparison table:

Framework:	Language:	Distributed approach:	Interactive UI:	Additional features:
Composition	Python	YES	-	FIPA compliant
JADE	Java	YES	YES	FIPA compliant
JIACv	Java	YES	-	-
SARL	Java	YES	-	-
SPADE	Python	YES	-	FIPA compliant

At the end it has been decided to modify and extend the COMPOSITION system to our scope. This choice is motivated by the fact that this system is similar to SPADE, the one that seems to offer the more complete set of features, but it is developed inside Links Foundations and so the support to it is higher, moreover this approach allows to face directly with some criticisms concerning MAS and also a more complete understanding of these systems. Moreover, Composition already implements configurations file which ease the deploy of different agents.

3.2 "Composition" MAS

The COMPOSITION marketplace is a fully distributed multi-agent system designed to support exchanges between involved stakeholders. It is particularly aimed at supporting automatic supply chain formation and negotiation of goods/data. The COMPOSITION marketplace exploits a microservice architecture (based on Docker) and relies upon a scalable messaging infrastructure based on RabbitMQ. The System is divided in two main categories of agents that can be defined a priori, depending on the kind of services provided:

- Marketplace agents
- Stakeholder agents

The first ones are the agents providing the services that are crucial to the marketplace to operate, they build the infrastructure of the system. While the second ones are the agents exploiting the infrastructure for exchanging goods, they can be divided in other two different categories: Requested and Supplier, that from an implementation point of view, they are very similar and share a large set of features, especially the communication protocols used for the interaction with stakeholders and other agents. It works as follows:

1. Both Requester and Suppliers subscribe to the system.
2. Requester initiate a new supply chain and waits for proposals.

3. Suppliers build an offer to the requester and wait for the confirmation.
4. Requester send to the user a list of the highest offers.
5. Requester receive the response of the user and closes the deal by sending confirmation to the chosen supplier.

At the present Composition does not implement any kind of intelligence.

Chapter 4

NT-MAS

4.1 Objective

The final objective of the **NT-MAS** is to improve production aspects related to industry scenario. With this in mind, a set of features and requirements were taken into consideration during the implementation of such a system. Inside the industry a wrong decision can be worth a lot of money, so it is important to take the right decision and be as less risky as possible. Introducing a complete automatic system inside a factory managed by human employer is tricky. The machine learning is based on known data, it can take the right action in case of a known situation, but what if the situation that it is facing is never happen before? In this case a decision taken by an agent could be wrong. A human supervisor must always be present in these situations, in order to take control and perform the correct move.

NT-MAS was developed to allow human supervisors to always monitor the system and to take control of each agent action. This semi-automatic system has two main benefits, on one hand it helps the human in monitoring the process and in the decision making, on the other this method improves each agent decision, since it will keep learning from each human's validation.

To allow the user to interact with the system, a dashboard is also attached to the NT-MAS. Inside this dashboard the relevant data of each agent are displayed and in case actions are required a notification will appear allowing user to have the last word on each decision.

NT-MAS was thought to be used inside a production industry line, where communication and coordination between production and inspection machines are essential in optimizing the process. This task is often done by the human, which monitors both machines and acts in case of fault. But it is not so easy for a human to maintain control if the size of the production process increases. NT-MAS is designed to be scalable, in fact it automatically detects the number and the type of

agents active inside the system. Each agent can be deployed on a single machine and in case the number of machines varies, the system can easily adapt to it.

The developing of NT-MAS starts from "**Composition**", a Multi Agent System presents at Links Foundations. Composition was created to be scalable and generalizable, moreover it provides a distributive approach and it is **FIPA** compliant. This system has been modified to suit the industry scenario and beside of the previous features, some new functionalities were added to the agents, such as integration of online machine learning, communication with a dashboard for displaying the data, implementation of the interaction with the user and also a different storage system based on a My-SQL database.

4.2 Architecture and list of technologies

In this section is shown the architecture of the system and the technologies used to implement it.

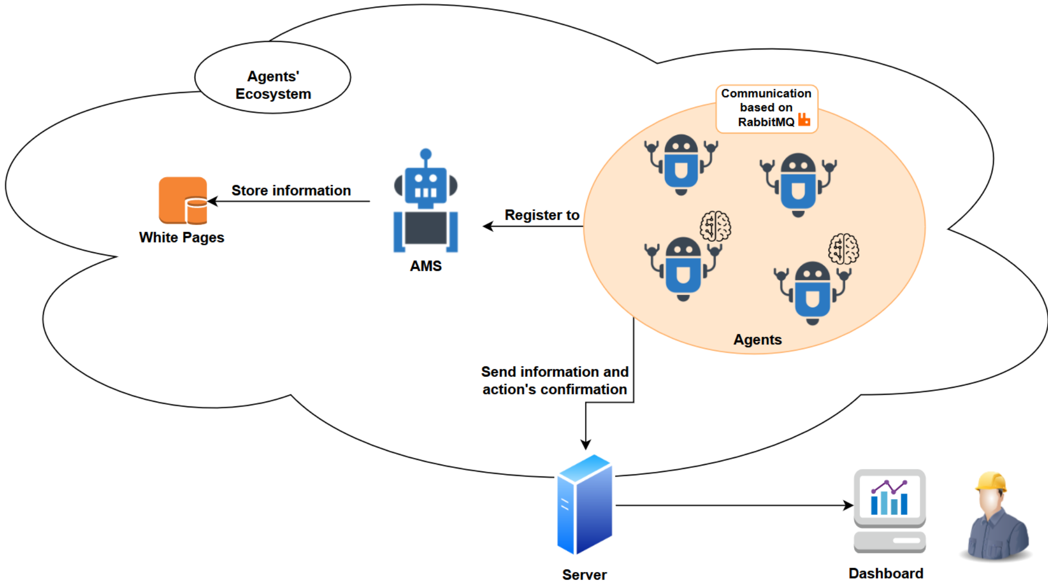

Figure 4.1: NTMAS Architecture

- **Agents**: they are the ones that perceive the environment and act upon it. They are deployed upon a different machine inside the production process. They are divided in producer and inspector.
- **Agent Management system (AMS)**: A particular type of agent, it does not interact with any machinery, but it controls the agents inside the system, allowing them to know each other.

- **White Pages**: It is a database to store agents' information. It is only accessed by the AMS.
- **Server**: It stores the data collected by each agent and allows them to communicate with the dashboard and retrieve user interaction.
- **Dashboard**: A web interface to allow data visualization and user interaction.

Each agent is deployed inside a docker container, this allows the system to be distributive and to maintain a clean and autonomous environment. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. The communication between these agents is done through **RabbitMQ** message broker, which is based on **AMQP** message protocol, an alternative to MQTT protocol. **AMQP** stands for Advanced Message Queuing Protocol, it is a messaging protocol that enables conforming client applications to communicate with conforming messaging middleware brokers. **AMQP** works as follows:

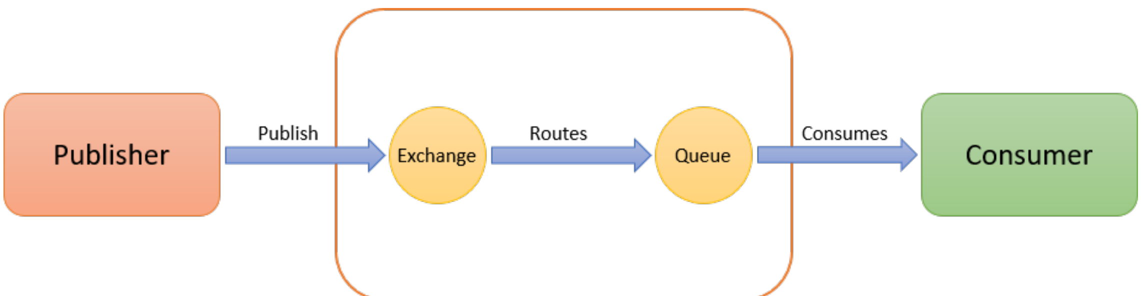

Figure 4.2: RabbitMQ working flow

The core idea in the messaging model in **RabbitMQ** is that the publisher never sends any messages directly to a queue. Actually, quite often the publisher doesn't even know if a message will be delivered to any queue at all. Instead, the publisher can only send messages to an exchange. An exchange is a very simple thing, on one side it receives messages and the other side it pushes them to queues. Different types of exchange exist, in this project two were used:

- **Fanout**: sends message to all the queue bound.
- **Direct**: sends message only to a specific type of queue, based on the binding key.

The core of the system is the Agent Ecosystem, following **FIPA** architecture standard, it is composed by Agent Management System (**AMS**), White Pages service and common Agents. The first two components are the pillars of the ecosystem,

in fact they keep track of any other agent alive and let the agents know each other. Attached to the core part, there is a web user interface supported by a server, which receives agents' data and actions' confirmation. While the server stores data perceived by common agents, the user can monitor the production process through the dashboard, where graphs containing machines' data are shown together with a list of alarms from inspection machine and confirmation of agent decision. In the next paragraphs all the components of the system are introduced.

4.2.1 Agent Management System

The Agent Management System is the first component to be deployed. It is the central point of the system, it manages the communication and controls the presence of other agents. Each agent has to register to AMS in order to communicate and be part of the system. The AMS does not rely upon AMQP, but it runs a flask server, receiving HTTP requests. In particular the AMS provides the following methods:

- **Register_agent**: it is used to register an agent inside the system, generating a new "agent_id" and storing these data inside the white pages.
- **Get_registered_agent**s: this method returns a list of all agents currently alive in the system.
- **Deregister_agent**: this method removes an agent from the system and deletes its data on the white pages.

4.2.2 White Pages

The White Pages service help the AMS in his work, it stores a database with all the online agents information and only the AMS has access to it. This service is based on a MySQL server and it is deployed inside a docker container. The database contains a table called "agents_table" with the following format:

Agent_id	Agent_role
The unique identifier for the agent. It is the primary key for the table, so it has the constraint of being unique.	The role of the agent inside the system.

The Agent_id uniquely identifies the agent inside the system, it is also useful for implementing the communication queue with RabbitMQ.

The Agent_role describes the type of agent, it can be Inspection or Producer.

4.2.3 Common Agents

The common agents are the ones deployed upon the industrial machine. NT-MAS has two different types of common agents: Inspector and Producer. These are deployed upon different types of machine inside an industry. They have different behaviour and sub-goal. They both derive from a base class called "capAgent", then each of them implements a different behaviour. The first thing done by both of them is the registration at the AMS in order to receive a unique agent_id. Once this step has been successfully completed, the "activate" method is called by each agent, which connects them to the RabbitMQ broker and they start listening to incoming messages.

Each agent can listen to two exchange types:

- **Fanout**: used to communicate with all agents inside the system.
- **Direct**: based on agent_id, used to communicate among specific agents.

Moreover, each agent comes with a flask service interface that allows programmers to interact with them while they are running.

Inspector Agent

The inspector agent is the one deployed on the quality inspection machine. It can monitor the quality of the good produced, especially the number of faulty pieces and sends alerts in case this last number goes above a given threshold. The threshold can be set inside the configuration file loaded at the agent start-up. This class has two main methods:

- **Start_reading_data**: the agent starts reading the data of the inspection machine and checks if the number of defects is below the given threshold.
- **Send_alarm**: this method is invoked to send an alarm message to all the other agents inside the system.
- **On_message**: this method is called each time a new message arrives to Inspector queue. It displays the message on console.
- **Send_data_to_server**: this method is called at each cycle of "start_reading_data". It sends the current data to the server storage.

The alarm message is a **JSON** string with the following schema:

```
{
   "description" : "An alarm message from an inspector agent",
   "type" : "object",
   "properties" : {
      "agent_id" : {
      "description" : "The unique identifier of the agent",
      "type" : "string"
      },
      "type" : {
         "description" : "Type of the alarm",
         "type" : "string"
      }
   }
}
```

Producer Agent

The producer agent is the one deployed on the production machines, the one responsible of creating or modifying the goods. This agent keeps track of the machine variables and incorporates the intelligent part of this system. This class has the following methods:

- **Start_reading_data**: this method is called as soon as the producer registration is completed. The agent starts reading the data from the production machine.
- **Generate_error_vector**: this method is invoked in case an alarm from the Inspector is received. It calculates and sends the current error vector to other producers.
- **On_message**: this method is called in case of a new message received. It checks the content of the message and calls on_error_vector_received or send_message based on the sender of the message.
- **On_error_vector_received**: this method is called each time a new error vector arrives. It keeps track if all the other producers have sent their status, if so, it checks for the highest error vector and in case calls the evaluate_action_ML method.
- **Evaluate_action_ML**: this method uses the machine learning model to predict the action to undertake.

- **Send_data_to_server**: this method is called at each cycle of "start_reading_data". It sends the current data to the server storage.

The **Error Vector components** are defined as:

$$y_i(t) = x_i(t) - \frac{1}{J} * \sum_{j=1}^{J} x_i(t-j)$$

Where:

- $x_i(t)$ is the current value of the i-th variable.
- $x_i(t-j)$ is the value of the i-th variable at time $(t-j)$.
- J represents the window of values considered in the mean.

The producer will subtract from the current value of each variable the mean of the lasts J values. It has been decided to use this simple mathematical method, because this can evaluate how much the current behaviour of the machine has changed compared to past values. If a fault has happened, in most cases, it will affect the machines variables, so it is possible to detected it by looking at how much these variables changed with respect to the past.

This operation is done each time an alert is received, next the calculated Error Vector is sent to all the other Producer agents inside the system, exploiting the direct exchange.

Each Producer waits until all the Error Vector of the other Producers have been received, next the agent with the highest sum of the Error Vector components will be elected and it will take a decision basing on a Recurrent Neural Network, called LSTM network. This is done by calling the "evaluate_action_ML()" method inside the "capProducerAgent" class. The result of this function represents the action chosen by the algorithm.

The chosen action is sent, together with all the other possible ones, as a Json string through HTTP POST request to the server. At this point the producer will wait for a confirmation from the user before taking the action. The response is also used by the agent to update the LSTM network.

4.2.4 Server and Dashboard

Inside **NT-MAS**, the server is used to store all the data of the production process. In this way the agents inside the system can use less resources. A client, running the dashboard, connects to the server and retrieves all this information to show them to the user. An example is shown below:

Figure 4.3: NTMAS Dashboard

The dashboard also allows to keep track of all the faults happened inside the production process, as can be seen from the figure below:

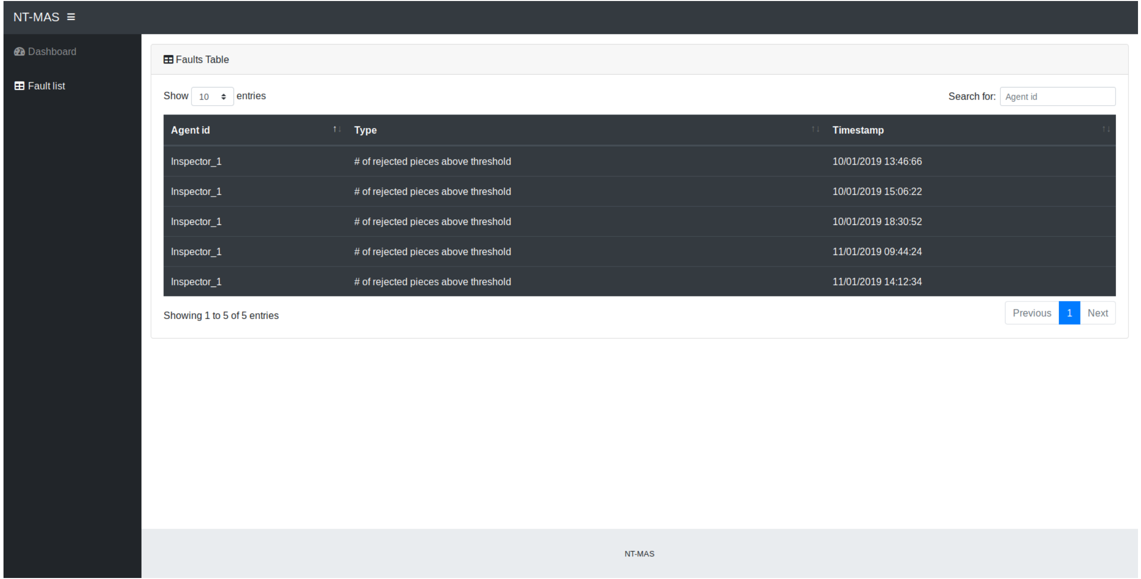

Figure 4.4: NTMAS Fault-list (Dashboard)

4.3 System Behaviour

In the next paragraphs there is a deeper description of the steps performed by the **NT-MAS** components.

4.3.1 Registration to AMS

This is the first action done by a new agent. It is required to register to Agent Management System in order to announce itself and be part of the system.

The operation is done following these steps:

- Agent creates a Json string, following his configuration parameters
- Agent sends this Json string through a POST request at “¡AMS_address¿/register/agent”
- AMS receives the requests and creates a unique agent-id which is sent back to the activating agent
- AMS also stores the new agent information inside the White Pages service, using a MySQL connector

The schema of the **JSON** message sent by the agent is the following:

```
{
   "description" : "A message used by user to register an agent",
   "type" : "object",
   "properties" : {
      "agent_id" : {
      "description" : "The unique identifier of the agent, empty",
      "type" : "string"
   },
      "agent_role" : {
         "description" : "An agent can be either requester or supplier",
         "type" : "string"
      }
   }
}
```

It is important to notice that the agent_id field will be empty, since the agent does not have an identifier when the call is performed, it will be generated and filled by the AMS and sent back to the agent.

The agent class also provides a method to de-register an agent. Similarly to what is done with the registration, in case an agent leaves the system it has to call the “deregister()” method. A POST request is made at “¡AMS_address¿/deregister/agent” with the following **JSON** schema:

```
{
   "description" : "A message used by user to deregister an agent",
```

```
    "type" : "object",
    "properties" : {
        "agent_id" : {
            "description" : "The unique identifier of the agent",
            "type" : "string"
        }
    }
}
```

The AMS removes from the White Pages service the entry associated with the agent_id received.

Here there is a figure showing the activation of a new agent inside the system:

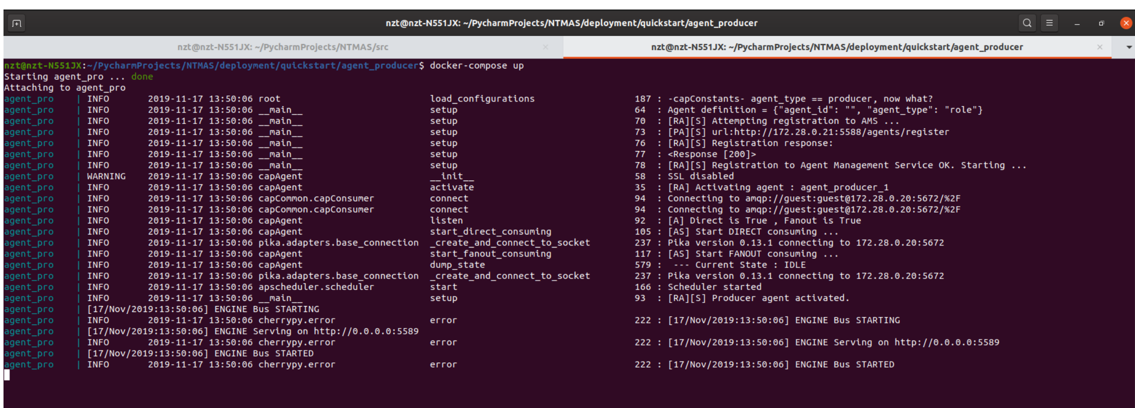

Figure 4.5: NTMAS: registration example

As can be seen from the figure, the first step done after loading the configuration file is the registration, next the agent will connect to the two AMQP exchanges in order to send messages to other agents.

4.3.2 Exchange of a message

The agent, when initialized for the first time, will call the method "activate()". Inside this method, two new connections are created, one for a "directExchange", another for "fanoutExchange". Moreover, for each connection a new thread is lunched, it will consume incoming message from his connection.

The communication with other agents is done by means of the "start_publishing(message, exchange, queue)" method inside the base class "capAgent". Where:

- **Message**: is a string representing the payload
- **Exchange**: is the name of the exchange to which forward the message, can be fanout or direct

- **Queue**: is the name of queue of the receiver to which the message is delivered, it is represented by the agent_id

Each agent has a unique queue to receive messages. Once that a message is delivered by an exchange to this queue, a new thread will handle it, by calling the "on_message()" method from the "capAgent" class. In NT-MAS both producer and inspector agents override this method.

4.3.3 Alarm and faulty machine evaluation

In case of normal condition, no messages are exchanged between agents. While in case of alarm producers and inspector start to communicate with the following schema:

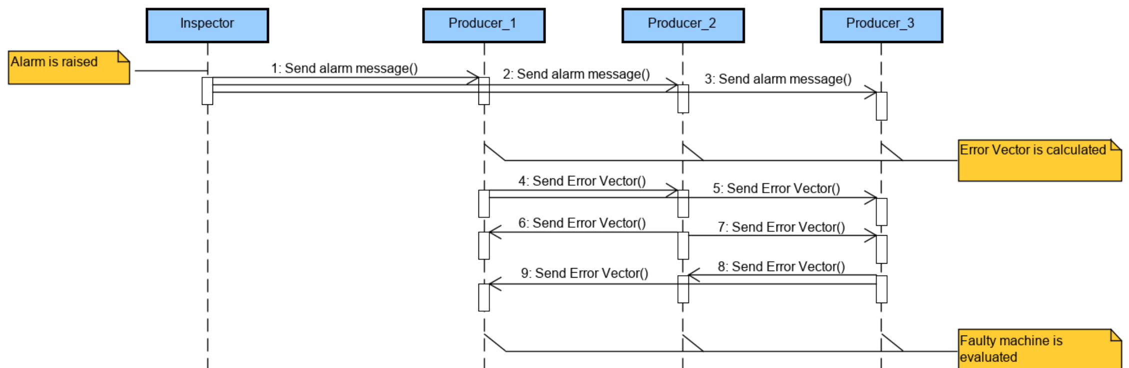

Figure 4.6: NTMAS: sending error vector

When an alarm is raised, the Inspector agent sends it to the producer, using the fanout exchange. Once the alarm arrives to the producers, they will start to calculate the current error vector and then they will send it as a message through a direct exchange, so each agent will get the error vector of each other producer. Next, the highest error vector is evaluated and in this way the faulty producer is known by everyone.

4.3.4 Action choice and machine learning

The producer agent with the highest error vector will use its current data to choose an action, in particular it will rely upon a machine learning model, called Long Short-Term Memory network. This particular type of network allows agents to select an action relying not only on the current input, but also on the previous ones.

It is important to notice that, other algorithms have been taken into account for this system, such as Random Forests and Mondrian Forests. But the LSTM was the only one able to provide an easy online learning and a high customizability.

This network has some requirements: it takes as input a matrix containing the lasts N values of V variables The structure of the tree is the following:

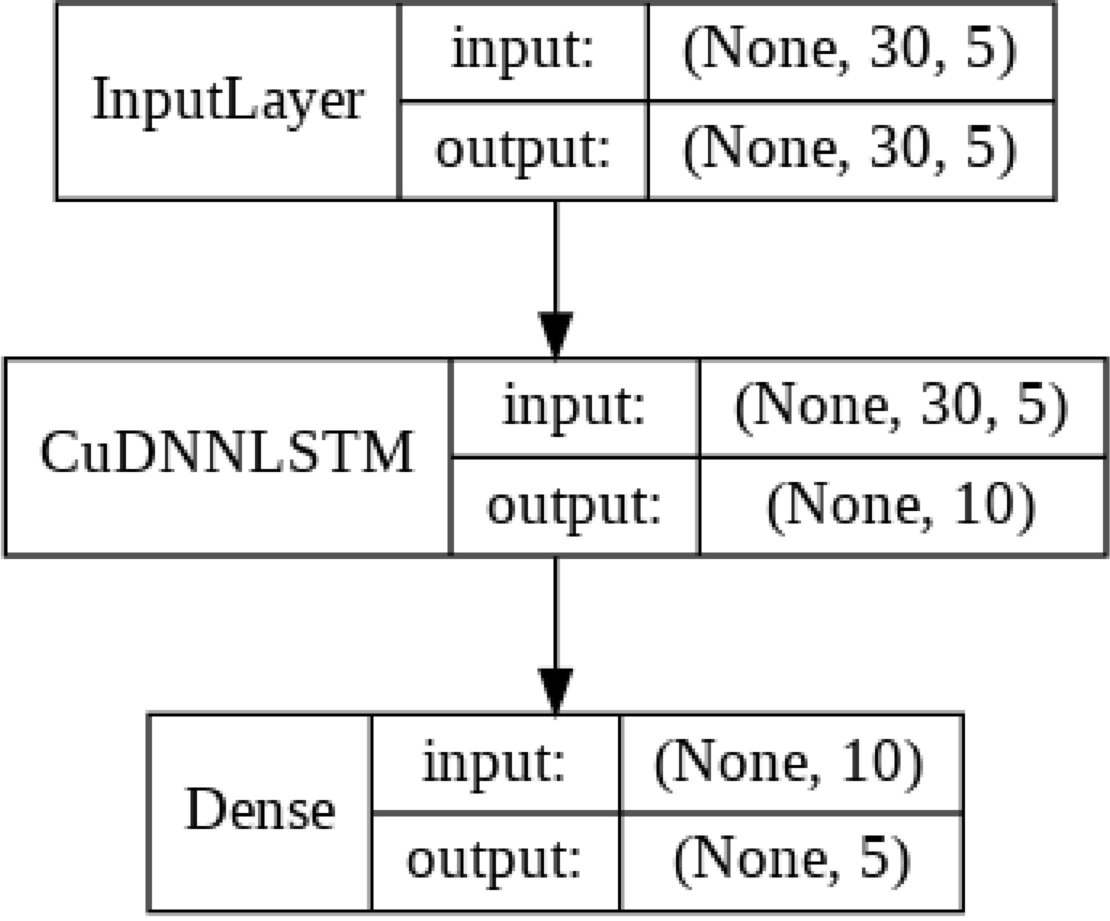

Figure 4.7: network model

This network takes as input the last **N** values for each variable and gives as output a classification of the action required for that state. Moreover, each decision of the user will update the weights inside the network, so that the algorithm can adapt better to the deployed scenario. It is important to point out that the model can be directly loaded from a ".h5" file, so that one can have an already trained model and keeps updating it with the user's decisions.

4.3.5 Interaction with the user

In this section it is shown how the interactions with the user take place. The client has a JavaScript code that keeps asking for the "agent_action" list to the server, initially empty. Once that the producer agent sends an action request to the server, this will save the request content inside "agent_action" list. At the next request, the client will get a non-empty string that will trigger a "div" container in which the

producer's action request is displayed.
Here an example is shown:

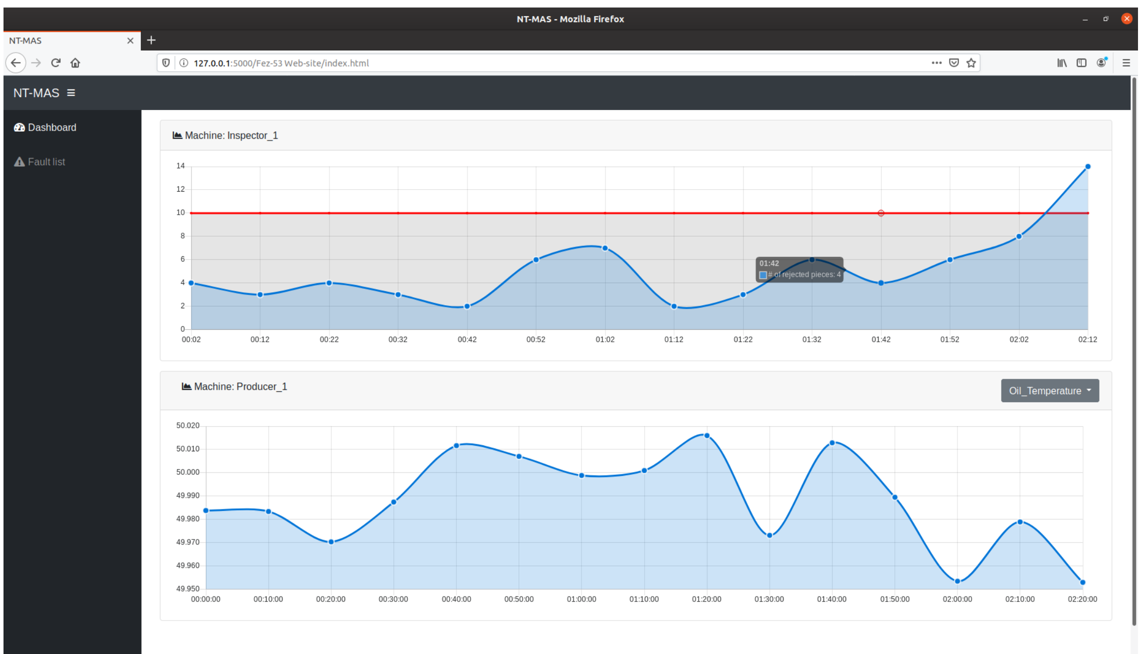

Figure 4.8: fault on inspection

In this example the production is having a fault because the variable monitored by the inspector agent is above the line threshold. At this moment the producer agents starts to find a solution to this error. An example of solution with possibility of choice by the user can be seen here:

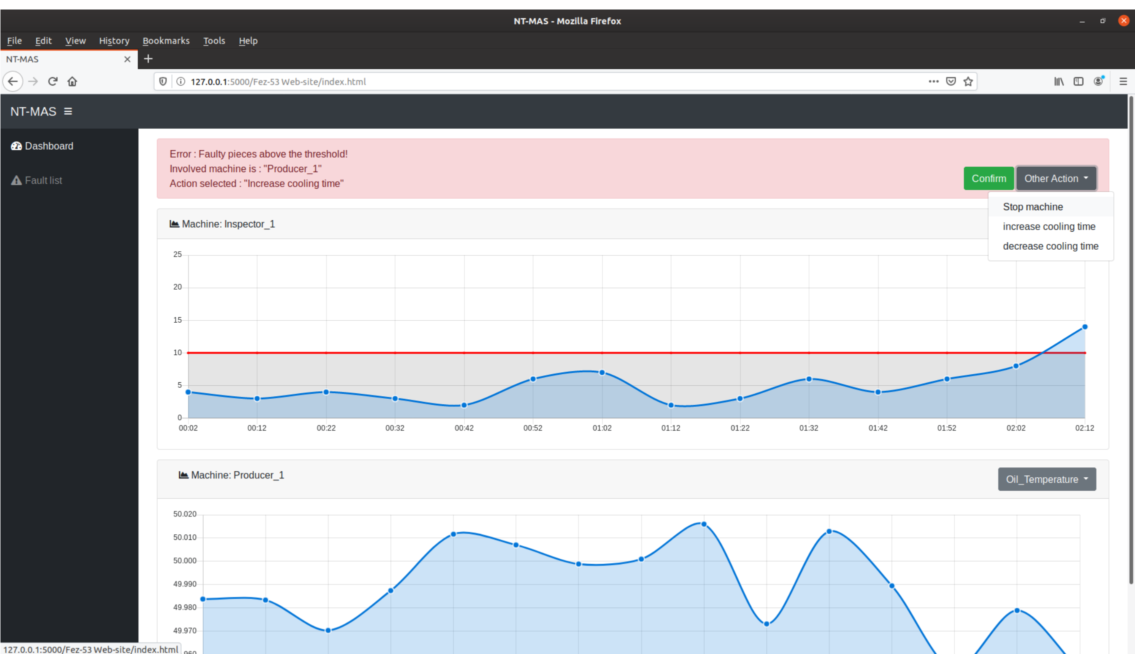

Figure 4.9: fault, action is possible

As can be seen, inside the error section there are two buttons, the green one will confirm the action decided by the producer, while the other contains a dropdown menu with all the other possibility for that particular agent. Once that a button is clicked, the "agent_action" list is emptied and a HTTP POST request containing a Json response is sent to the asking producer.

Chapter 5

Evaluation

In this chapter is reported a demonstration of how the NTMAS behaves in a production environment. The data analysed were provided by Links Foundation and they come from a European project concerning an industry producing coffee's capsules with injection molding technique.

5.1 Injection molding

Injection molding is the most commonly used manufacturing process for the fabrication of plastic parts. This process requires the use of an injection molding machine, raw plastic material, and a mold. The plastic is melted in the injection molding machine and then injected into the mold, where it cools and solidifies into the final part.

5.2 Use case description

The considered factory produces coffee's capsules and lids. Those items are made of PLA, a completely biodegradable and compostable plastic type.
The scenario is composed of:

- 3 Injection molding machines
- 1 Quality check machine

The first 3 injection molding machines are the one that actually produce the capsules, while the quality check machine instead performs a control on each piece and discard the defective ones.
The production machines are connected with the quality machine by means of a series of conveyors belts. Each producer has a conveyor belt that transport the products in another larger conveyor belt, common to everyone. In this belt all the

pieces get mixed, so it is impossible for the quality check machine to know by which injection machine the pieces are produced, especially in case of faulty pieces. This leads to a hard work for the human supervisor, who has to monitor every step of the production process.
During the long-term manufacturing of the coffee capsules, parameters of the injection molding machines can slightly change due to various changes of the environment and/or machine components (temperature and humidity in the factory, deviations in the energy supply system, overheating, deviations in the quality of the plastic granules, wearing of machine parts, components breakdown). These changes give rise to production of faulty pieces. In order to counteract to this, some actions have to be performed by the supervisor, who constantly monitors the production line. Unfortunately, due to the distance between the machines, the process of understanding which machine needs a parameters update is slow and wearisome, as a result a lot of faulty plastic pieces are produced, before a countermeasure is taken.

Figure 5.1: The real factory where data coming from

In this scenario a multi agent system could help to monitor the process variables,

it also could automize the entire production line by exchanging information between intelligent agents, but a complete automatization is tricky and risky. What if something unpredictable happens? A lot of waste will be produced and time and money will be lost. The ability of NTMAS to leave to the supervisor the last word on each decision can help to avoid unwanted behaviour of the system. Due to the characteristics of the machine involved in the considered scenario, it has been decided to deploy the Producer agent on the injection molding machine and the Inspector agent on the quality check machine.

In the next section is presented the data available and how the NTMAS is adapted to this scenario. The objective is to point out the behaviour of the developed multi agent intelligent system inside a production process and the capabilities that this system provides.

5.3 Database description

In this section is described database generated by the industry. It contains four datasets of one working day each, one dataset for quality check machine and other three for injection molding machines.

5.3.1 Injection molding data

Each injection machine generates a set of data in each molding cycle. Every molding cycle lasts around 5 to 10 seconds. The complete set of data contains:

Timestamp:	time of the beginning of the cycle
Mold Closing Time:	time taken for the mold to close
Tonnage:	clamp force, which is needed to keep the mold closed during the injection molding cycle
Shot Size:	Position of the screw at the end of the plasticisation phase (shows the max. volume that can be injected)
Shot Length:	Shot Size - Cushion (shows the volume that is really injected)
Transition Position:	screw position where the switch over takes place
Maximum Fill Pressure:	Maximum pressure of the injection molding cycle, this occurs after the plastic melt is strongly compressed, while the screw stops and is moved back until the holding pressure is reached
Hold Pressure Zone - 1:	Pressure for the first plateau of the holding pressure
Hold Pressure Zone - 2:	Pressure for the second plateau of the holding pressure
Back Pressure:	Pressure during the plasticization time
Mold Opening Time:	time taken for the mold to open
Oil Temperature:	temperature of the machine oil
Extruder Temperature -1:	Temperature of the heating belt at extruder position 1
Extruder Temperature - 2:	Temperature of the heating belt at extruder position 2
Barrel Head Temperature:	Temperature at the barrel head
Cooling Time:	time to cool the plastic parts (between end of the holding pressure and the start of the mold opening time)
Injection Hold Time:	time during which the mold is closed and the plastic is injected inside the mold
Cycle Time:	time required for a complete cycle

Among these variables, only a subset of them can be tuned, while others are read only. The modifiable variables are:

- Mold Closing Time
- Tonnage
- Mold Opening Time
- Extruder Temperature -1
- Extruder Temperature -2
- Cooling Time

5.3.2 Quality check data

The quality check machine analyses a group of capsules each 10 seconds. Every cycle gives this information:

- Timestamp
- Number of pieces analysed
- Number of good pieces
- Number of defective pieces

For the purpose of the system, only the number of defective pieces is taken in consideration. If this number goes above a given threshold, it is considered as a problem in the production.
Both quality check and injection molding data are extracted from a csv file.

5.3.3 Error simulation

Before deploying the NTMAS, an analysis of the system was made. What came out from it is that the data provided contains some anomalies that are not caused by a problem inside the machine or inside the production, they are neither predictable. It was found out that, every once in a while, the supervisor monitoring the system stops the injection machine for a check. Unfortunately, it is not known which anomalies represents a real error and which not, moreover during those checks some parameters may be tuned and/or some components substituted, making it impossible to understand the natural behaviour of the system. An example is given by the figure below, where the spikes represents a machine restart.

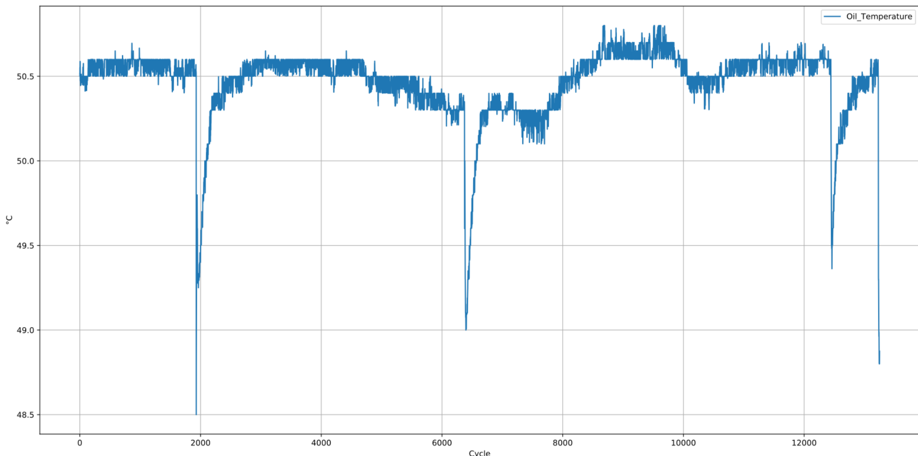

Figure 5.2: Original data

This situation requires a deeper analysis and knowledge of the process that are beyond the scope of the thesis. So it has been decided to remove these anomalies and simulate, starting from the real data, some errors known by the literature.
The situations that can lead to a production of a faulty piece are many. Some are caused by a breaking of a component, others by a machine overheating, or even a slightly change in the raw material. So, it has been decided to consider 4 types of errors.
They are:

- Overheating of the machine, due to a raise up of environment temperature
- Loss of pressure, caused by a break of a component
- Plastic part stuck in the cavity, in case the capsule falls too slow
- Increase of melt temperature, due to a change in the raw plastic material

Each situation affects only a subset of the injection machine variables, moreover, for each situation an action is required.
Here it is reported a table summarizing everything:

Error	Involved variables	Action required
Overheating	Increase of Oil_Temperature	Increase Cooling_Time
Loss of pressure	drop of Max_Fill_Pressure	Stop the machine
Plastic part stuck in the cavity	Jump up of Shot_Length, Jump up of Shot_Size	Stop the machine
Increase of melt temperature	Increase of Barrel_Head_Temperature, Decrease of Max_Fill_Pressure	Reduce Extruder_Temperature

Starting from the original version of the injection data, a cleaning was made and errors were inserted following the previous table. The results can be seen here:

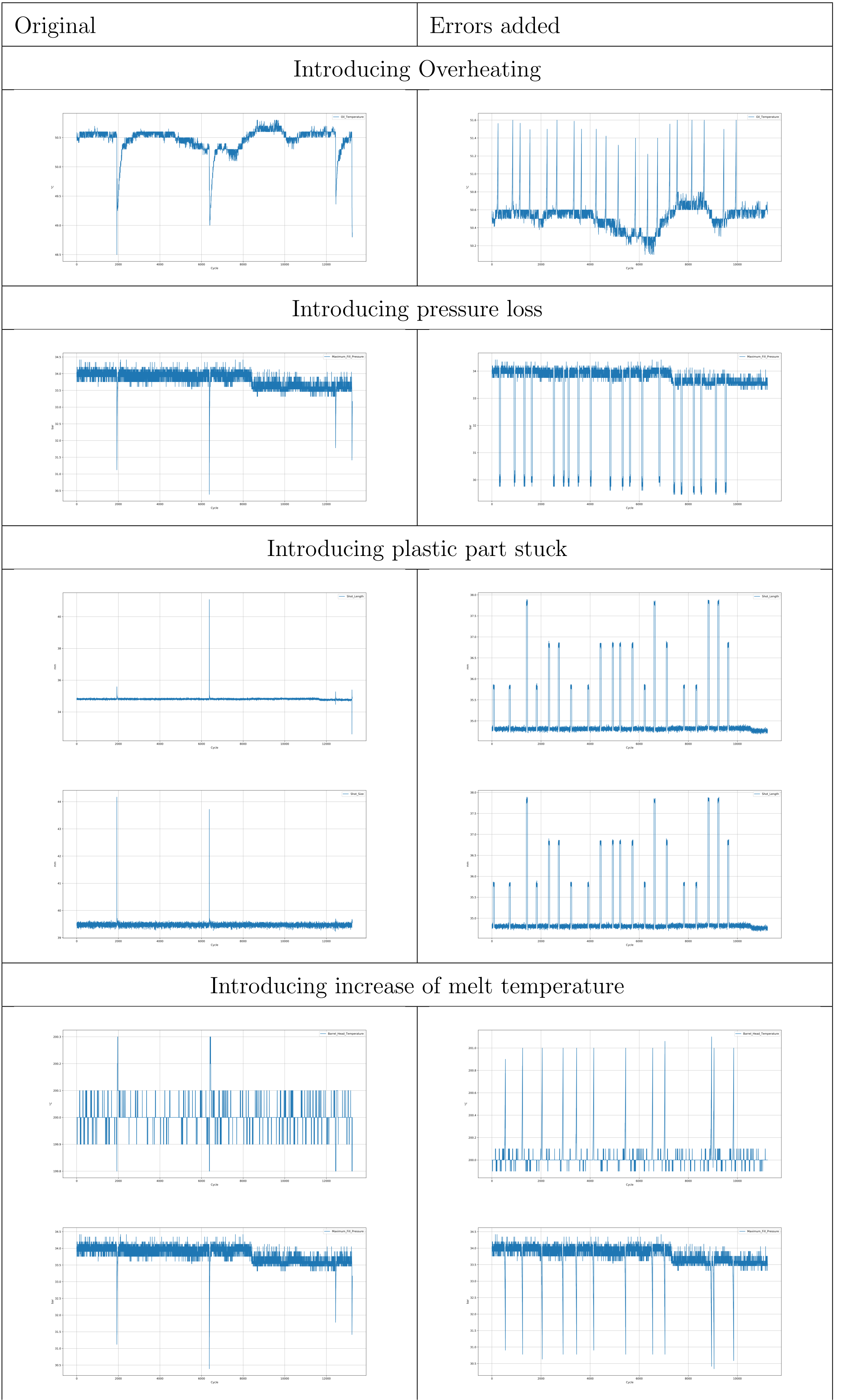

Table 5.1: Simulating errors

It is important to notice that the error inserted are all in different location of the dataset, for simplicity we consider that an injection machine can have only one error at a time. Moreover, the duration of each error is 50 machine cycles, that is around 5 minutes.
To keep track of the inserted errors, an array, called “labels” containing the position and the type of errors was also generated.
In order to simulate the production of waste, also the data of the quality check machine has been changed. An increasing trend has been inserted in the same interval of previous errors. The result is shown here: In the figure the line in orange

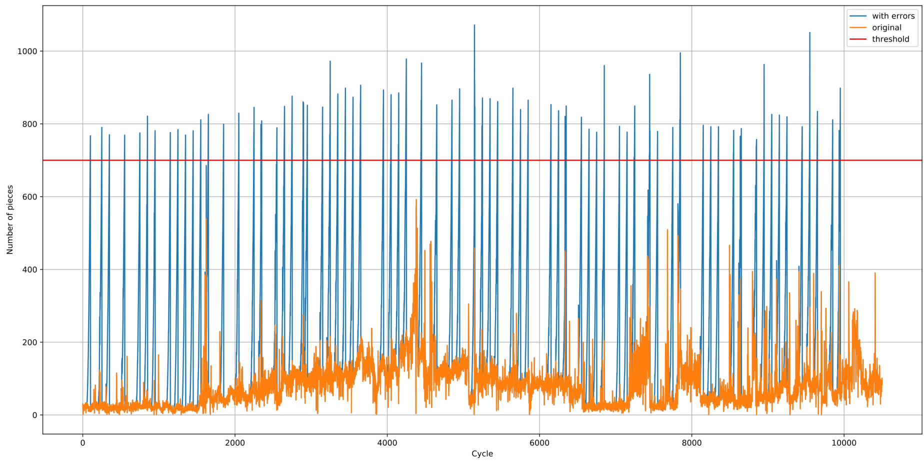

Figure 5.3: Quality check simulation data

represents the original data, while the blue one represents the errors added. Instead, the red line is the threshold at which an alarm should be raised.

5.4 Network training

In this section it is explained how the set-up of the LSTM network was performed. This network exploits only the injection machine data, in particular the subset of variables related with the simulated errors, while the quality check ones are used only by the Inspector agent to raise an alarm.
The LSTM network exploited takes as input the values of each variable in the last 30 cycles, while gives as output an array of size equal to the number of error cases considered, containing the probability of each error.
First of all, the numbers of inputs and outputs have been set. The number of inputs depends on the number of variables affected by the simulated errors, they are:

- Oil_Temperature
- Max_Fill_Pressure
- Shot_Length
- Shot_Size
- Barrel_Head_Temperature

In order to train the model, an input matrix with dimension $[N_cycles, 30, 5]$ was generated. Where the N_cycles is the number of values that are present for each variable, 30 is the size of the window of values at which the LSTM layer will look at and 5 are the number of variables considered.
It is important to note that, before shaping the input matrix, each variable is normalized with a Min-Max method which scales the range of the data to $[0, 1]$, this gives the same importance to all the features and allows to have better performance of the model.
Instead, to model the output, a set of number has been chosen to represent each situation in the following way:

- No error = 0
- Overheating = 1
- Pressure loss = 2
- Plastic stuck = 3
- High melt temperature = 4

These data have been derived from the "labels" array generated during the errors' entry phase. In order to train and test the model, the input data is split into 70% of train set and 30% of test set. Next the model was trained with a batch size = 50 and repeated over 200 epochs, as they were found to be sufficient to learn how to classify correctly the datapoints. The results of the training can be seen in the figures below:

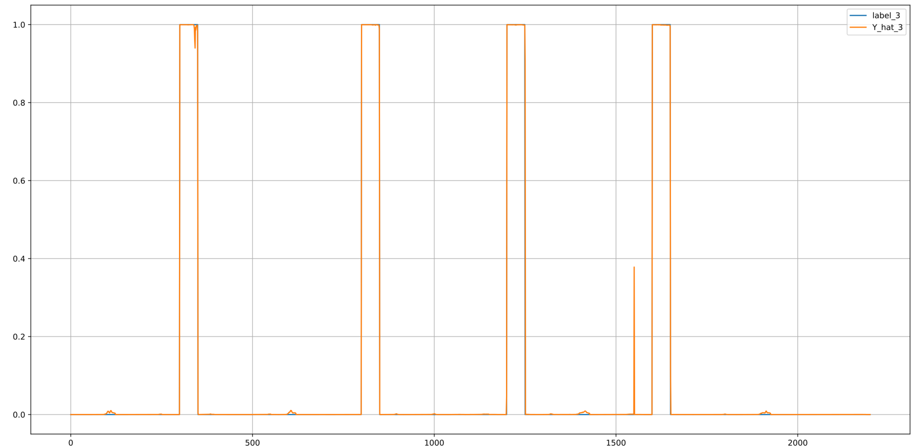

Figure 5.4: Error 3 classification

This is the prediction of the plastic stuck, the orange, which represents the classification of the model, covers almost always the blue line, which represent the true situation. The value of these functions is the probability that a plastic stuck-error is happening in that particular cycle. Moreover, as can be seen the model still has some uncertainty in some cycles, for example around the 1550.
Here all the prediction on the test set are shown:

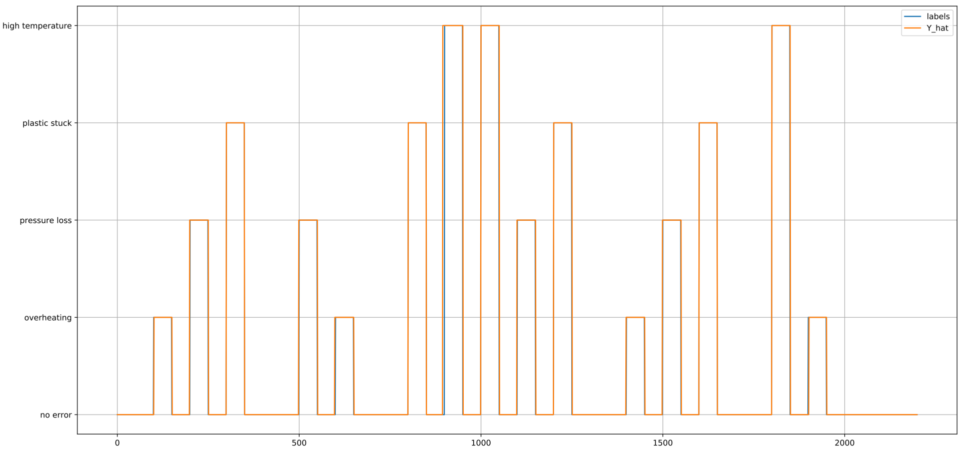

Figure 5.5: Errors classification

Again, the blue line is the correct value, while the orange one is the predicted one, named **"Y_hat"**. On the y axis there are the values of the output concerning

each situation. These two functions mostly overlap to each other, this means that the model learned to recognize quite well all the cases.
Here is also shown the loss:

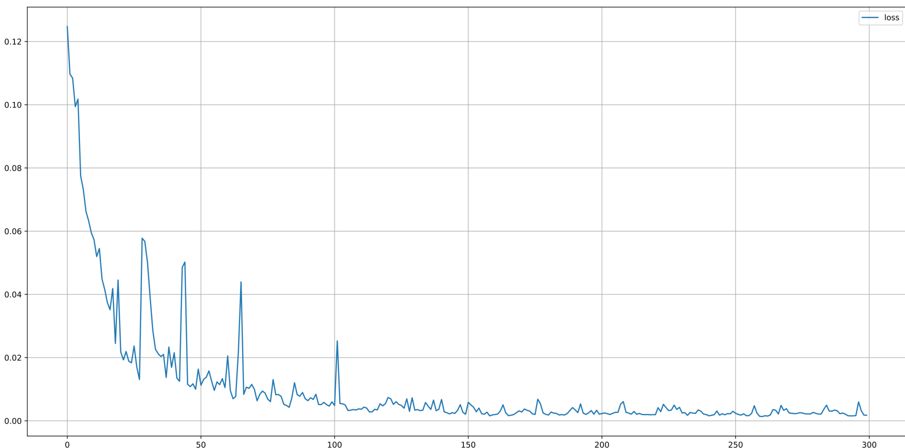

Figure 5.6: Loss

As it is possible to see the loss starts to be stationary after 150 epochs, this is why it has been chosen to train over 200 epochs, not more, because a higher value can be a waste of time and resources.
Next the model was saved inside a ".h5" file, this format allows to store also the weights inside the network, in this way the agent on which the machine learning is deployed does not have to perform expensive operations, like training a new model from zero.
Moreover, once loaded, the states of the LSTM neurons are reset and the model will keep learning. Indeed, when a prediction is made, it will be sent to the user interface, in order to have a confirmation, what is returned is the correct label for that situation. This label is used to fit again the model, this time with a batch size of 1.

5.5 System behaviour

In order to test the behaviour of the system, the use case was first simplified at 1 Quality check device and 1 injection molding machine. Over these, one Inspector agent and one Producer were deployed respectively. Then the system was resized to the original composition, 1 quality check device and 3 injection molding machine,

and the agents were deployed accordingly. In both cases it is highlighted the communication skills among them and how they help to solve the problem, together with the limits that they have.

5.5.1 1 Inspector - 1 Producer

In this scenario there are one Inspection and one Producer. They are both loaded with the previously simulated datasets.

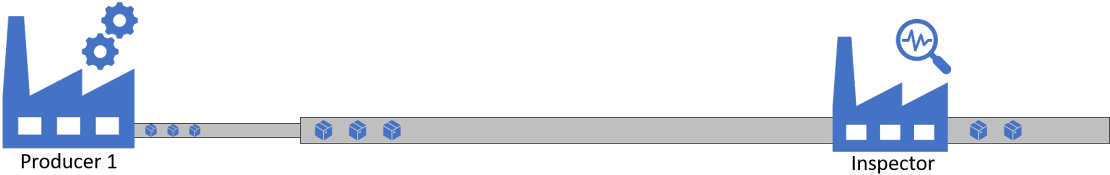

Figure 5.7: 1 Producer, 1 Inspector

Thanks to the agentService class, it is possible to start the behaviour of each agent with a unique script, since it is essential in this simulated case a correct timing between the two agents, while it is not always a problem inside a real production scenario.
In order to monitor both agents' behaviour, two command line have been opened. Inside them it is possible to see the states through which they go. Additionally, it is also shown the user interface that displays the values read by the devices.
The first thing done by each agent when deployed is the registration to the AMS, as explained inside the previous chapter.
Next both agents will start reading the data written, for the purpose of this simulation, inside a csv file. The information gathered from the environment are directly sent to the server through HTTP. Here the data is stored and made available for the user-interface.
In the first phase, the production process proceeds well and no messages between agents are exchanged.
The dashboard shows these readings:

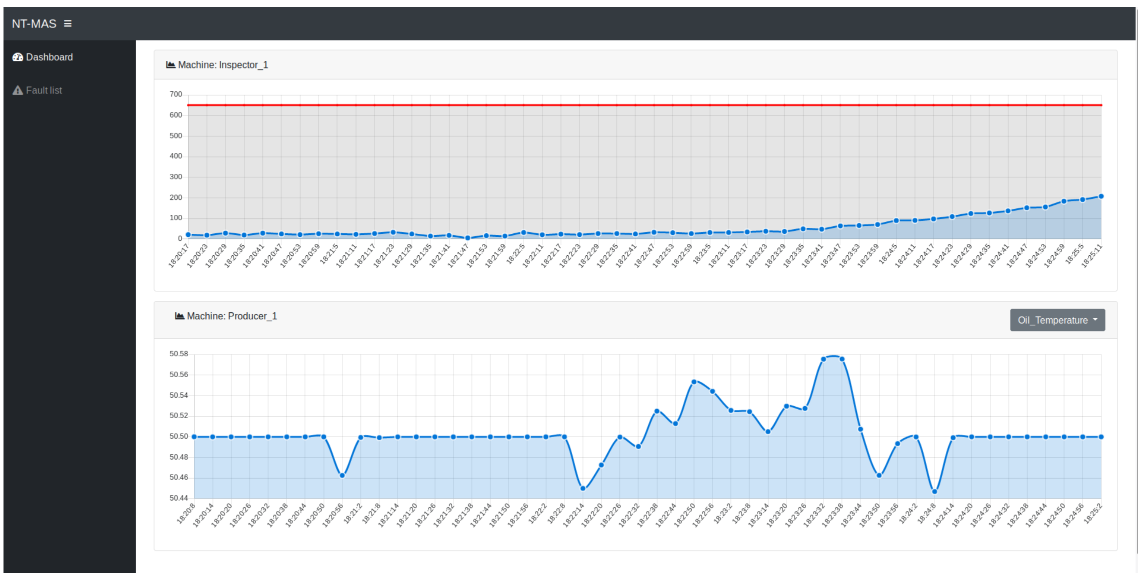

Figure 5.8: No error detected

As can be seen, inside the box of the "Producer_1" there is a dropdown menu, which allows the user to select which type of variable plot inside the graph.
After some cycle, the number of rejected pieces by the inspector starts to increase because of the simulated error.
As soon as the value monitored by the Inspection agent goes above the threshold, an alarm is sent to all the Producer inside the system, in this particular case only one.

```
nzt@nzt-N551JX: ~/PycharmProjects/NTMAS/deployment/quickstart/agent_inspector
nzt@nzt-N551JX: ~/PycharmProjects/NTMAS/deployment/quickstart/agent_inspector    nzt@nzt-N551JX: ~/PycharmProjects/NTMAS/src
agent_ins    | INFO      cherrypy.error                 error                         : [27/Nov/2019:02:07:51] ENGINE Bus STARTED
agent_ins    | INFO      __main__                       test_exchange_message         : [INS] ERROR found: Number of rejected pieces above threshold
agent_ins    | INFO      __main__                       test_exchange_message         : [INS] Sending alert to Producers
agent_ins    | INFO      capAgent                       send_informative_message      : [PRO] Sendind messsage to: agent_producer_1
agent_ins    | INFO      capCommon.capPublisher         connect                       : Connecting to amqp://guest:guest@172.28.0.20:5672/%2F
agent_ins    | INFO      pika.adapters.base_connection  _create_and_connect_to_socket : Pika version 0.13.1 connecting to 172.28.0.20:5672
agent_ins    | INFO      pika.channel                   close                         : Closing channel (0): 'Normal shutdown' on <Channel number=1 OPEN conn=<SelectConnection OPEN socket=('17
2.28.0.34', 33204)->('172.28.0.20', 5672) params=<ConnectionParameters host=172.28.0.20 port=5672 virtual_host=/ ssl=False>>>
agent_ins    | INFO      pika.connection                close                         : Closing connection (200): Normal shutdown
agent_ins    | INFO      pika.connection                close                         : Connection.close is waiting for 1 channels to close: <SelectConnection CLOSING socket=('172.28.0.34', 33
204)->('172.28.0.20', 5672) params=<ConnectionParameters host=172.28.0.20 port=5672 virtual_host=/ ssl=False>>
agent_ins    | INFO      pika.channel                   _on_closeok                   : Received <Channel.CloseOk> on <Channel number=1 CLOSING conn=<SelectConnection CLOSING socket=('172.28.0
.34', 33204)->('172.28.0.20', 5672) params=<ConnectionParameters host=172.28.0.20 port=5672 virtual_host=/ ssl=False>>>
agent_ins    | WARNING   capCommon.capPublisher         on_channel_closed             : Channel was closed: (0)
agent_ins    | INFO      pika.connection                _on_terminate                 : Disconnected from RabbitMQ at 172.28.0.20:5672 (200): Normal shutdown
```

Figure 5.9: Inspector sending alert

This is a screenshot of the command prompt of the inspection at the moment in which the alert is sent. The alarm message contains also a brief description of what happened, in this case "Number of rejected pieces above threshold".
Before sending the message, the inspection asks the number and the type of registered agents to the AMS, then it uses this information to publish the message inside the queue of each Producer.
Looking at the prompt of the Producer, instead, we can see different actions:

As always, the first action is the registration to AMS, while in the last lines we can see different operation:

```
nzt@nzt-N551JX: ~/PycharmProjects/NTMAS/deployment/quickstart/agent_producer
agent_pro   | INFO      2019-11-26 20:57:19 __main__                          setup                          77  : <Response [200]>
agent_pro   | INFO      2019-11-26 20:57:19 __main__                          setup                          78  : [RA][S] Registration to Agent Management Service OK. Starting ...
agent_pro   | WARNING   2019-11-26 20:57:19 capAgent                          __init__                       58  : SSL disabled
agent_pro   | INFO      2019-11-26 20:57:19 capAgent                          activate                       36  : [RA] Activating agent : agent_producer_1
agent_pro   | INFO      2019-11-26 20:57:19 capCommon.capConsumer             connect                        94  : Connecting to amqp://guest:guest@172.28.0.20:5672/%2F
agent_pro   | INFO      2019-11-26 20:57:19 capCommon.capConsumer             connect                        94  : Connecting to amqp://guest:guest@172.28.0.20:5672/%2F
agent_pro   | INFO      2019-11-26 20:57:19 capAgent                          listen                         92  : [A] Direct is False , Fanout is True
agent_pro   | INFO      2019-11-26 20:57:19 capAgent                          start_fanout_consuming         117 : [AS] Start FANOUT consuming ...
agent_pro   | INFO      2019-11-26 20:57:19 pika.adapters.base_connection     _create_and_connect_to_socket  237 : Pika version 0.13.1 connecting to 172.28.0.20:5672
agent_pro   | INFO      2019-11-26 20:57:19 capAgent                          dump_state                     579 :  --- Current State : IDLE
agent_pro   | INFO      2019-11-26 20:57:19 __main__                          setup                          93  : [RA][S] Producer agent activated.
agent_pro   | [26/Nov/2019:20:57:19] ENGINE Bus STARTING
agent_pro   | INFO      2019-11-26 20:57:19 cherrypy.error                    error                          222 : [26/Nov/2019:20:57:19] ENGINE Bus STARTING
agent_pro   | [26/Nov/2019:20:57:19] ENGINE Serving on http://0.0.0.0:5589
agent_pro   | INFO      2019-11-26 20:57:19 cherrypy.error                    error                          222 : [26/Nov/2019:20:57:19] ENGINE Serving on http://0.0.0.0:5589
agent_pro   | [26/Nov/2019:20:57:19] ENGINE Bus STARTED
agent_pro   | INFO      2019-11-26 20:57:19 cherrypy.error                    error                          222 : [26/Nov/2019:20:57:19] ENGINE Bus STARTED
agent_pro   | INFO      2019-11-26 20:57:21 capAgent                          on_message                     97  : [PA]-on message- received: [ERROR] from: agent_inspection_1, Type: Number of rejected
 pieces above threshold
agent_pro   | INFO      2019-11-26 20:57:21 capAgent                          evaluate_EV                    165 : [PA] Evaluating Error Vector
agent_pro   | INFO      2019-11-26 20:57:21 capAgent                          evaluate_EV                    166 : [PA] I am the faulty machine!!!
agent_pro   | INFO      2019-11-26 20:57:21 capAgent                          evaluate_action_ML             170 : [ML] Status: Overheating; Action proposed: increase cooling time
agent_pro   | INFO      2019-11-26 20:57:21 capAgent                          evaluate_action_ML             171 : Action sent to dashboard, waiting for a response...
```

Figure 5.10: Producer prompt

- The received alert from the inspection
- The calculation of the error vector
- The choice of the faulty machine, since in this case just one producer was deployed, no more operations were needed
- The prediction of the LSTM network, showing the status predicted and the corresponding action proposed
- The waiting for the user response, that can confirm or change the action

As soon as the Producer has sent the request, on the dashboard it will appear the proposed action, together with the other possibility:

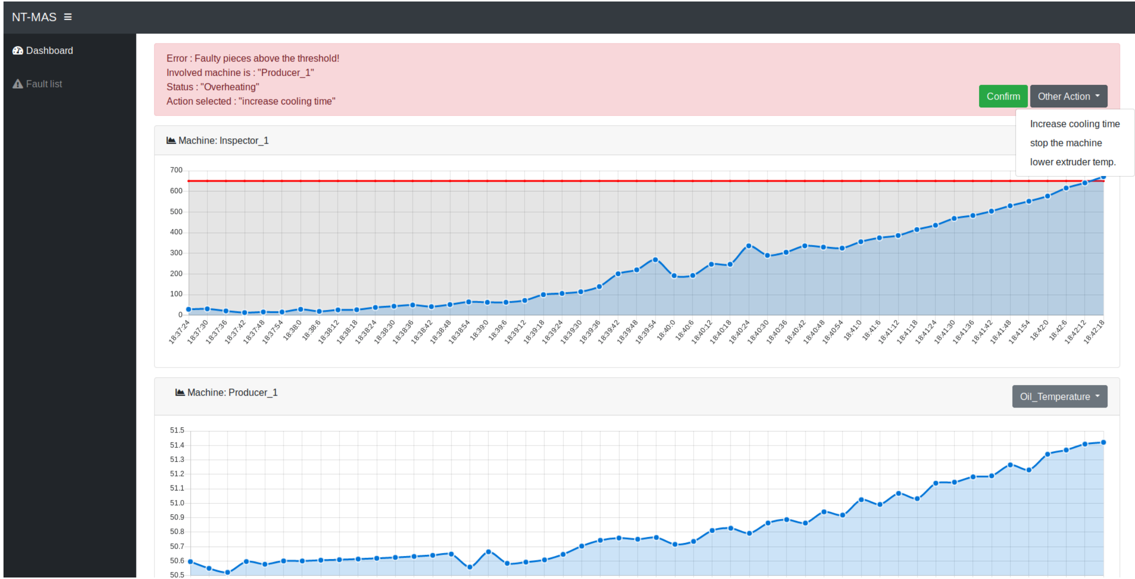

Figure 5.11: Dashboard interaction

Here, before deciding which action take, the supervisor can monitor all the variables to have a clear understanding of what it is happening inside the production process.

After the right action is selected a message will arrive to the Producer agent, which will update its LSTM model if the action is different from the expected one.
In this case, due to the size of the system, the number of messages exchanged only between the agents is equal to 1, if only producer and inspection are considered. Moreover, the computation of the error vector is useless, since no other producers are involved and the faulty machine can be only one.

5.5.2 1 Inspector - 3 Producers

In order to show the ability of the agents to identify the faulty machine, a new test was performed. In this scenario there are one Inspection and three Producers.

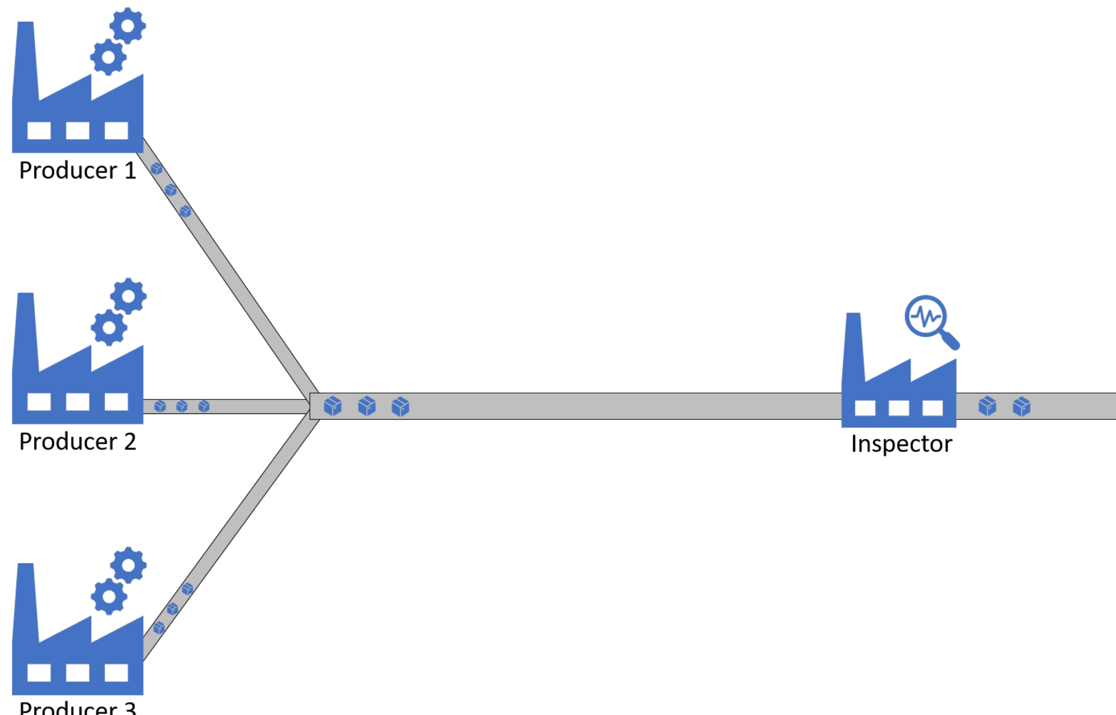

Figure 5.12: 3 Producers, 1 Inspector

While the Inspection agent is loaded with the same dataset of the previous tests, the Producers are loaded with different type of dataset. One of them, the "Producer_1" has the dataset containing the error simulated previously, the "Producer_2" instead has the original dataset cleaned and the "Producer_3" has the same dataset of the "Producer_2" but reversed.
When the number of rejected pieces increase the Inspector agent will send the alert, but this time not only to a single producer, but to all of them, as can be seen in

this console:

Figure 5.13: Inspector sending multiple alerts

When the alert arrives to a Producer, it will ask to the AMS the total number of Producer agent inside the system, by mean of an HTTP POST request. This information is exploited to decide how many error vectors send and receive, before identifying the faulty machine.
At this point each producer will generate its error vector and will send it to all the other ones inside the system. Here is an example by the "Producer_3":

Figure 5.14: Producer_3 prompt

Inside each agent there are two counters:

- N_PRODUCER_REGISTERED
- N_ERROR_VECTOR_RECEIVED

The first one is initialized with the response received by previous call to the AMS, while the second one is initialized to zero and increased by one each time a new error vector arrives. As soon as these two counters are equals, the highest error vector is evaluated and the corresponding producer appointed as faulty.
In this case the appointed agent is "Producer_1" as defined by the simulation. The

simulation than continues as in the previous test, the action is predicted by the machine learning method and then confirmed by the supervisor through the dashboard.

5.6 Results

In both cases the NTMAS correctly handled the situation and the system has adapted correctly to the introduction of more agents. In particular two kinds of collaboration ability arise:

- the first one between inspector and producer, for the understanding of a problem inside the production process.
- the second between multiple producers, for electing the faulty machine.

Moreover, the system proved to be capable of handling the interaction with the user and to gain advantage from it.

Chapter 6

Conclusions and Future Work

This thesis proposed, in the context of Industry 4.0, a semi-automatic multi agent intelligent for production processes. With the aim of build a MAS that incorporates an intelligent data analytic tool, capable of classify production anomalies such as machines errors or malfunctioning and to provide a countermeasure given a set of possible actions. This platform also provides a dashboard that allows user to monitor the production and to confirm the agent action before it takes place.
In order to evaluate the behaviour of the system, a real scenario, with partially simulated data was exploited. The system has proven to correctly classify the anomalies introduced in the scenario and to be scalable when more agents are deployed, but some limits are presents:

- It is not possible to add a new action without re-train the entire machine learning model.
- The system could not handle multiple alerts at the same time: if during the user's decision another alert rises, the previous one is discarded.
- If an agent stops working unexpectedly, the AMS is not able to recognize it.

Besides removing these limits, future research should aim to deploy the proposed system in a real production environment to test its effectiveness and to collect feedback from company employee.

Bibliography

[1] Stan Franklin and Art Graesser. "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents". In: *International Workshop on Agent Theories, Architectures, and Languages*. Springer. 1996, pp. 21–35.

[2] Agent Working Group. "Agent Technology, Green Paper". Version 0.91. In: (Mar. 2000). DOI: 10.13140/RG.2.2.22680.19206.

[3] Bjorn Hermans. "Intelligent Software Agents on the Internet: Chapters 6-7". In: *First Monday* 2.3 (1997).

[4] *JADE*. URL: https://jade.tilab.com/.

[5] *JIACv*. URL: http://www.jiac.de/agent-frameworks/jiac-v/.

[6] Ahmed Mohamed and Hosny Abbas. "Multi Agents System for Industrial Applications". In: (Feb. 2013). DOI: 10.13140/RG.2.2.22680.19206.

[7] *SARL*. URL: http://www.sarl.io/about/.

[8] *Understanding LSTM Networks*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[9] Michael Wooldridge and Nicholas R Jennings. "Agent theories, architectures, and languages: a survey". In: *International Workshop on Agent Theories, Architectures, and Languages*. Springer. 1994, pp. 1–39.