Faculty of Science, Technology and Communication

Analysis and Development of Autonomous Systems with the Help of Proactive Computing

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master in Information and Computer Sciences

Author:
Marlene Carina MÜLLER

Supervisor:
Prof. Dr. Denis ZAMPUNIERIS

Reviewer:
Prof. Dr. Pierre KELSEN

Advisor:
Dr. Jean BOTEV

August 2015

Declaration of Honor

Hereby I declare that the present academic work is my own work. All activities directly involved in it were carried out by myself.

Furthermore I assure that I have not used any resources other than those declared in the Bibliography section of this document. Any part that has been literally or conceptually adopted from printed, unprinted or Internet sources is cited accordingly with a precise indication of the source.

Also I assure that this academic work has not been submitted as a part of or as a different assessment. The submission includes three printed versions as well as an electronic version of the present work. I confirm that all versions are identical.

I am aware that a false declaration will have legal consequences.

Luxembourg, 20^{th} *of August 2015*

Marlene Carina Mï¿$\frac{1}{2}$ller

Acknowledgments

Let me begin with my special thanks to Professor Denis Zampunieris, who offered me the great opportunity to fulfil the work for my Master Thesis within his research and development team. He offered me a very interesting topic and supported me with constructive discussions and valuable support throughout the time of my internship.

Further I would like to thank Jean Botev for being part of the Jury and supporting my preparative work with objective comments.

I am also grateful for the technical support I received from Sandro Reis. He provided me with the framework and helped me to set up the necessary configurations.

Moreover I would like to express my appreciation to Remus Dobrican for all discussions about the design and development of autonomy. Especially his exaggerated ideas opened new perspectives for me.

All assistance from Gilles Neyens, either technical or within discussions about both our thesis' was greatly appreciated.

Altogether the whole team surrounding Professor Denis Zampunieris offered an enjoyable environment for my Master Thesis work and I am very grateful that I had the possibility to join the team for the second time during my studies.

Additionally I would like to thank Professor Pierre Kelsen, our student director, for accepting me for the Master Program and being part of my Jury.

Finally, I would like to express my appreciation towards my family and friends for their continued support and encouragement during my Masters Degree. They motivated me, believed in me and supported me throughout the past two years.

Abstract

The work in this Master Thesis contributes to the field of autonomous systems and is looking to explore the possibility of using proactive engines for building autonomous systems. Many approaches to the topic of autonomous systems exist, the contribution to the existing research is the usage of a proactive engine center piece.

The proactive engine developed in Prof. Zampunieris' research team at the University of Luxembourg is used to monitor and support systems. The proactive behaviour is used to add functionalities and properties to stand-alone systems.

The ability of adding properties to existing systems is pursued in this work. It is investigated if the possibility of using the proactive engine as control unit for autonomous systems is given. The goal is to add all properties to a system such that this system can be denoted as autonomous. This means the proactive engine needs to supervise and manage the system in a way that the required interaction with humans can be reduced to a minimum.

The proactive engine makes use of sensors to monitor the system and its environment. From the gathered data decisions can be taken and the system uses sensors to adapt to its environment and manage itself.

The adapted proactive engine that can serve to add autonomy to existing servers could in the future also serve as control unit for newly designed systems.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In this first chapter a presentation to the topics involved in the research and a definition of the purpose of the internship will be given. The goal was to design an autonomous system based on a proactive engine. In order to create a common basis of communication a short introduction of autonomous systems and the field of the proactive computing is given. Then the specific contribution to these fields will be defined.

1.1 Autonomous Systems

The topic of autonomous systems is a relatively new field in modern information technology. Research in this field treats the idea of systems that can deal with themselves without requiring the interaction of a human being except for setup purposes. The goal is to build systems that combine the actual function of the program and the control over the system such that human monitoring or intervention can be reduced to a minimum. The origin of this vision is the fact that the complexity of IT systems is increasing at a high speed.

The higher complexity of systems entails a higher cost for maintaining these systems. Realising backups or updates of software can be a tedious task with a high time investment depending on the size of the system. Another aspect that should not be left out of consideration is a changing environment. Systems are not just set up once in their environment but must steadily be adopted to the changes in their environment which can include a lot of configuration work for large systems.

Modern companies spend up to 50% of their total cost of ownership (TCO) to maintain their communication and computing resources. The maintenance includes backups, updates and the prevention of failures [17]. To reduce these high costs, the tendency of the evolution in the development of IT systems is towards more autonomy. Reducing the

maintenance work for IT-staff raises the possibility to focus on the design of new systems and the improvement of the companies' main tasks [24].

A requirement for autonomous systems is that safety and security is guaranteed. The system must be protected against malicious attacks over the connected networks. Big companies have IT-staff that is solely responsible for the security of the internal systems - autonomous computing will allow a reduction of workload.

A system that is denoted to be autonomous is a system that can function without human interaction. Such un-administrated systems are in charge of their own monitoring and controlling. The system is supposed to recognise changes in its environment and automatically adapt the internal process to any new surroundings. This idea contradicts the traditional way of designing software where the exact sequence of events is anticipated.

The design of an autonomous system focusses on the goal that the software is supposed to reach. This target is chosen during the development phase. The system itself searches for solutions and selects the actions to be performed in order to reach this objective. [15]

An example of an autonomous but predefined system is the regulation of temperature, that heats when it is too cold and cools the air down if it is too hot. This very simple system does not need human interaction like a traditional radiator. All possible states that the system can be in, have predefined solutions. Thus it is not hard for the system to find the correct solution to the goal of keeping a constant temperature in the room. With increasing complexity of the system the possibility of determining every reachable state fades away. For large IT-Systems it is impossible to find all potential situations during the design phase thus the approach of the radiator can not be adopted for real world autonomous software systems. The solution to the goal must be dynamically determined during the runtime by the system itself. The only components that will be defined in the design phase are the goal to be reached and a certain strategy as to how a solution is to be found.

In the field of autonomous systems the term "self-* properties" is used frequently to determine all the properties that an autonomous system offers. The dynamic adoption to a changing runtime environment is also known under the notion of self-configuration. Many organisations provide 24h a day and 7 days a week service. The possibility of taking the system down in order to change some conditions or configuration by human interaction is rarely given. Internal functions of the software should be changed in order to ensure a correct functioning of the system at any moment in time. In order to achieve this, autonomous systems must be environment- and self-aware [11].

The system must not only be able to observe its environment via sensors but also needs to be able to interact with its environment by using effectors [5]. Through this context-awareness the system can adapt itself during runtime and thus reach the predefined goals depending on its environment and the strategy.

Correct functioning of a system also includes the management and masking of possible internal hardware or software failures. Most of the current systems are configured in a way that they give an alarm or notify a human in case there is a malfunctioning. Autonomous systems however are characterised as self-healing, which means that the system must find a new way of reaching the defined overall goals by itself. Additionally the system will try to restore the malfunctioning part in order to be able to reuse it. This revolution would overcome the time-delay between the notifying alarm and the human reading the failure status, checking the program, resolving the malfunctioning and maybe also rebooting the system.

Optimisation of software systems is an important factor during the design phase. The performance of the service delivered by the system should be as high as possible, whilst energy consumption and produced costs must be kept as small as possible. Autonomous systems should be self-optimising and self-organising, this means the solution found by the system should not only reach the goal, but should also be optimal with respect to performance or costs. In order to adapt the selection of possible actions to the state of the system it is very important to self-monitor the workload and all system components.

Another very important property of an autonomous system is self-protection. The security of a software system must be included in the design phase. An autonomous system deals with a changing environment and must therefore also adapt its security and protection mechanisms to any new environment.

1.2 Proactive Computing

The origins of proactive computing go back to the year 2000 when Tennenhouse published his article [21], in which he analysed the human-computer interaction. He states, that in order to reach the computers per humans breakpoint, where networked computers outnumber the humans on the planet by a hundred or a thousand to one, we need to change from a human-centred perspective to a human-supervised perspective. His vision is that computers do not only react to commands coming from humans but can also work on their own for the user by having a certain goal.

This first definition of a proactive system is the foundation of the development of a system that is able to make decisions and take initiatives. The proactive engine developed within the research and development team of Prof. Dr. Zampunieris was designed to supervise the learning platform moodle in order to improve the e-learning experience. This engine was used to add proactivity to Learning Management Systems (LMS). The idea of this engine is to have access to the systems' state via their database using a database wrapper. It can easily be added to different kinds of systems.[4]

Internally the proactive engine is a Rule Running System. In general, every task to be performed is modelled in a Rule. Rules can be repetitive, in this case they have a monitoring

function and wait for certain kinds of events. Upon occurrence of such an event other Rules can be generated which are then responsible to take actions. The generated Rules are then managed by a component called QueueManager which executes the Rules in the queue one by one.

A very important property of proactive systems in this context is that a proactive engine can react to events or the absence of events without knowing the exact sequence of the events.

1.3 Purpose of the internship

The presented work concentrates on the possibility of building an autonomous system using the technology of the field of proactive computing. The goal is to develop a new kind of proactive engine such that it can monitor and guide an existing system. The proactive engine is a part of a system that is autonomous, it takes over a big part of the supervising, management and administration work for this system. Using the new engine, new properties are added to the existing system so that this can then be considered as autonomous.

The field of proactive computing can also play a big role during the design of a new system that is supposed to be autonomous. The control functions of the newly developed system can be gathered together and modelled in control Scenarios and Rules.

The development of a suitable proactive engine includes a detailed evaluation of the state of the art of autonomous systems. The analysis of the field of autonomous systems includes the definition of necessary and optional properties that a system must have in order to be considered autonomous. Furthermore, the existing approaches and ideas in the field are presented.

The work of this Master Thesis does not intend to improve the proactive engine, or to make the engine itself autonomous. The goal is to use the proactive engine to make a different system autonomous.

1.4 Structure

This document is structured as follows, Chapter 2 presents the work that has been conducted in the field of autonomous systems and the approaches to the topic that have been evaluated through different works. The potential and the properties of autonomous systems are analysed, but also the concerns about the new form of software are brought to mind.

In Chapter 3 an introduction to proactive computing is given. The internal structure of the proactive engine and the properties of the engine are explained in detail.

Chapter 4 elaborates the main contribution of this work. It explains how the properties of an autonomous system can be implemented by using the proactive engine and its Rules and Scenarios. The functioning of an autonomous system that is build on the basis of a proactive engine is presented step by step. At the end of the chapter an outlook on possible improvements is given.

A proof of concept is presented in Chapter 5. First the model of an existing system is shown which is then followed by the presentation of the proactive engine that makes this system autonomous. A precise presentation of the internal components of the engine is given along with implementation details.

The work is concluded in Chapter 6 where personal feedback on the internship is given. Further an outlook on future projects is also presented.

A glossary and an abbreviations list can be found at the end of the document, followed by the bibliography containing all resources used.

Chapter 2

Autonomous Systems

This chapter will give an overview of the state of the art in the field of autonomous computing. The idea and the motivation behind autonomy is presented and specific terminology is defined. Further, existing approaches to the topic of autonomous computing are presented and the properties of the given systems are analysed.

2.1 Introduction

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it", a quote of Mark Weiser from the year 1993 [23]. In the near future humans will get less and less aware of the usage and necessity of technology in their lives. Computerised objects are hidden in many devices and they will at some point communicate with each other without the interaction of a human being because they are networked and completely independent. In this context I would like to mention the Internet of Things (IoT), the term signifies the connection of computerised devices that are Wi-Fi enabled or use blue tooth to communicate with each other. These so called smart devices are used to make life more comfortable. The

upcoming trend are home-devices that are computerized in order to be controllable over the internet e.g. thermostats, windows or lights. [19, 3]

In many domains we use embedded software for important decisions. This has more than once led to disasters or near-disasters, so that the development of safer and more reliable software for these purposes is unavoidable [18]. Currently many organisations spend a big part of their TCO to avoid failures, whose consequences would be catastrophic. The development of self-managed and cost-efficient computer systems is the main point of interest in the field of autonomous systems at this moment.

The term of Autonomous Computing (AC) was coined by the company IBM in 2001. Their vision of AC is a system that manages itself in order to reach a goal that was defined during the design of the system. The inspiration for these systems is the human body where generated cells can take over any task, and grow into their environment and adopt new functionalities [9]. IBM states the following aspects that define these new species of systems. An autonomous system is supposed to automatically configure and adjust all components, whilst also steadily searching for the possibility of optimisation for the overall system. On the other hand the system should be able to detect internal software issues, repair these and protect itself against malicious behaviours. These properties are described in a more detailed way in Section 2.4.

Autonomous systems can only achieve the above stated properties by monitoring all tasks that are conducted internally. The system must be aware of its own state and of its environment. Systems must adapt to changes in their environment and interact with them in order to be able to exist and work without the intervention of human actions.

2.2 Potential of autonomous systems

The main motivation for autonomous systems is the reduction of high maintenance costs that computer systems have. IT-staff in companies is charged to do regular backups and updates. Furthermore failure prevention and resilience management is another task of system administrators. With growing size of IT-systems this task becomes harder and as humans are not impeccable, errors made by system administrators can lead to malfunctioning of the software. According to [16] 40% of all computer problems are attributable to human faults, which in turn is another motivation to improve the management of software systems.

Another important aspect are failing systems. In the current situation a system failure may be detected by a user of the system or an alarm is triggered because the system is in an unstable or insecure state. Now a human system administrator must identify the specific issue in the system and find a solution to resolve the given problem. Depending on the importance of the system this means a system administrator must either be available on site permanently twenty-four-seven or for less important systems the time delay between

the detection and the resolution might end up in being long during unfavourable times such as weekends.

However the issue stated above can be resolved by developing self-healing systems. A self-healing system can identify the issue as it arises and is able to automatically and instantly search for a solution to solve the problem. This would mean that the system failure can instantly be resolved so that less users are affected by the issue and the time delay is reduced to a minimum [10].

The authors of [16] propose a 6 step self-healing system which includes real time monitoring of error events. The system can thus directly react to the failure. The intervention of several agents is the key point in this project, which reduces the usage of resources for error handling and the size of log files. The error is analysed and a diagnosis is drawn from it. The decision of the solution is either taken directly or the web server is searched for a specific solution for the problem encountered.

In [17] the authors state that a cost efficient and effective protection against security threats can only be achieved with a certain degree of automation. The software thus needs to defend itself against viruses or other malicious attacks. Autonomy could also prevent accidental releases of information or other security threats.

It is impossible to determine every single state that the system can be in, which means that the development of a system with a predefined solution to every outcome is not feasible. The complexity of systems nowadays is too high. Hence why the system needs a certain degree of autonomy so that it can take decisions and react to changes in a desirable way.

The authors of [18] compare an autonomous system to the body of a human being or an animal. The body has sensors and effectors in order to interact with its environment. Figure 2.1 shows the visualisation. The property of being self-managed can be split into the four objectives shown in the figure. Taking the body as an example they can be explained in a simple way. Self-protection against attacks or illness is a natural behaviour of a body. The immune system is responsible for the protection against viruses and in case of an attack the body produces adrenalin so that one can defend oneself. And in case of an injury, self-healing means the wound cures itself if it is not too severe. Self-optimisation means that the body is aware of its performance and tries to enhance it. Self-configuration is the adaptation to new circumstances.

In order to achieve these objectives the body needs the attributes displayed in the figure. Again these can be explained in a straightforward way, by taking the human or animal body as an example. It is aware of itself and of its environment and monitors what it is doing. There is a permanent adjustment of actions and behaviour to the changing environment and given circumstances.

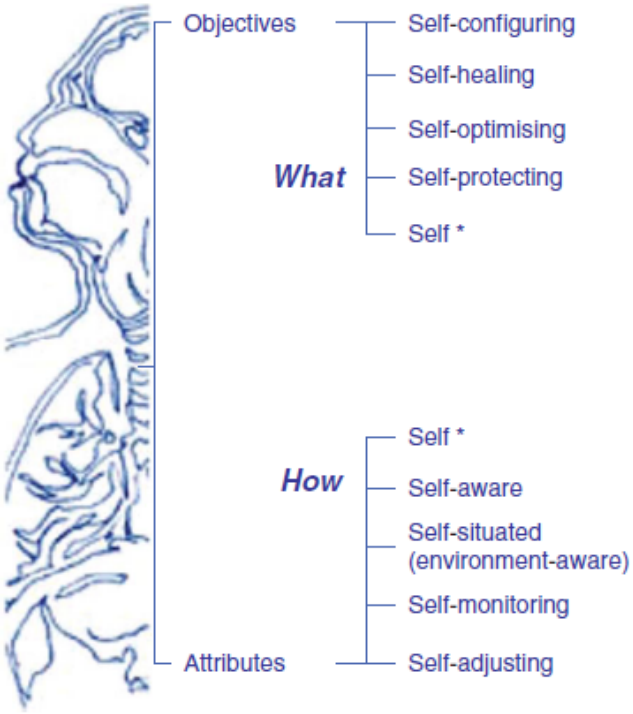

Figure 2.1: The functioning of the autonomous systems inspired by the human body [18]

2.2.1 Level of Autonomy

The development of autonomous systems is a trend that has slowly arisen out of the traditional development allowing to analyse the so called level of autonomy of existing and new systems. This level defines the maximum human interaction required and the minimum part that a system needs to deal with itself.

For some systems it is also possible to add new components subsequently without changing the system in order to make it more autonomous. The authors of [24] defined the levels shown in Table 2.1.

In the table it can be read that a trend is clearly given, more and more tasks are transferred to the responsibility of the system and the systems administrator is discharged of many tasks so that it becomes possible for one administrator to be in charge of an increasing number of systems without having an increased workload.

The level `Basic` represents a fully reactive system that can only execute the tasks that are assigned to it. A system that is denoted as `Managed` gives feedback to the system administrator and makes the guiding of the system already easier. Systems that are

Level	Administrator Role	System Role
Basic	Set Up, Monitor, Enhance system	Execute Tasks
Managed	Analyse information, Make decisions, Take actions	Collect information in a consolidated view
Predictive	Make decisions, Take actions	Recognize patterns, Provide advice on action
Adaptive	Observe, Ensure following policies	Take the right actions
Autonomic	Monitor business processes, Alter objectives	Operation is governed by business policies

Table 2.1: Level of Autonomy of a System (Table from [24])

`Predictive` support the user further by analysing the previous actions. Today most systems for the great public are able to do this. Further `Adaptive` systems, however, can decide about actions and find a right solution. The support by the administrator is reduced to observation and ensuring the policies of the company are respected. Finally a fully `Autonomic` system is basically able to work without administrators, meaning the system decides about actions in order to reach the objectives given, and the only interaction that could be required here would be to change these objectives. [24]

2.2.2 Doubts

Another aspect arising with the development of user-less systems is security. It has to be thought of the consequences when such systems are out of control. A system that can decide on its own what solution to choose or what to do next can harm in a wide dimension. Examples for such behaviour reach from smaller damage to a high extent. For example one could have systems that publish secret information about business tasks to the internet, which could harm an organisation. Furthermore, a system could just shut down completely and refuse the service that an organisation offers to their clients. A higher damage would be the publication of sensitive information about people or patients. But the most severe consequences could, for example, occur from machines in hospitals that could decide to switch off in which consequence patients could die.

A real example of autonomous systems making a fault of a human worse is the stock market collapse in 2010. Computer based trading dragged the prices down over night. This happening shows clearly that with the involvement of self-deciding systems a human error could have a large magnitude of chain reactions [7].

The development of autonomous systems is pushed by the military so that machines or drones could be used in a war without getting humans in a dangerous environment. The

magnitude of wrong decisions during wars can only be imagined. Currently the American law requires at least one human being in the decision making process of a drone firing at a person [15].

The authors of [15] are concerned about the development of autonomous systems driven by the military and economics section. These systems have a drive to a high self-preservation. Furthermore, such systems often have an urge to gather as much resources as possible to guarantee an optimal functioning. These resources might often stay unused if they would be liberated, other systems in the same environment could make use of them. It can be observed, that with this "anti-social" behaviour of some systems the other systems in the same environment do not have the possibility to work in high performance. Another concern is self-replication of systems which could also lead to dangerous behaviour.

Often the sentence "the systems can just be shut down if they behave incorrectly" is lanced in this context. But systems that implement the self-protection property might find a way to bypass the "switch off" as they are able to find solutions in order to accomplish their goals even if the circumstances are not perfect. And on the other hand, sometimes the start of an undesired action has already negative impact even if it is stopped directly.

An exaggerated example of systems that are out of control is given in modern films were robots that where assigned a simple but helpful task suddenly turn into monster machines that are determined to dominate the world and kill the entire population.

2.3 Existing approaches to the topic

A certain amount of autonomy of a system has been necessary in order to bring computers and other systems into peoples houses. It must be possible for a person who is not a computer science specialist to use the computer. Examples for self-configuration in the every day life are the connection of a USB stick to the computer or the connection from a computer to a remembered Wi-Fi network.

For more complex systems there are two approaches to make them more autonomous [12]. The top-down approach attempts to add autonomy to an existing system. This idea requires the system to support runtime monitoring and needs a component to modify parameters during runtime. Mostly this is done by adding control loops to the system. The original design of the system is not changed, the system is only extended by an additional component.

The other possibility is the bottom-up approach which gives the development of systems a new perspective by including the self management property from the start. These systems are usually decentralised, meaning the development and the adaptation during runtime follow the same goals. These systems seem to rebuild themselves during runtime in order to adapt themselves so that their goals are reached.

2.3.1 Events

One of the approaches to the world of autonomous systems is by grouping types of events together meaning that during the design phase the developer can define which of these events are relevant for the system and how the system should react to them. First of all for this perception it must be clear that everything can be modelled as an event, in this case even the absence of a happening is an event. Proactive systems are one of the types of systems that use Rule-based engines that analyse events and react in a defined way. The authors of [5] distinguish three types of events: The foreseen events, to which the system reacts, the expected events, events that are planned, which the system waits for and the unforeseen events, which are not planned [2].

During the design of a system a goal needs to be defined and depending on the events that occur on the way to that goal during the lifetime of the event, the system should build a solution. Therefore directions must be given to the system, for example by predefined Rules. These Rules should provide the system with the possibility to react to something unplanned in a correct way.

We can use the comparison of a human in a new environment, for example during a trip to a foreign country. The person will inform him/herself before the journey about Rules and good behaviour in this country in order to be able to react in a reasonable way to any encounters during his/her stay in the foreign country. Translating this to a computer system means that the developer needs to predefine enough Rules so that the system is able to make decisions depending on the situation or the environment. These decisions can be named correct, if the following process brings the system closer to the goal.

In current proactive systems the sequence and timing of events is unknown but the exact solution to each event is given by the designer, this means that such systems can only react to foreseen events. The possibility of reacting to the absence of an event is the first step in the right direction, as in this case the system can see that the progress is interrupted and something needs to be done by self-initiative. The future behaviour of a system can be influenced without knowing the exact behaviour and process while implementing the system.

Events that are unknown to the designer of the project are out of the scope of this project. This topic might be treated using techniques of Artificial Intelligence. This decision can be supported by an example, in section 2.2, where the system is compared to a human body. So if one imagines an unknown event for a human, we can see that also humans do not always react in the correct way in such situations, for example during a bomb scare or if a tiger was to suddenly appear in the middle of a street. A person, who is not educated as to how to react in these kind of situations will most probably be in a state of panic. The reaction of the human body to unknown events can not be foreseen. Thus the reaction of systems to unknown events is not included in this study.

2.3.2 Monitoring and self-awareness

A crucial fragment of autonomous systems is the self-awareness. Only systems that are aware of their internal state and of their position in their environment can take logical and good decisions. In Figure 2.2 the internal control loops that are necessary to provide self- and environment-awareness are shown. These systems need a component that is responsible for monitoring the systems' actions and processes. The authors of [11] show in their paper that monitoring and self-awareness is the basis of other self-* properties [12].

The model presented in [11] uses low-level monitoring from where the information is reported to a higher level instance for analysis and decision taking. Currently monitoring is done with hardware components such as sensors but software can also be included in this part of autonomous systems. Sensors are potential failure points and must be included in the system at design time, even though software could adapt the analysis of different parts of the system to the current situation in the environment. The monitoring function presented in [11] uses a function with event IDs which can be used to group events dynamically. This is done by masking parts of these ID's in order to make analysis easier and implement different thresholds.

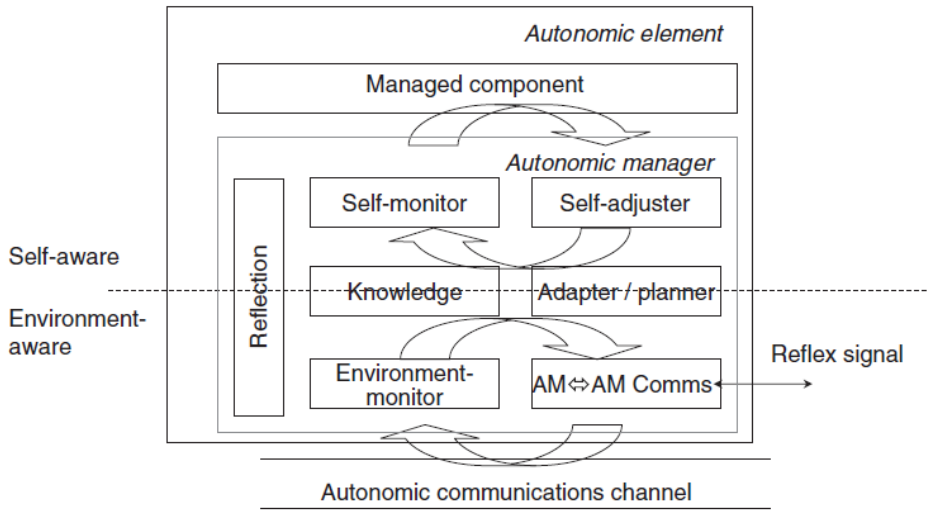

Figure 2.2: Reflection layers of a system that is self-aware [18]

2.4 Properties of an autonomous system

The book "Autonomic Computing - Principles, Design and Implementation" [12] is used as fundamental reference for properties of autonomous systems in this thesis. The definition given in the book is completed using other resources as indicated. The authors of the book identify the key features that are presented in the following. These features support

the idea that the designer of an autonomous system can specify the high-level objectives of the software without taking into consideration the technical details on how to achieve the goals.

2.4.1 Self-Configuration

Reconfiguration of internal values is important in order to react to changes in the environment of the system. These events can be expected or unpredicted, either way the system will identify the happening and search for the best solution within the system. In a system composed of several units, this property is responsible of including new elements and dealing with the loss of some components.

The fundamental idea of self-configuring systems is that during the design the system is told what is desired rather than how to achieve it. The configuration property is the enabler for self-healing, self-optimisation and self-protecting presented in the following.

2.4.2 Self-Healing

The recovery from internal failures is essential for systems that must provide continuous service without human interaction. This means that the system must detect failures or predict problems and can react to this finding instantly. The robustness of an autonomous system must be of a high level.

Via self-awareness the system is able to identify potential issues or components that are vulnerable for failures. If such vulnerabilities are identified, the software must find a solution in order to recover the state of the system. This solution should avoid the recurrence of the same problem, which means the cause of the problem should be identified and eliminated. Further it is important to make sure that the solution to one issue does not cause other issues in the system's functionality.

The authors of [12] include in this property the finding of the root cause of the issue and not only the treatment of the possible consequences. In [16] an example using a control loop including monitoring, filtering, translation, diagnosis, decision and feedback is given for the self-healing property.

2.4.3 Self-Optimisation

The system constantly searches for solutions that could make the operations as effective as possible. Improving the performance while reducing costs and keeping a high quality of service is the goal of self-optimisation within an autonomous system.

The different concepts of the optimisation might be conflicting. For example the system might save energy by putting some resources to sleep and only working with the smallest amount of resources required but on the other hand the performance might be higher if more resources were used which also reduces workload. Although the optimisation within the components of the system does not automatically optimise the whole system.

Therefore the perfect solution must be established throughout constant analysis of the internal state and the executions of the software. The optimisation of a system is a process over time. By analysing, the system can learn and find better solutions. This again can be compared to the human body which builds up endurance when exercising every day.

2.4.4 Self-Protection

In this context protection includes more than the security aspect. The system needs to protect itself against malicious attacks whilst also being able to handle accidental user errors. Further self-protection also includes the aspect of failed self-healing where errors persist. Protection in this case means that the failure can be encapsulated so that the system's functionalities are not affected.

In addition, self-protection can also include pre-emptive checks in the system upon intrusion alert, in order to detect attacks or failures as soon as possible. It can include the shielding of some components of the system from an insecure environment, for example in an open network. Comparing to a human immune system, self-protection can be seen as a digital immune-system, in a perfect implementation it does not require user intervention or awareness [12].

Depending on the standard of a data centre many protection functions are already implemented. In [8] the tier standards of modern data centres are explained shortly.

2.4.5 Further Properties

The authors of [18] define properties for self-managed system as visualised in Figure 2.3. The system is supposed to take a proactive initiative to improve the system and be effective in terms of meeting the expectations and requirements of the user. Robustness defines the ability of handling system errors and ensuring recovery. Flexible and easy to use systems should be adaptable to the users needs. And as last cautiousness in the handling of risks and transparency behind systems actions are desirable properties.

In the book "Autonomous Computing" [12] the main properties `self-configuration`, `self-healing`, `self-optimisaition` and `self-protection` are gathered together as `self-chop` properties. All further properties that are necessary to achieve lower degreed

Figure 2.3: Visualisation of the properties defined in [18]

of autonomy are denoted as extended self-* properties, where the * can be replaced by many properties. The most important examples are given below:

- `self-adapting`: The system is able to adapt itself depending on the changes in its environment.
- `self-adjusting`: The system is able to modify its internal functioning.
- `self-critical`: The system can analyse if it is currently meeting the predefined goals or not.
- `self-diagnosis`: The system can identify existing problems and anticipate potential issues.
- `self-organised`: The system assembles independent elements in a decentralised manner in order to constitute a functioning system.
- `self-recovery`: The system can recover from partial or general failures.
- `self-stabilising`: The system is able to return to a legitimate stable state after a finite number of execution steps starting from any arbitrary state.

Further self-* properties are listed in the book [12].

Chapter 3

Proactive Computing

This chapter is dedicated to the introduction into the field of proactive computing. The proactive engine presented in the following is used as framework and as add on for existing systems to become autonomous. All meaningful details of the functioning of the engine will be explained so that a common basis is built for the analysis and design of an autonomous system.

3.1 Introduction

The notion of proactive computing incarnates the idea of human supervised computing. Tennenhouse defines the human-machine breakpoint in his article about proactive computing [21]. He defines it as "the point at which the number of networked interactive computers will surpass the number of people on the planet". The realisation that human centralised computing has its limits brought IT researchers to the development of more independent new approaches.

Human supervised systems are supposed to work without the interaction of a human. They can react to events that happen in their environment. Proactive systems are context aware and by monitoring their environment or the system that they are responsible for they can react to the absence of events. The sequence of the events happening does not need to be known in advance.

3.2 The existing engine

The original proactive engine was developed to add proactivity to Learning Management Systems (LMS). The idea of this engine is to have access to the systems' state via its

database, using a database wrapper. Therefore it can easily be added to different kinds of systems.[4]

The heart of the engine is a Rule Running System. Generally speaking, every task that has to be done is modelled into a Rule. Rules are enqueued into two different queues and the QueueManager executes them one by one. The following vocabulary explanations will describe the different components of the engine.

A **Rule** is a Java class that has a specific form. Rules can have fields and must have 5 methods (dataAcquisiton, activationGuards, conditions, actions, ruleGeneration). These methods have specific functionalities. The `dataAquisition` is used to read the required data from the state of the supervised system in the database using a database wrapper or interface, the `activationGuards`-method performs a check whether the Rule will be activated or not. If the activation is granted, the `conditions` perform further checks using data from the database. In the `actions`-method is the code that should be executed and the `ruleGeneration`-method clones the Rule itself or creates different Rules that will be added to the `nextQueue`. Consequently the `actions`-method is only performed if the `activationGuards` and the `conditions`-methods resolve to true. The `ruleGeneration`-method is only performed when the Rule was activated, but is not affected by whether the `conditions` are met or not. [13]

A **Meta-Scenario** is associated to a specific job of the engine. It stays in the engine as long as the job needs to be done. Often Meta-Scenarios are waiting for changes and check the database at every activation. Upon the expected change, other Rules are generated in order to react to the change. Its structure is like the one of a Rule. As long as the Meta-Scenario stays in the engine it has to clone itself in the `ruleGeneration`-method. [13]

The **currentQueue** and the **nextQueue** are two Java queues that hold Rules and Meta-Scenarios. The `currentQueue` holds the Rules that are executed in the running iteration. When a Rule generates other Rules during its execution these are added to the `nextQueue`. [13]

An **Iteration** is one cycle in the work flow of an engine. In one iteration a certain number of Rules is handled. The number is defined by a parameter in the database. There are some parameters in the database that limit the maximum time for one iteration, the maximum number of Rules per iteration and the minimum time between two iterations. [13]

The **QueueManager** is the control unit of the engine managing the two queues. Every iteration consists of three parts. In the beginning the `currentQueue` is saved into the database for backup reasons. Then the Rules of the `currentQueue` are one by one dequeued and executed using the five methods. By executing Rules, the generated Rules are added in the `nextQueue`. At the end the execution time and the number of Rules are saved in the database for statistical reasons and the items of the nextQueue are added to the `currentQueue` whereas the `nextQueue` is emptied.
Due to the parameters that limit the time for the iteration and the maximum number of

Rules mentioned above it is possible that the iteration might need to be ended before the `currentQueue` is empty. In this case the Rules of the `nextQueue` are added at the end of the `currentQueue` and the new iteration starts with the Rules that where left in the previous one. [13]

There are two possibilities for the termination of the engine. Either if there are no Rules left in the queues or if there is a stop signal in the database. In this second case the Rules in the queue are saved in the database and the iteration of the engine is stopped. The engine can easily be restarted with the same Rules.

The engine does not only have access to the database of the system it is monitoring, but also has an own database schema. The project is written in Java, uses XML-files and Hibernate to map the classes to the database. This database is multifunctional. It stores all necessary information for the engine, such as parameters and signals. Furthermore, it is used to save messages or other data that is used by the Rules and it serves as backup for a quick resume after a restart or in case the engine crashes.

The engine is able to communicate with other engines. This step was realised within two Bachelor Thesis' [13, 14] and enlarges the possibilities of application of the engine. Thus two systems that depend on each other's work or progress, can be monitored together and information can be exchanged via the engines, which is a new opportunity to make systems less dependable of humans.

In recent work the engine was adapted in a way that it can also run on mobile devices. With this addition new opportunities and use cases are open in the field of proactive computing. The involvement of mobile devices enables real time reactions and more flexible monitoring. [22]

3.3 Properties

The main properties of the proactive engine that are a fundamental prerequisite to build autonomous systems are presented in the following. The choice of proactive engines as a tool to construct an autonomous system is further evaluated in Chapter 4.

A system is proactive if it can react to expected events without the interaction of an user. The system is context aware and can thus monitor what is happening. The iterative construction of the proactive engine makes it possible to wait for several events in parallel. Hence the sequence of the events is unimportant and does not need to be known in advance. Further it is possible to identify the absence of an event and the system can then react for example by giving alerts or notifications.

The operational area of proactive engines is the monitoring of programs which interact with users. Their task is to help the user by supporting and guiding him. Which kind of analysis is done or what part of the connected system is supervised can be declared within

the Rules and Scenarios so that the proactive engine is, due to its flexibility, applicable to any system.

In the context of supporting an user the following example can be given. The first proactive engine was developed in order to monitor a the e-learning platform of the University of Luxembourg. The users, students that are using the platform, are supervised by the engine and their behaviour is analysed. In the case of a student forgetting about an assignment the engine notifies the concerned person. In addition to the above, all students are contacted if new assignments or course material is provided by the tutor. The proactive engine thus enhances the learning experience and adds properties to the existing system.

Since the Rules and Scenarios in the proactive engine are executed in each iteration, it is possible to have a close to real-time monitoring of a system's state. The state of a system can be determined using the internal database of the system. The proactive engine that supervises an existing platform or software has access to this database and can thus track all conducted steps.

The engine is aware of what is happening internally within the software system that it is responsible for. Since it has an overview of the complete system, it can be denoted as self-aware. Decisions for actions can be taken out of several reasons from different origins.

Due to the connection between two proactive engines it is possible to make decisions not only dependant on one software system but from several where a common decision can be taken and conducted on all systems using the proactive engines. This connection can even be established between systems that are not programmed in the same programming language. The personalisation of the proactive engine towards a system is done using the Rules and Scenarios and the engines among themselves can send commands to one another.

Chapter 4

Building Autonomous Systems with Proactive Computing

This chapter presents how a proactive system can be used as basis to add autonomy to an existing system. It is shown that the proactive engine can take over tasks so that the existing system receives the missing properties in order to be autonomous. The goal is not to improve the proactive engine but to use it as central control unit for an autonomous system.

4.1 Introduction

Building an autonomous system or making an existing system autonomous means that a control unit must take over the work of a system administrator or IT-staff member. This work includes maintenance such as updates, backups and failure recovery but also security and protection against possible attacks. The proactive engine, that constitutes this central unit is responsible for controlling, managing and protecting the complete system.

In this context it is very important that the system is self-aware, which means that the proactive engine that is added to or built into the system is able to know exactly in which state the system is and is able to monitor all processes that take place within the system.

Furthermore, the central control unit is responsible for keeping the system in a correct state in order to be able to work. This entails that an overall goal has been defined where

the system will be guided to. Necessary actions for the work-flow must be identified with respect to several properties and constraints, hence the central unit must have the ability to interact with every single component of the system or manage the system via an entry point or interface.

4.2 Using Proactivity

In the given approach we chose to use a proactive engine as basis for the central control unit, entitled "the Brain" in the following. By using a proactive engine as Brain it does not make a difference whether we use a top-down method where the system is designed to be autonomous, or a bottom-up method where the Brain is added to an existing system in order to make it autonomous. The proactive engine presented in Chapter 3 is very flexible and can thus be adapted to any kind of system.

The proactive engine needs access to the database of the concerned system in order to read its state. The Rules and Scenarios in the engine can be developed for the needs of this system. If the system by itself does not yet implement any properties of autonomous systems, the Brain can take over the complete monitoring and controlling. However, if the existing system, to which the engine is added to ensure complete autonomy, is already implementing a part of the necessary properties, the engine must interact with other control units in the system. These constraints play a part in the logic that is implemented within the Rules and Scenarios.

The list of properties of an autonomous system varies depending on the used resources. The most important properties are listed in Section 2.4. The system is supposed to take action on its own and find the best way to achieve predefined goals. The proactive engine implements a logic which is adaptable to a wide range of systems.

The property of the proactive engine of monitoring and reacting to events and the absence of events can support the system in the issue of self- and environment awareness. Only a system that is conscious of changes within its internal state and its environment has the possibility to take decisions on its own. The actions that are performed in the system should serve the achievements of goals that are predefined during the design of the system.

4.3 Control Loop

The control loop is a procedure that is implemented within the proactive engine in order to make the controlled system autonomous. The loop presented in Figure 4.1 is iterative and all stages of the process are implemented as independent tasks. The Brain guarantees small reaction times to any kind of event taking place in the system due to the iterative functioning.

The first step shown in the control loop is the collection of data, which means the monitoring of the system and its processes. This can be done by reading from the database of the existing system or using sensors to connect to the environment. The state of the system and the environment must be determined on a regular basis in order to implement the properties "self-awareness" and "context-awareness".

In a next step the collected data is analysed. There are several properties that influence the analysis. Depending on the implemented properties, models, criteria of the system and constraints in the environment, a local recommendation is elaborated. If the state of the system corresponds to the properties that the system is supposed to hold, no change is required, otherwise an action that could solve the local issue is evaluated and proposed.

The collection of these recommendations is the input of the decision step in the control loop. Depending on strategies and the goals of the system a list of actions is created. The decision stage is further explained in Section 4.6.

The final step of the loop is the execution of the actions. With this step the system adapts its state to the given situation. The actions are either directly executed within the system or via an interface that gives the Brain the possibility to communicate with the system. It can also include the use of effectors that influence or communicate with the environment of the system or other external components.

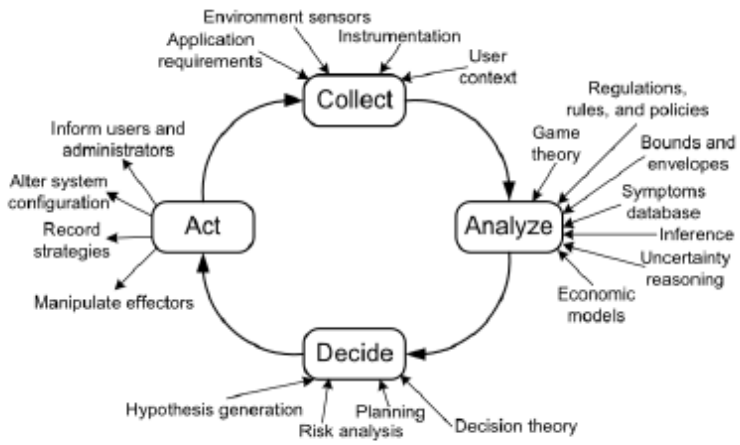

Figure 4.1: Autonomous Control loop presented in [5] (Figure from [6])

4.4 Partitioning

The approach of using a proactive engine as basis for the Brain opens a very flexible way of building autonomous systems. The main idea of this method is that the controlling of the system is partitioned into smaller tasks so that the different parts are easily manageable.

The Scenarios explained in the following section are each responsible for a small task. The design of an autonomous system is made less complex when using a proactive engine. The properties of the system do not have to be seen as a complete set but can be treated one by one. Each property is evaluated as if it would be the only one being implemented. The overall decision can then be taken using the results from the local analysis. This logic is again implemented in a separate place.

The partitioning, the independently working property Rules and the decision Scenarios make the design and realisation of autonomous systems a straightforward task. The complex idea of a system taking decisions on its own and adapting to every condition is taken apart into small non-complex tasks.

Since the proactive engine is an iterating Rule Running System, and the stages of the control loop are implemented within the Scenarios and Rules presented in the following section, the different steps of the system can be performed in parallel.

4.5 Scenarios and Rules

In this section the usage of Scenarios and Rules within the proactive engine are investigated in detail. As explained above the desired properties of the autonomous system to be built are analysed independently. Each property may include several actions and reactions to events within the system. For every monitoring or analysis task one Scenario is designed and each action that can be taken within the system is formulated in a Rule.

To clarify the previous statement the following example is given. When it comes to optimisation in a system, we can see that there are several possibilities to improve the functioning of the system. One could be boosting the performance by activating more resources if needed. The belonging Scenario would then monitor the workload of the system and decide whether it is necessary to use more resources. Another aspect of optimisation could be to reduce the energy consumption. In this case the number of used resources is reduced to a minimum. A different Scenario is responsible for the necessary monitoring and analysis. Both Scenarios belong to the property `self-optimisation`. For each property the possible Scenarios will be defined.

All Scenarios elaborate a best decision for their special case. These recommendations can be contradicting, seeing the previously given example of optimisation. On the other hand, however, two independent Scenarios can conclude the same decision from different

analysis, then the recommendation is given twice which would be unnecessary. Hence why these local decisions must be analysed and cleaned.

4.6 Decision of actions

The list of local recommendations is the input for a set of special proactive Scenarios implementing the third step of the control loop described above. The desired output is a ready to execute list of actions for the system.

The first decision Scenario retrieves the list in an ordered manner from the database and splits it into subsets that are distributed to the other Scenarios. There are several Scenarios that treat sets of logically dependent recommendations.

In order to transform this sub-list of recommendations to the list of actions, all double added decisions are deleted from the list first. Following this, the contradicting recommendation can be eliminated using priorities. That means some properties might be more important for the system than others meaning their recommendations will be stronger than contradicting ones from less important properties.

The Requirement for this approach is the knowledge of a final set of possible commands that the proactive engine can execute on the system. These commands are modelled as Rules such that they can be added into the queue and executed during the next iteration of the engine.

4.7 Strategies

The decision Scenario presented above works with priorities of the properties, which are defined in a so called Strategy. In the case of contradicting commands the planning Scenarios will consult the strategy in order to determine the most important command.

It is possible that the priorities are not always the same for all properties depending on the workload of the system or the time of the day so that these strategies are interchangeable. There might be several strategies for the system, at any time exactly one strategy is activated. Depending on some criteria, for example the peak times of the system, the strategy can be exchanged.

The design of the Strategy depends on the available commands that are possible to be in contradiction with one another. It is not necessary to mention actions that can be performed independently of the rest of the action set. The priorities are saved in form of a priority table in the database and a very simple Rule is responsible of adapting these priorities if needed.

4.8 Self-learning

The interchanging of strategies must not be predefined, it is possible to add a new property to the system by making the choice of the strategy dynamical. The choice of the strategy influences the performance of the system. In this case the Rule that is responsible for adapting the priorities in the database table will include a logic to determine the "system-wellness". Furthermore, it is necessary to remember previous choices of the priority together with the corresponding "system-wellness" over a certain timespan. From the collected data it can then be determined which priorities are the best choice at which time.

Chapter 5

Proof of Concept Example

This chapter is dedicated to the prototype implementation of a proactive engine that is used to make an existing structure an autonomous system. I am giving a small introduction explaining why this prototype is a suitable example. Then the existing setting and the goal to reach is presented. The sections 5.3, 5.4, 5.5 and 5.6 are dedicated to the presentation of different parts of the Brain. Section 5.7 gives the implementations.

5.1 Introduction

As a proof of concept for autonomous systems, the example of a server farm was chosen. A server farm is a collection of a certain number of servers which are in charge of answering user requests. Server farms are a special form of data centres. The proactive engine will be used in order to make the server farm an autonomous system. This means the server farm does not depend on human interaction anymore in order to ensure correct functioning.

Big and well-known server farms include **Google's Datacentres** - estimated around 13 server farms throughout the world - **Amazon** - about 450,000 servers in 7 datacentres in the world - **Facebook** - a massive datacenter with 62,000 m^2 - and **Microsoft** - one of the first to build their own datacentres. Nowadays there are a lot of so called cloud providers, which build datacentres and rent calculation power to smaller companies. [1]

The proof of concept engine that is presented in this work is a basic example and includes the parts that are necessary, in order to prove the new approach of building an autonomous system using a proactive engine as a Brain.

The example given in the following concentrates on fundamental implementations of the properties. In a real world example there are more aspects and constraints that need to be taken into account when designing the proactive Scenarios and property Rules, such as security and more complex initial system settings. But for the purpose of proving that a proactive engine can be used as central control unit in an autonomous system, this can be disregarded.

In the following the "Brain engine" or "proactive Brain" or only "Brain" is the newly designed proactive engine that is added to the server farm, "the existing system".

5.2 The setting

This proof of concept example is based on a reduced model of a server farm, but the idea can also be applied to real world server farms or other more complex systems. The modelled server farm is fed with user requests by a dispatcher component and managed via an interface by a system administrator.

The goal of the present work is to make systems autonomous using a proactive engine that takes over management functions and controls the system in order to reduce the work of the system administrator to a minimum.

The existing server farm and the proactive engine making the system autonomous are presented in detail in the following.

5.2.1 Existing server farm

In order to understand the procedure of making a system autonomous using a proactive engine we must first have a look at the existing situation. Figure 5.1 depicts the collection of servers in the farm. The central entry point for the user requests is the dispatcher which is in charge of distributing the requests evenly over the available servers. Each request is answered by one of the server instances. The state of the system is saved in a central database based on the activities of the dispatcher and the servers. A detailed structure of this database is given in Section 5.3.

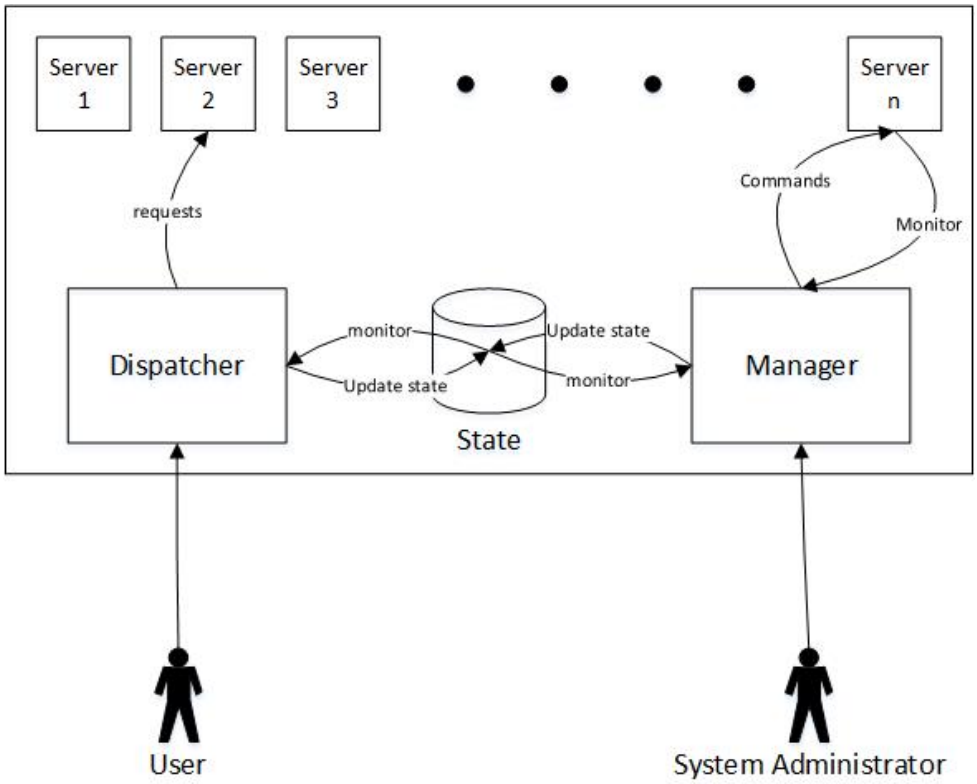

Figure 5.1: A server farm with System Administrator

The component which is of particular interest for this project is the manager, which is an interface that gives the system administrator the possibility to control the behaviour of each server. Control functions include scheduling regular backups for each server instance and the treatment of failures. In case of a failing server the server might be shut down and restarted using the last backup. Another interesting approach can be the regulation of the number of active servers depending on the amount of requests sent by the users.

The central database is hereby accessed by the manager and the dispatcher. The dispatcher saves the incoming requests to the database and reads from the state which servers are

currently available for the treatment of requests. The manager component reads the requests and creates statistics about the workload balance over a time period. Thereby the system administrator can schedule the backups, put servers to sleep during calm periods and make all servers available for request processing during peak times. All commands from the system administrator to the servers are recorded in the database.

Another task of the manager component is the monitoring of the servers so that the response time for each request can be saved in the database and an alert can be given in case of failure or malfunctioning of one of the servers.

5.2.2 Adding autonomy

In the following it is demonstrated that it is possible to give the existing server farm all the properties needed to be considered as autonomous by adding a dedicated proactive engine and Scenarios to it. The Brain component that was developed during my internship accesses the internal state of the system and communicates with the manager interface, allowing it to take over the tasks of the system administrator. Figure 5.2 depicts the new situation of the autonomous server farm. This example uses a top-down approach by implementing the necessary Scenarios and Rules in the proactive engine in order to give the autonomous properties to the system. This new system, supervised by the Brain does not need a system administrator, meaning that the only interacting humans are the users that send the requests to the server farm.

The interaction between the Brain component and the existing system is done over the state database and the manager interface. The engine can read the current situation of the system and monitor the servers via the database. The proactive Scenarios in the Brain analyse the data and decide about action to be performed on the servers. These actions are transmitted using the manager interface. The proactive Scenarios in the Brain are explained in Section 5.5.

5.3 The state

The central database of the server farm contains the necessary information about requests and servers for the engine to be able to analyse the current situation of the system and determine actions to be performed.

The first table in Figure 5.3 is populated by the dispatcher, every incoming request is saved as a new entry in the `Requests` table. The origin of the received request is saved along with a description of the request and the `ServerID` of the server the request was given to. Later the manager will complete the database entries by monitoring the server that handled the request and by determining the `ResponseTime` for each request.

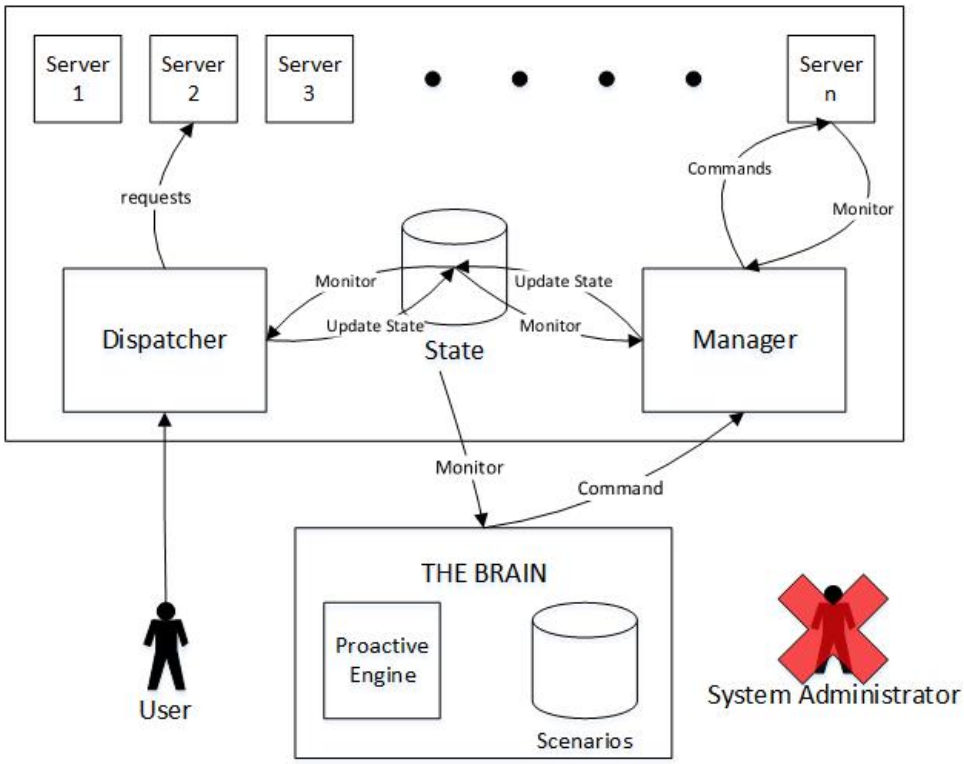

Figure 5.2: The proactive engine makes the server farm autonomous

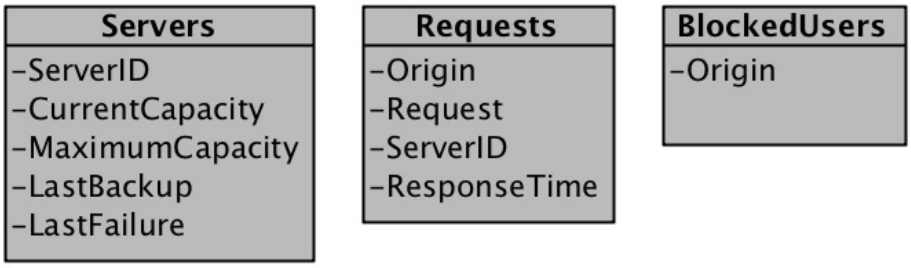

Figure 5.3: The State Database

Besides there is a table `Servers` in the state database which contains information about the usable capacity of the servers. Each server has a certain level of usable resources and a maximum capacity expressed in number of requests per minute. For example a server might only use 50% of the available resources to treat requests. In this case the `currentCapacity` would be set to 50 by the manager, the dispatcher can read this information and send less requests to the given server. A server that is not running at full capacity produces less heat, thus the energy consumption of the server farm can indirectly be reduced.

The last table `BlockedUsers` includes the users that where identified as malicious or possible attackers by the proactive Scenario that is responsible for the self-protection of the system. Future requests from these users are no longer accepted or treated. This table is thus populated by the Brain.

The information of the state database can be accessed by the Brain. The engine can then analyse this information, determine actions and give commands to the manager in order to change the state.

5.4 The control loop

The control loop is the essential part of the developed Brain. Figure 5.4 shows the different steps of the MAPE loop [18]. The previously existing proactive engine was modified so that the four tasks can now be performed in the loop. In the following the different steps of the loop are defined precisely in relation to the example of the server farm.

Monitoring is the collection of data from the database. The proactive engine has access to the state database of the server farm. The dispatcher and the manager components populate this database with the internal information. The proactive Scenarios read the data that is relevant for the property they are responsible for.

Analysis of this data is done in the proactive Scenarios that are depicted in the next section. These Scenarios analyse their collected data regarding a property of the system and conclude on a local decision.

Each of these local decisions is saved in the **List of recommendations**. This list is necessary because each Scenario analyses the state only through a special filter. This means the systems state is analysed with regard to each property independently. An overall decision can then be drawn from the entries in this list.

The list is then treated in the **Plan (or Decide)** part of the control loop. The list of recommendations is analysed whether there are contradicting or overwriting Rules. Depending on an overall goal the priorities of the commands can be set differently. The result of this step is a set of command Rules.

The **Execution** step includes the generation of the Rules and their execution on the manager.

In this proof of concept example monitoring and analysis is done within the property Rules and the decision and execution is included within the decision Scenarios. The visualization in Figure 5.4 shows a single instance of a MAPE cycle. Many instances run in parallel due to the design of the proactive engine, meaning that the reaction to changes in the system is very fast.

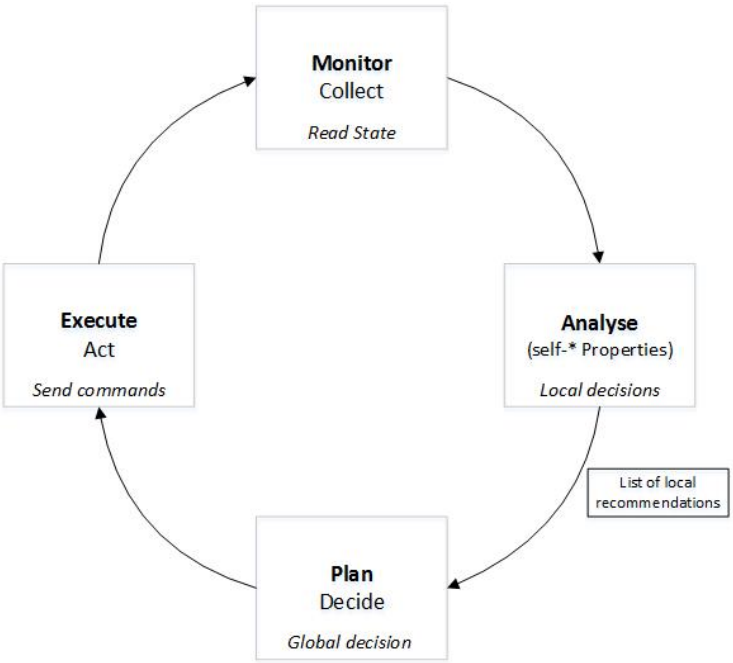

Figure 5.4: Illustration of the MAPE-Loop

5.5 Scenarios implementing the Properties

5.5.1 Self-configuration

In the given system of the server farm the self-configuration is defined as follows. The dispatcher must immediately adapt to the changing resources, since the servers that are available for the treatment of the incoming requests can change their current capacity in order to be cost efficient. The dispatcher uses a list of capacities of the servers as input for the distribution algorithms for the request. This list is kept up-to-date with the proactive self-configuration Rule.

5.5.2 Self-healing

This property includes regular backups for the servers. Each server should have one backup a day. Thus for each existing server there is one Rule which checks for the last backup and instructs a new backup if needed. Backups are necessary to guarantee recovery from failures in the systems.

These backups include the system backup of the servers. The data backup is not included in this Scenario, it is assumed that necessary data is stored in a different system, which

is responsible for backups and data redundancy, for example "the cloud". Nevertheless it would be possible for the proactive engine to also include data backup in the autonomous system. The proof of concept presented in this thesis is limited and does not take all possibilities into account.

5.5.3 Self-optimisation

Optimisation is about improving the cost efficiency of the system. There are two Scenarios in this proof of concept that aim to enhance the performance of the server farm. The two Scenarios are contradicting as optimisation can be seen from many different aspects. The proactive engine includes an interchangeable strategy that will solve this problem and in order to show the functioning of this idea the Scenarios were chosen as follows:

The first optimisation Scenario for this proof of concept is in charge of decreasing the energy costs for the server farm. The number of incoming requests is measured and if this number is small some of the servers can be put to sleep in order to reduce the energy consumption of the system.

The second optimisation Scenario improves the performance of the system towards the user. The goal is to reduce the response time for the requests to a minimum. In order to achieve this, the number of incoming requests is analysed and a recommendation to increase the capacity or switch on sleeping servers will be given.

5.5.4 Self-protecting

A system that protects itself must consider two aspects of protection; The reaction to an attack in order to ensure that the consequences stay as insignificant as possible as well as the proactive protection before the attack has occurred.

The manager component is monitoring the servers and writing their state in the database. If the Brain sees anomalies in the behaviour of one of these servers an attack can be assumed. The **reaction** can be to disconnect this server from the network so that a malicious attacker can no longer harm the system. With this kind of action the system should be protected against spoofing attacks. Spoofing includes the masking of IP Addresses and is e.g. often used within Denial of Service (DoS) attacks [20].

Furthermore, the Brain can analyse the list of requests in the state database if one particular user is sending multiple suspicious requests. A **proactive** solution would be to block future requests from the same origin.

5.6 Decision of Actions

This section explains the process of finding the set of action to be performed on the server farm. From the list of recommendations feed by the proactive Scenarios implementing the properties of the system, a set of actions will be determined autonomously. The system thus meets all properties listed above.

5.6.1 Decision Scenario

This Scenario is in charge of the first sorting of the results from the property Rules. The recommendations produced during the analysis of the data are retrieved from the database. The recommendations are grouped so that logically dependant commands can be analysed together. In the case of the server farm the blocking of users is independent from the commands concerning the servers. The commands for the servers are grouped by server ID so that a decision can be taken for each server.

The groups of recommendations are given to two new decision Scenarios presented below, that will create lists of actions from their input.

5.6.2 Block User decision

All recommendations that want to block a user end up in this Rule, the only further treatment is the deletion of double entries. In a rare case it could happen that the analysis of the incoming requests is performed twice before the decision Scenario has emptied the list of recommendations so that there are two orders for the same user. If no double entries are found, all command Rules are generated resulting in the users being added to the blocked list in the database.

5.6.3 Server decisions

The recommendations for the servers are grouped by `serverID`. For each server one of these Scenarios is created, the input is the list of recommendations concerning this server. The task of this server decision Rule is to analyse the proposals for the given server and decide depending on the current strategy which action results. This Rule generates exactly one action for the given server.

5.6.4 Set of possible commands

The list of recommendations passed from the analysis step to the plan Scenario as well as the set of commands given to the manager after the plan Scenario are subsets of the following list of possible commands.

- **Change Capacity to x%** - Reduces or increases the performance of a server. A server which uses only 50% of its resources to respond will only deal with half of the maximum number of requests per minute.
- **Start a Backup with x%** - A part of the servers resources are used to perform a backup, this entails a reduced power for request handling.
- **Shut Down** - Shuts down the server the command was sent to.
- **Switch On** - Switches on the server in question.
- **Block a User x** - Blocks further requests from a given origin.

5.6.5 Strategies

The Strategy holds a priority list for the commands which will influence the processing of the list of recommendations in the decision Scenarios. Depending on some constraints the strategy can direct the system to an overall goal.

For example a server should not be shut down and run at 100% at the same time. However, these two actions could be contained in the list of recommendations. The decision being made might depend on external conditions which are mirrored in the priority list of the strategy. In the case of the server farm the decision is related to the time of the day. In our example we will use two different strategies dividing a 24h day in night and day time. The different strategies can be interchanged depending on the day or night time.

The day strategy prefers high capacities of the servers such that the performance of the system is increased and the user requests are answered as fast as possible. During night time the lower capacities are preferred in order to avoid high electricity costs.

5.7 Implementation

The server farm is modelled using the database tables presented in Section 5.3 The State. The dispatcher, the manager and the servers are not implemented as real components. The Brain is a fully functioning proactive engine, that accesses the state database and has an internal database. In the following the individual parts of the Brain engine are presented in detail.

5.7.1 Property Rules

The so called Property Rules are in charge of analysing the state according to a certain property of the system and find local decisions that they save in the recommendation list.

Self-Configuration

Description: This Rule is responsible for keeping the dispatchers overview over the capacities of the servers up to date.

Parameters: None.

Data Acquisition: Reads from the state database the percentage current capacity along with the maximal capacity in requests/minute for each server.

Activation Guards: Run every 10 minutes.

Conditions: True.

Actions: Calculates the current capacity in requests/minute and sends this information to the dispatcher.

Rule-Generation: Clones itself.

Self-Healing

Description: This Rule ensures regular backups for each server. An instance of this Rule exists for each server.

Parameters: The server-ID

Data Acquisition: Reads the time of the last backup for the given server and checks if the server is not currently handling a request.

Activation Guards: Checks every day.

Conditions: The backup is more than a day ago and there is no request running.

Actions: A backup Rule for this server is saved in the recommendation list.

Rule-Generation: Clones itself.

Self-Optimisation1

Description: This Rule is responsible for the energy consumption reduction.

Parameters: None.

Data Acquisition: Reads on one side the average number of requests per minute for the timespan of the last twenty minutes and on the other side the current capacities of the servers.

Activation Guards: Checks every 20 minutes.

Conditions: The sum of the current capacities highly outnumbers the incoming requests.

Actions: Shut Down Rules for some servers will be added to the recommendation list.

Rule-Generation: Clones itself.

Self-Optimisation2

Description: This Rule is in charge of reducing the response time for the user requests.

Parameters: None.

Data Acquisition: Reads on one side the average number of requests per minute for the timespan of the last twenty minutes and on the other side the current capacities of the servers.

Activation Guards: Checks every 20 minutes.

Conditions: The number of the incoming requests outnumbers the sum of current capacities.

Actions: Change Capacity or Switch On Rules for concerned servers will be added to the recommendation list.

Rule-Generation: Clones itself.

Self-ProtectingPro

Description: This Rule is responsible for blocking malicious users.

Parameters: None.

Data Acquisition: Reads the origins from the last 20 incoming requests.

Activation Guards: Checks once per hour.

Conditions: One user submits more than half of the last 20 requests.

Actions: Adds a Block User Rule for this IP-Address to the recommendation list.

Rule-Generation: Clones itself.

Self-ProtectingRe

Description: This Rule is responsible for disconnecting servers in case of an attack.

Parameters: None.

Data Acquisition: Reads the response times from the last 20 requests together with the server IDs of the servers that treated these requests.

Activation Guards: Checks every 20 minutes.

Conditions: A response time is higher than 2 minutes.

Actions: Adds a Shut Down Rule for the concerned server to the recommendation list.

Rule-Generation: Clones itself.

5.7.2 Command Rules

Block User

Description: This Rule is used to prevent a certain user from sending further requests.

Parameters: The IP-Address of the user

Data Acquisition: None.

Activation Guards: True.

Conditions: True.

Actions: Adds the IP-Address to the database table Blocked Users.

Rule-Generation: None.

Change Capacity

The implementation of the `Change Capacity` Rule is shown in Listing 5.1

Description: This Rule is responsible for changing the current capacity of a server.

Parameters: New capacity in per cent and the server ID

Data Acquisition: Reads the old capacity from the database and checks whether the server is currently treating a request.

Activation Guards: The capacity can only be decreased if the server is not treating a request at that time.

Conditions: True.

Actions: Changes the current capacity for the given server in the database table `Servers`.

Rule-Generation: Clones if not activated otherwise none.

Listing 5.1: The implementation of `Change Capacity` Rule

```
1  public class ChangeCapacity  extends AbstractRule{
2
3    private long server_id;
4    private int newCap;
5    private int currentCap;
6    private boolean hasRequest;
7
8    /**
9     * Default constructor, mandatory for Hibernate to build object
10    */
11   private ChangeCapacity() {
12     setType(RuleType.SCENARIO);
13   }
14
15   public ChangeCapacity(long server_id, int newCap) {
16     this();
17     setServer_id(server_id);
18     setNewCap(newCap);
19   }
20
21   protected void dataAcquisition() {
22     currentCap = ((ServerfarmDbWrapper)engine.dataEngine).getCapacity(
           server_id);
23     hasRequest = ((ServerfarmDbWrapper)engine.dataEngine).hasRequest(
           server_id);
24   }
```

```
25
26   protected boolean activationGuards() {
27     if (currentCap <= newCap || !hasRequest){
28       return true;
29     } else
30       return false;
31   }
32
33   protected boolean conditions() {
34     return true;
35   }
36
37   protected void actions() {
38     ((ServerfarmDbWrapper)engine.dataEngine).updateCapacity(server_id,
         newCap);
39   }
40
41   protected boolean rulesGeneration() {
42     if (!super.getActivated()){
43       return true;
44     } else
45     return false;
46   }
47
48   public String toString() {
49     return "Change Capacity of Server "+this.server_id+" to "+this.
         newCap+"%.";
50   }
```

Shut Down

Description: This Rule is responsible for shutting down a server.

Parameters: The server ID.

Data Acquisition: Checks if the server is currently treating a request.

Activation Guards: The server is not treating a request.

Conditions: True.

Actions: The current capacity of the server is changed to zero in the database table `Servers`. (Which is a model of physically shutting down the server.)

Rule-Generation: Clones if not activated otherwise none.

Switch On

Description: This Rule is responsible for switching on a server.

Parameters: The server ID.

Data Acquisition: None.

Activation Guards: True.

Conditions: True.

Actions: Changes the current capacity of the given server to 100 in the database table `Servers`.

Rule-Generation: None.

Start Backup

Description: This Rule simulates that a backup is run on a server.

Parameters: The server ID.

Data Acquisition: Checks whether the server is treating a request at this moment.

Activation Guards: The server is currently not treating a request.

Conditions: True.

Actions: Changes the current capacity to zero for the given server in the database table Servers

Rule-Generation: `Stop Backup` Rule for this server.

Stop Backup

Description: This Rule simulates that the backup is finished on the given server.

Parameters: Server ID and a time.

Data Acquisition: None.

Activation Guards: The time given as parameter is smaller or equal to now.

Conditions: True.

Actions: The current capacity of the given server is changed to 100 in the database table Servers.

Rule-Generation: Clones if not activated otherwise none.

5.7.3 Decision Rules

Decide Scenario

Description: This Scenario is responsible for the first analysis of the list of recommendation. The entries are logically grouped.

Parameters: Timestamp last execution.

Data Acquisition: The list of recommendations from the internal database of the engine.

Activation Guards: Every 5 minutes.

Conditions: Non-empty list of recommendations.

Actions: Creates input lists for Decision Rules.

Rule-Generation: Clones itself.

Block User Decision Rule

Description: This Rule generates the actions for blocked users.

Parameters: The subset concerning blocking users of the list of recommendation.

Data Acquisition: None.

Activation Guards: Non-empty input list.

Conditions: True.

Actions: Creates the list of actions after deleting double entries.

Rule-Generation: Creates Rules for every action.

Server Decision Rule

Description: This Rule generates actions concerning the servers. For each server occurring in the list of recommendations one instance of this Rule is generated.

Parameters: The subset concerning a discrete `serverID` of the list of recommendations

Data Acquisition: None.

Activation Guards: Non-empty input list.

Conditions: True.

Actions: Analyses the given actions depending on the current strategy and decides an action for the given server.

Rule-Generation: Creates a Rule for the decided action. Can be a `Shut Down`, a `Switch On` or a `Change Capacity` Rule.

5.7.4 Database Wrapper

The database wrapper implements the access to the database tables of the internal state of the server farm. This is a very important connection point that enables the Brain to control the existing system which makes it autonomous.

In this proof of concept example the database wrapper on one hand includes functions to read the current state of the system that are used in the monitoring part of the control loop where data about the server farm is collected. On the other hand the commands given to the manager are modelled by changing the state directly in the database.

Some of the methods to access the database of the server farm in order to determine the state are shown in the Listing 5.2

Listing 5.2: Some methods of the Database Wrapper

```
1  public ResultSet getAllCapacities() {
2    String q = "SELECT idServers, currentCapacity FROM Servers";
3    return MySQLOperations.getArrayFromSelect(conn, q);
4  }
5
6  public ResultSet getLastRequests(){
7    String q = "SELECT Origin, ServerID, ResponseTime FROM Requests WHERE
         Arrival >= NOW() - INTERVAL 2 Hour;";
8    return MySQLOperations.getArrayFromSelect(conn, q);
9  }
10
```

```
11 public int getCapacity(long server_id){
12   String q = "SELECT currentCapacity FROM Servers WHERE idServers = "+
         server_id;
13   return MySQLOperations.getIntFromSelect(conn, q);
14 }
15
16 public long updateCapacity(long server_id, int capacity) {
17   String q = "UPDATE Servers SET currentCapacity = " + capacity + "
         WHERE idServers =  "+ server_id;
18   return MySQLOperations.executeUpdate(conn, q);
19 }
20
21 public boolean hasRequest(long server_id){
22   String q = "SELECT serverID FROM Requests WHERE responseTime = -1";
23   ResultSet a =  MySQLOperations.getArrayFromSelect(conn, q);
24   return true;
25 }
26
27 public long blockUser(String ip_address){
28   String q = "INSERT INTO BlockUsers (`IPAdress`) VALUES ('"+ip_address+
         "')";
29   return MySQLOperations.executeUpdate(conn, q);
30 }
```

5.7.5 Internal database of the proactive Brain

A proactive engine works with an internal database shown in Figure 5.5 where the necessary system parameters are saved in `SystemParameters` and `Signals`. Further the database includes tables for each kind of Rule and a table to save Rule IDs of the Rules currently in the queue so that the engine can recover easily in case of a failure. The first row of tables in the figure are therefore dedicated to the functioning and tuning of the engine and its iterations.

The second row represents the database tables, which are used to save the command Rules. Each Rule has a belonging table in the database to store the parameters. The IDs of these Rules are then saved in the queue. The saving of the Rules is a sort of backup that makes recovery easy in case the engine fails. The third row represents the tables that are used to save the property Rules of the engine.

Further the Brain needs in comparison with the existing proactive engine two more tables in its database. These hold the list of recommendations that is populated by the property Rules and the list of actions which is produced by the decision Scenarios.

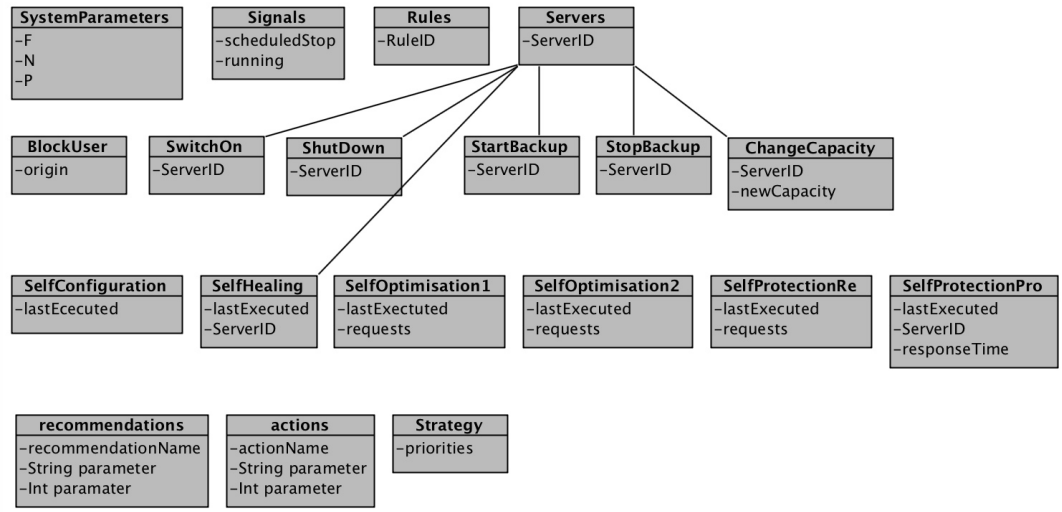

Figure 5.5: Internal Database of the proactive engine

5.8 Limitations of the proof of concept implementation

This chapter presents a proof of concept which is deliberately kept simple. The given example was developed to prove the idea that autonomous systems can be designed using the technology of proactive computing. The goal was to use the existing proactive engine with as little changes as possible to add autonomy to systems.

In the given example of the server farm complex protections or healing procedures for the servers are not treated.

The so called monitoring of the users and servers that is implemented is restricted to simple decisions. In reality constraints are more complex implying that the Scenarios listed above give a minimal idea of what is possible to achieve with this technology.

Also the recovery of a request in case the server fails during the treatment of the request is not part of this proof of concept as it is assumed that this part is already included in the existing server farm. Failing servers are identified by the manager, thus a current request of this server has no `ResponseTime` and should be restarted by the dispatcher on a different server. Therefore, requests have to be capable to produce roll-back procedures as inconsistent data must be avoided by all means.

In a real system the overall work-flow of the control loop and the used techniques stay the same, the complexity is included in the property Rules and the command Rules.

Since the complete server farm is modelled as database tables there are admittedly no real backups that are performed and no real requests that are treated. Furthermore, this example does not include the unblocking of users.

As a last remark, one should consider that the form of the `Recommendations` and `Actions` table in the internal database of the engine depends on the set of possible commands. It is deliberately kept simple in this example. In bigger systems the best solution might be to only save the ID of the Rule, as it is done for the `Rules` table which represents the queue of the proactive engine.

Chapter 6

Conclusion

6.1 Future work

Seeing as the here presented Master Thesis only includes a simplified example, the next step in the usage of proactive engines in the field of autonomous systems would be to develop a Brain component for a real existing system in order to make it autonomous.

Furthermore, there are some other improvements that could be undertaken within the Brain component which were not realised during the internship. In the proof of concept Scenario the advisor runs a simple algorithm to change the strategies depending on the time of the day. One could imagine more complex strategies and decision algorithms that include peak times of the system, in order to make the Brain suitable for bigger software in the industry.

A future version of the Brain should include the self-learning property briefly presented in Section 4.8. The best strategy should be chosen dynamically. If the performance of the system can be measured and is influenced by the different strategies, a self-learning Rule could learn from previous strategy choices and thus adapt the decision algorithm. In this case the Brain includes an adaptable proactive Scenario which chooses the current strategy. In addition a proactive observation Scenario would monitor and compute the "systems-wellness" and would then draw a conclusion from the outcome.

During the development of the Brain, the question raised, whether it would be a good solution to have the decision stage as a Scenario in the iterations of the proactive engine or if an improved solution where the decision step is called between the iterations should be developed. In this case it would guarantee that all property Rules were to be executed before the decision Scenarios is run so that all recommendations would be taken into account.

6.2 Personal feedback

This Master Thesis is a first step into the usage of proactive engines in the field of autonomous systems. The proactive engine that has been developed by the research team around Prof. Dr. Denis Zampunieris at the University of Luxembourg is very flexible in its utilisation. With this work I demonstrated that it can also be used in order to build autonomous systems or to make an existing system autonomous by adding a proactive Brain component to it. The internship I performed in the research team during my Master's Degree included research work in the field of autonomous systems and the adaptation of the existing proactive engine, allowing it to act as a central control unit within an autonomous system.

The initial objectives of creating a transformed proactive engine to support systems were met in a straightforward way. The time limit of the internship made the design of the Brain and the development of a proof of concept example feasible. The newly created technique was not applied to a real world system, but the presented work does include a reduced example to show its functioning.

The existing proactive engine was not transformed on a conceptual basis. In order to serve as Brain the decision Scenario and the possibility to use a strategy for the given decision was added. This seemed to be the most logic and straightforward solution to the initial assignment of building the possibility to make existing systems autonomous.

In this sense I am proud that I was put into the situation to be able to provide an introduction to the topic of the analysis and design of autonomous systems with the help of proactive computing, which has the potential to grow further in many different areas.

Abbreviations

AC: Autonomous Computing

AI: Artificial Intelligence

AS: Autonomous System

DoS: Denial of Service

IoT: Internet of Things

IP: Internet Protocol

LMS: Learning Management System

PE: Proactive Engine

TCO: Total cost of ownership

Glossary

Autonomous Property: Autonomous or self-* properties give a system the ability to be independent of human interaction and act on its own.

Autonomous System: An autonomous system is a stand-alone system that is able to function in a changing environment without human interaction.

Dispatcher: Part of the server farm. Entry point of the requests. Responsible for a balanced distribution over the available servers.

Proactive Engine: A proactive engine is a Rule Running System, that works for the user. It can react to events and the absence of events.

QueueManager: The QueueManager is the control unit of a proactive engine. It is responsible to execute the Rules in the queues.

Rule: A Rule is a Java class in a specific form. Each Rule is responsible for a particular task.

Scenario: Scenarios are particular Rules that stay in the engine. Often monitoring tasks are done in Scenarios. They clone themselves in their `RuleGeneration` method.

Server farm: A server farm is in this context a system including several servers, able to answer requests that are distributed to them by a dispatcher.

Strategy: A strategy is a plan, that gives priorities to certain properties of the system.

System Administrator: Human person in charge of management and monitoring of a system.

Bibliography

[1] Facts and Stats of Worlds largest data centers. https://storageservers.wordpress.com/2013/07/17/facts-and-stats-of-worlds-largest-data-centers/.

[2] B.H.C et al. Cheng. Engeneering self-adaptive systems through feedback loops. In *Software Engeneering for Self-Adaptive Systems, Lecture Notes in Computer Science*, volume 5525, pages 1–26. Springer, 2009.

[3] Michael Chui, Markus Löffler, and Roger Roberts. The Internet of Things. http://www.mckinsey.com/insights/high_tech_telecoms_internet/the_internet_of_things.

[4] Sergio Marques Dias, Sandro Reis, and Denis Zampunieris. Proactive Computing Based Implementation of Personalized and Adaptive Technology Enhanced Learning over Moodle. In *IEEE 12th International Conference on Advanced Learning Technologies*, pages 674–675, Rome, Italy, July 2012. IEEE.

[5] Remus-Alexandru Dobrican and Denis Zampunieris. *Moving Towards a Distributed Network of Proactive, Self-Adaptive and Context-Aware Systems*. The Sixth International Conference on Adaptive and Self-Adaprive Systems and Applications, 2014.

[6] S. Dobson et al. A survey of autonomic communications. In *Auton. Adapt. Sys.*, volume 1, pages 223–259. ACM Trans., 2006.

[7] A. Ettel and H. Zschäspitz. Verheerender Dominoeffekt löst Kurssturz aus. http://www.welt.de/finanzen/article7523344/Verheerender-Dominoeffekt-loest-Kurssturz-aus.html.

[8] Vivek Gite. Explain: Tier 1 / Tier 2 / Tier 3 / Tier 4 Data Center. http://www.cyberciti.biz/faq/data-center-standard-overview/.

[9] IBM Thomas J. Watson Research Center. *The Vision of Autonomic Computing*. IEEE Computer Society, 2003.

[10] IBM Thomas J. Watson Research Center. *The Vision of autonomic computing*. IEEE Computer Society, January 2003.

[11] David Kramer, Rainer Buchty, and Wolfgang Karl. Monitoring and self-awareness for heteogenous, adaptive computing systems. In *Organic Computing - A paradigm Shift for Complex Systems*, pages 163–177. Springer, 2011.

[12] Philippe Lalanda, Julie A. McCann, and Ada Diaconescu. *Autonomic Computing Principles, Design and Implementation*. Springer, 2013.

[13] Marlene Müller. Bachelor Thesis - Connection of Proactive Engines (Université du Luxembourg), 2013.

[14] Gilles Neyens. Bachelor Thesis - Communication of Proactive Engines (Université du Luxembourg), 2013.

[15] Steve Omohundro. Autonomous technology and the greater human good. *Journal of Experimental and Theoretical Artificial Intelligence*, pages 303–305, 2014.

[16] Jeongmin Park, Chulho Jeong, and Eunseok Lee. Proactive self-healing system for application maintanance in ubiquitous computing environment. In *ICCSA*, pages 430–440. Springer-Verlag Berlin Heidelberg, 2006.

[17] Roy Sterrit, Mike Hinychey, and Emil Vassev. Self-managing software. In *Encyclopedia of Software Engeneering*, pages 1072–1081. Taylor and Francis: New York, 10 Nov 2010.

[18] Roy Sterritt and David W. Bustard. Autonomic systems. In *Encyclopedia of Software Engineering*, pages 111–117. Taylor and Francis: New York, 10 Nov 2010.

[19] techterms. Internet of Things. `http://techterms.com/definition/internet_of_things`.

[20] techterms. Spoofing. `http://techterms.com/definition/spoofing`.

[21] David Tennenhouse. Proactive computing. *Communications of the ACM*, pages 43–50, May 2000.

[22] University of Luxembourg. *Enhancing Mobile Devices with Cooperative Proactive Computing*, 2014.

[23] Mark Weiser. The computer for the 21st century. *Scientific American Ubicomp Paper*, 1991.

[24] Mona A. Yahya, Manal A. Yahya, and Dr. Ajantha Dahanayake. Autonomic computing: A framework to identify autonomy requirements. In *Complex Adaptive Systems*. Procedia Computer Science 20, 2013.