Computing and Informatics, Vol. 37, 2018, 424–456, doi: 10.4149/cai_2018_2_424

USE OF SELF-HEALING TECHNIQUES FOR HIGHLY-AVAILABLE DISTRIBUTED MONITORING

Włodzimierz Funika

AGH-UST, Faculty of Computer Science, Electronics and Telecommunication
Department of Computer Science, al. Mickiewicza 30, 30-059 Krakow, Poland
e-mail: `funika@agh.edu.pl`

Abstract. The paper addresses the self-healing aspects of the monitoring systems. Nowadays, when the complex distributed systems are concerned, the monitoring system should become "intelligent" – as the first step it can guide the user what should be monitored. The next level of the "intelligence" can be described by the term "self-healing". The goal is to provide the capability that a decision made automatically by the monitoring system should force the system under monitoring to behave more stable, reliable and predictable. In the paper a new monitoring system is presented: AgeMon is an agent based, distributed monitoring system with strictly defined roles which can be performed by the agents. In the paper we discuss self-healing in the context of monitoring. When the self-healing of the monitoring system is concerned, a good example is the case where it is possible to lose the monitoring data due to the storage problems. AgeMon can handle such problems and automatically elects substitute persistence agents to store the data.

Keywords: Monitoring, self-healing, distributed systems, reliability, high availability

Mathematics Subject Classification 2010: 68-M14, 68-M15

1 INTRODUCTION

Nowadays computer systems become more and more complicated. This statement is especially up-to-date when the distributed systems are concerned. The number of distributed systems is rapidly growing. A good example for this trend is cloud storages and cloud computing. A few years ago those terms were known only to a limited number of people who worked in IT industry or to scientists. Today, solutions which use clouds are available broadly for different kinds of end-users.

Complex distributed systems can be built with the components which cooperate together in order to achieve common goals. The monitoring of such components is especially challenging. Decomposition of a system results in a need in distributing the monitoring system itself. Moreover, the complexity of applications requires that the monitoring system provides some aspects of 'intelligence' – it should be possible to guide the user about what is the most optimal way of monitoring.

The most complex monitoring systems that are currently available are able to work in an autonomous way. It means that some or most of the operations are executed without user interaction. A monitoring system based on observations of an application can decide what action should be taken – for instance if any other monitoring should be performed or if a user interaction is required.

Similarly to *clouds*, terms like High Availability and Fault Tolerance are becoming very popular. The solutions which combine both High Availability and High Performance Computing (HPC) are becoming available for much more users [20]. This change drives changes in monitoring systems. The question is: *how can a Highly Available application be monitored?*

Increasing the complexity of a monitoring system results in a higher probability of faults in a system. In such a situation, the system should recover from a fault. In other words, it should be able to perform **self-healing**. Healing can be also considered from a perspective of an application. A good monitoring system in addition to a regular monitoring can provide a way of healing the application. Based on predefined rules, the system could take an action to help the application – for instance to restart its components or disconnect a failed resource.

There is a big number of monitoring systems available on the market. Unfortunately, the existing solutions do not provide all the features which are required by some of the modern applications. The existing monitoring systems:

- are sometimes used to monitor highly available applications or systems but they are not themselves highly available or fault tolerant,
- do not provide self-healing capabilities,
- are hard to deploy and complex to use,
- usually manifest problems when it comes to integration with applications (in order to heal the application).

In order to monitor a highly available application, monitoring system should also be highly available. This will minimize the risk of loosing important monitoring data gathered at system runtime.

The *main objective of our research* is to verify whether self-healing techniques can be used to build highly available and reliable monitoring systems needed for developing and maintaining highly available applications.

In order to achieve this goal, we introduce a new model of monitoring system. The model provides:

- *self-healing* capabilities which will help to implement high availability requirements; the system cannot loose any monitoring data gathered during a monitoring session,
- *distributed, loosely-coupled architecture* – the design of the system should be based on a distributed architecture, the system should be able to monitor distributed applications,
- *autonomicity* – it should be possible to define and deploy the rules and actions which will be executed automatically by the monitoring system,
- *capability of integration with an application* – the model should allow for integrating the monitoring system with an application in order to: heal it, or to provide monitoring data which can be used by an application in its regular functioning.

We have built a new monitoring system called *AgeMon* that was used to evaluate the proposed solution. This name will be used in further paragraphs of this paper.

The main objectives of the research are as follows:

- analyze different aspects of the High Availability and Self-Healing,
- evaluate if Self-Healing concepts can be used to provide High Availability solutions,
- select the best Self-Healing techniques which can be used in the monitoring systems,
- create a reusable and generic model of a Self-Healing monitoring system,
- create a prototype of monitoring system which will provide self-healing components. It should be possible to reuse these components in other monitoring systems.

The rest of the paper is organized as follows: Section 2 introduces the key concepts related to self-healing and reliability. Section 3 brings an overview of the challenges faced when monitoring the modern distributed systems. The motivation for a new monitoring system is introduced. Section 4 describes the architecture and implementation of the AgeMon system; later on, the HA and self-healing concepts in the context of monitoring systems. Section 5 presents the results of the system tests. The last section summarizes the paper.

2 RELATED WORK – RESEARCH BACKGROUND

There exist a considerable number of monitoring systems for distributed environments available on the market, some being more domain-adjustable like SemMon [22], while others are more specialized. Some of the monitoring systems are very well known and de-facto became industry standards, like Nagios [23] or Ganglia [24]. Various monitoring systems are described at the end of this section. We are going to start with introduction to some key concepts that are important to understand before deep diving into self-healing monitoring systems.

2.1 High Availability

The availability of a system determines if the system is able to provide the required service. When a service cannot be used by the user, it is said that there is an outage in the system. Downtime is duration of a time when the system is unavailable [17]. **Highly Available (HA)** system is designed to avoid losses of a service by reducing failures and downtimes of the system. System availability can be measured and is usually expressed as a percent of time when the system is available in a particular year. A system which provides 99.999 % percent of availability is considered as a high-availability system (the term *five-nines* is also used). The downtime of such a system should not take more than 5.5 minutes per year.

High Availability systems are reactive – emphasis is on a failover and a recovery [18]. In addition, *continuously available systems* group applications with a proactive approach. Such systems try to detect and prevent errors *in advance*.

2.2 Self-Healing

Self-healing is the ability of a system to recover from a failure state. Additionally, a self-healing system should be able to perceive that its own operation is not correct [8]. A healing action can be performed in an autonomous way or could require a user intervention (assisted-healing systems).

Self-healing [12] is also considered in the context of **self-managing autonomic systems**. In such systems human operator takes on a new role. He/she does not control the system in a direct way. Instead, the user defines general rules and policies that guide the self-management process.

The key questions when self-healing is concerned is whether the healing can be done automatically (*self*) or with a user interaction. In this paper, self-healing is understood as an action which should not involve any manual user interaction during the failure detection and recovery. Therefore, a system with the self-healing functionality should also be autonomous (self-healing components of the system should be autonomous).

An automaticity of the system does not exclude a user interaction, for instance in a system setup. While the failure detection and recovery should be completely automatic, human interaction may be needed to define high level rules for decision making, templates, to define data sources, properties, etc. [13, 25].

In Figure 1 a state diagram of a self-healing system is presented [8].

There are three states of the system from the self-healing perspective. The most desirable state is a normal, healthy state. The system in this state works correctly, and should fulfil all the requirements. Nevertheless, the system should periodically

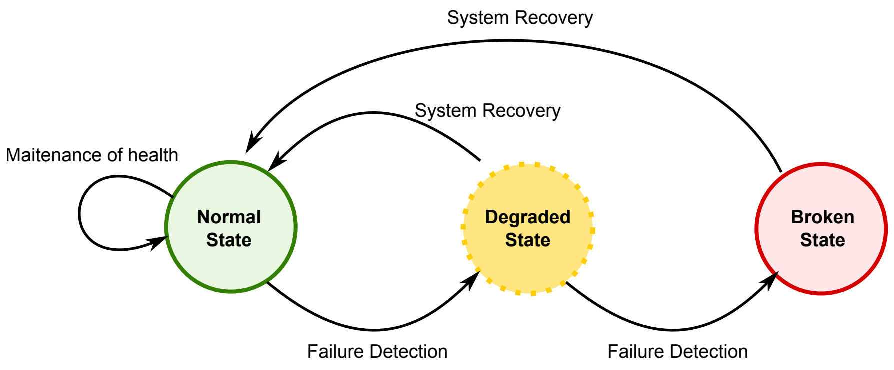


Figure 1. State diagram of the self-healing system

check its own state. It can be done by analyzing application logs. Additionally, the system needs to manage the state of its redundant components and maintain diversity – these tasks can result in failure detection, but the main intention is to keep the system healthy.

In addition to the tasks used to maintain a healthy system, there are tasks which are used to detect a failure in the system. There are multiple strategies here, for instance used to detect the missing components ([9, 10, 11]) or perform system monitoring [14].

A failure itself is a manifestation of an error caused by a system fault [3]. There is a number of classifications of errors. For instance, it is possible to classify software faults based on the circumstances needed to trigger an error [4, 5].

When a failure in the system is detected, the system is recognized as a broken one. Self-healing systems should be able to recover from this state, to heal themselves. One of the major concepts used here is the redundancy of components. With this approach, in case of a failure of a component, other components can take the responsibility of the failed component. It may decrease the performance of the system, but the system will continue to operate.

In loosely coupled environments, an additional problem is to detect a malicious fault, known also as the Byzantine Generals Problem. In order to solve such types of problems, a voting procedure should be implemented.

Other solutions for recovery are typically very specific to a particular system. Multiple attempts have been made to structure these problems. For instance, in one of the approaches, healing is done through cooperation between components. This type of recovery is becoming very popular [15, 16].

In addition to a normal and a broken state, a *degraded* state can be introduced. The system in this state can be considered working, but some of its functions could be limited. For instance, a performance indicator could be significantly degraded. If no action is performed immediately, the degraded state can be quickly turned into the broken state. As an example, let us consider an application which uses 99 % of memory. It is very probable that the system will fail soon, if no action to free the memory is performed.

2.3 Reliability

In general, software reliability is a probabilistic measure which can be defined as probability that software faults do not cause a failure during specified time [19]. From a mathematical point of view, it can be defined as the following function [19]:

$$R(t) = Pr\{T > t\} = \int_t^\infty f(x)\,\mathrm{d}x \tag{1}$$

where $R(t)$ = reliability, T = working time without failure, t = required (assumed/specified) working time without failure, $f(x)$ = failure probability density function.

From the industry perspective, there are two major metrics that are used to evaluate the reliability of a software:

- Mean Time Between Failures (MTBF) – elapsed time between failures in the system. This assumes that the system can be recovered (manually or automatically) from the failure (or the failure does not affect the overall functionality of the system).
- Failure Rate – frequency of the failures in the system.

The following equation defines correlation between MTBF and Failure Rate (λ):

$$\mathrm{MTBF} = \frac{1}{\lambda}. \tag{2}$$

Measuring the reliability is especially useful during the process of developing software. It can be used as a metric determining the current state of software and see if there are improvements during different phases of testing (unit testing, acceptance testing, integration testing, soak and stress testing).

2.4 Existing Monitoring Systems

Over years a considerable number of monitoring facilities were released. Below we give an overview of some representative monitoring systems.

2.4.1 Ganglia

Ganglia [24] is a scalable distributed monitoring system. It is deployed on more than 500 clusters over the world. It is designed to work in the high-performance computing like clusters and Grids. Its hierarchical design is based on the federation of clusters. Inside the cluster communication uses the multicast, while the communication between clusters within the federation is based on tree point-to-point connections.

Ganglia provides some aspects of high availability system. For instance, it is possible to specify multiple sources for the data in the `gmetad` components. It is used to failover in case one source in invalid. It is very scalable, but it does not provide advanced self-healing capabilities, and it is not possible to integrate this system with the monitored system in order to heal the system.

2.4.2 Autopilot

AutoPilot [26] provides an infrastructure for real-time, adaptive monitoring of distributed applications/systems. This monitoring tool can adaptively apply the data reduction based on the fuzzy logic. Autopilot allows also optimizing the application at runtime as the result of the application state observed by the monitoring system. This could also be used to *heal* the monitored system.

Autopilot is based on the Pablo Toolkit. It supports rule definition and some basic concepts of healing the application. It does not ensure high availability (it has a single point of failures) and does not provide any self-healing capabilities. It is also quite hard to use. The sensors and, especially, monitor tasks, need to be created from scratch for any new application. It does not support a common monitoring infrastructure like JMX.

2.4.3 GEMINI

GEMINI [27] is a Grid monitoring framework that fulfils space between resources monitoring components and monitoring services clients. GEMINI combines applications, infrastructure and Grid middleware resources monitoring by providing unified interfaces for data sources. As for monitoring, GEMINI performs measurements using a set of loadable modules called sensors which retrieve monitoring data by itself or using external, legacy applications for this purpose. GEMINI was developed within the K-Wf Grid Project but it is not intended to cooperate with this Grid system only; the idea behind providing a generic framework is that it could be easily adapted to various Grid environments. GEMINI is written in Java and is based on the Globus Toolkit libraries and services.

Monitors and sensors provide web-services which can be used by the clients to access the data. This solution allows developers to write extensions in an easy way. While GEMINI is very flexible, it does not allow for interactions with the monitored system. It is not a high availability system and it does not provide self-healing capabilities.

2.4.4 Aksum and JavaPSL

Aksum [28] is part of the Askalon [29] project, aiming to simplify the development and optimization of applications that can harness the power of Grid computing.

Aksum automatically searches for performance bottlenecks based on the concept of performance properties. In contrast to many existing systems, performance properties are normalized (values between 0 for the best case and 1 for the worst case), enabling the user to interpret the resulting performance behaviour.

Aksum is highly customizable, which allows the user to build or define an own performance tool. Performance properties are defined in JavaPSL [30], and may be freely edited, removed from or added to Aksum in order to customize and speedup the search process. The performance properties found can be grouped, filtered, and displayed in several dimensions as long as more experiment data become available.

The Askalon is focused on monitoring workflows of the applications deployed on the grid. It uses Globus infrastructure, therefore it is not possible to simply deploy this monitoring tool outside the grid. It does not provide an infrastructure to dynamically call the application logic, therefore its healing functionality is limited. It does not provide self-healing.

2.4.5 SemMon

The SemMon [22] is a monitoring system for distributed applications which enables adaptive monitoring. It is focused on monitoring distributed Java applications, but it can also be used to monitor OS specific capabilities.

The system is able to learn what is important to monitor in the current situation. The knowledge is gathered based on the previous user decisions. For instance, if the user decides that in the current state of the system, a specific capability needs to be monitored, the system will store this information, and use it in future to help the user or start the measurement automatically.

The definition of the monitoring capabilities (the capabilities which can be monitored by SemMon like CPU usage, number of threads) and metrics are expressed with the semantic description. They are described in the ontology, which brings an additional abstraction layer and could be used to improve the adaptation of the system. Based on the semantic dependencies between metrics, the system can automatically provide a hint to the user what needs to be monitored in the specific situation.

The SemMon system is a complete implementation of a robust system with semantics, which is not biased to any kind of underlying 'physical' monitoring system, giving the end-user the power of intelligent and computer-aided monitoring features like automatic metrics selection and collaborative work. On the other hand, it is not a high-available system. It has several single points of failures – e.g. reasoners or database with results. In addition, it does not provide any self-healing features.

2.4.6 Dynamic Monitoring Framework

Dynamic Monitoring Framework [32] is a solution which tends to address multiple challenges related to monitoring of the SOA based products. Service Oriented Architecture (SOA) is an architecture approach used in a wide area of solutions. It focuses on implementing business requirements as a service. One of the biggest advantages of such an architecture is the ability to support frequent changes of the requirements at runtime. For instance, it is possible to update a specific service without shutting down other services. It is also possible to change dependencies between services at runtime. While this approach addresses a lot of business challenges, it also adds complexity to the management and monitoring layers.

The Dynamic Monitoring Framework provides an interesting capability – ability to monitor a dynamically changing environment. Unfortunately, it does not provide any self-healing capabilities.

3 MONITORING SYSTEM MODEL

The model of monitoring system under discussion is designed to be based on the distributed, agent-based architecture. The distribution of the system enables the system to be more flexible and allows to dynamically fit into a complex *monitored* system. Since the monitored systems are often distributed, or even the monitoring system is used to monitor distributed values (like *network flow*) the distribution of the monitoring system becomes one of the most important requirements.

The distributed monitoring system usually consists of the following components: monitoring service/sensor and the user interface used to present the monitoring data to the user. It could be extended by additional components like database used to persist results or rule engines used for decision making based on the monitoring results.

One of the possible implementations of the distributed architecture may involve the agent-based approach [7]. This type of architecture brings a lot of values to the system. For instance, the scalability of an MAS system is provided very naturally. If one needs to have more resources in the system it is all about adding additional instances of agents. A similar situation is with concurrency. MAS systems are designed to be flexible and adaptable to the changing environment.

In addition, the attributes which are usually [6] discussed when MAS are concerned are *fault-tolerance* and *reliability*. The system does not have a single point of failures. The reliability is achieved collectively by all the agents of the system.

The main goal for the system model under discussion is not focused on the fully autonomous agents. Therefore, in the first implementation of the system, agents will have a limited extent of autonomy. Due to that limited autonomy, the reasoning coordination can be simplified. At the same time, in the AgeMon we address some of the common challenges with the agent based approach: communication between agents, cooperation or problem decomposition are the aspects that are implemented in the system. The agents are physically distributed, and need to cooperate in order to achieve the monitoring goals. Therefore the system under discussion is a *distributed and multi-agent* system.

3.1 Self-Healing, Distributed Monitoring System Architecture

The design of the monitoring system with the self-healing capability is considered at the very first stage. In fact, this requirement is the *driver* of the design.

There are two main requirements which need to be addressed by high-level design in order to provide self-healing of the monitoring system:

- redundancy of key components,
- high availability of the communication layer.

The redundancy of key components is the basic concept when the *high availability* feature is implemented. The redundancy is usually realized by physical or programmatic duplication of the resources/components in the system. This is a simple and straightforward approach used in different systems. Unfortunately, it has a huge disadvantage – it does not scale. For instance, if we have a system with 2 database components, and 2 oracles, addition of four more components to provide duplication would not impact the overall system performance. But in case there are 50 database components adding more 50 ones just to fulfil a redundancy paradigm could drastically degrade the system performance.

The redundancy of components is a key concept that needs to be introduced when the high availability is considered. Unfortunately, it is not enough – it is possible that the system consists of duplicated components, but these components cannot communicate with each other. Therefore, the communication layer between components/agents in the system should also feature high availability.

The design of the system under discussion should address these issues. The new monitoring system – AgeMon is intended to provide self-healing capabilities together with allocating the resources dynamically depending on the system state. The redundancy of the components should be available, but should only be used when no other way of providing high availability is feasible.

The AgeMon system is a distributed, agent based, self-healing monitoring system. The key concept in the system are *roles*– which are used to group functionality. For instance, the persistence role groups all the functionalities related to storing monitoring data in a persistent way. Use of this approach will simplify design and implementation and will also help define the best self-healing techniques for each role.

All the agents in the system are able to perform one or more roles. There are five types of the roles:

Monitoring Role – used to retrieve monitoring data and interact with the monitored system,

GUI Role – interacts with the system administrator and displays monitoring data,

CLI Role – enables advanced capabilities used by the system administrator,

Persistence Role – stores monitoring results in a persistent way (e.g. in a database),

Rule Role – dynamically reacts on changes in the system environment and performs actions based on the monitoring results.

To provide the dynamic self-healing capability in a scalable way, the roles can be automatically enabled or disabled by the agent in response to changes in the system. This is a very important aspect of the system – it is not required to have duplicated agents/roles since it is possible to start a new role only when needed. The communication between agents is based on a reliable messaging protocol. The stub of the protocol is provided by the Agent Communication Layer.

A sample deployment diagram is given in Figure 2.

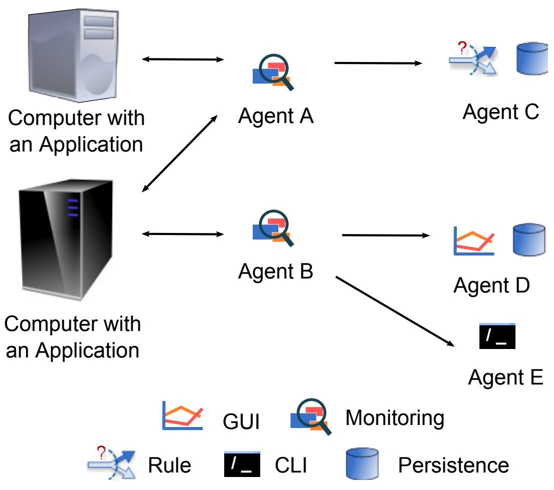


Figure 2. AgeMon System – a sample deployment

In this example scenario, two applications are going to be monitored. The arrows in the diagrams present the flow of the monitoring results which are sent between agents. There are two agents with the monitoring role enabled – **A** and **B**. Agent **B** is monitoring one application while the second agent is used to monitor two applications. The monitored data is sent to other agents. In the example, agent **C** is concurrently performing two roles – Rule and Persistence. It will be used to store the monitoring results in the database. At the same time the agent can perform user-define checks to test if the monitoring results should trigger an action. Agent **D** is also used to store the monitoring results in the database. In addition to this, it can present the monitoring data to the user. The agent **E** can be used by the system administrator to validate the state of the monitored system.

The above simple example illustrates some key-concepts of the system. It is possible that more than one role is performed by a single agent. On the other hand, it is possible that in the system there are more agents performing the same role. The agent is able to send monitoring results to more than one agent.

The next subsections provide a description of the Roles in the AgeMon System.

3.2 Monitoring Role

The monitoring role is used to retrieve the data from the monitored system and to *interact* with that system. Agents which perform this role act like sensors – they periodically ask for a new monitoring value and deliver it to the monitoring system. There are two types of measurements which can be done by the monitoring role: OS-related measurements and application specific measurements.

In the Java world, the Java Management Extensions (JMX) is a broadly accepted industry standard for monitoring and managing Java Applications [1, 2, 21]. This is not limited to the application – JMX can be used to monitor an Operating System, Networks or Devices. AgeMon utilizes this flexibility – support for JMX based monitoring is built-in. It is also possible to integrate it with other monitoring environments.

User is meant to select what he/she wants to monitor (select a *monitoring capability*) and create a monitoring task through a GUI. A monitoring agent will send a monitoring value to the destination agent(s) at the predefined intervals. Agent is also capable of *buffering* the results – it is possible to define the size of the packet used to send the data.

Whenever the monitoring data cannot be sent to the destination agent an election of a substitute agent (failover) will be triggered.

While probing the system under monitoring can be considered a fundamental task of the monitoring role – participation in handling the ‚feedback' procedure enables the *healing* of the monitored system.

Currently the system supports two *direct* ways of passing the information to the monitored system. With the first one the AgeMon system can call a method on an MBean. The MBean should be registered in the default platform MBean server. With the second way, the call can be performed on any static method defined by the user.

3.3 GUI Role

The GUI role is used to present monitoring data to the user. It is also meant to perform administrative tasks – for instance to define and deploy rules. The other rules provide important capabilities like persistence or healing, but it is possible to use the basic features of the monitoring system without them. On the other hand when the monitoring role and GUI role are not present, the system cannot operate in an efficient way.

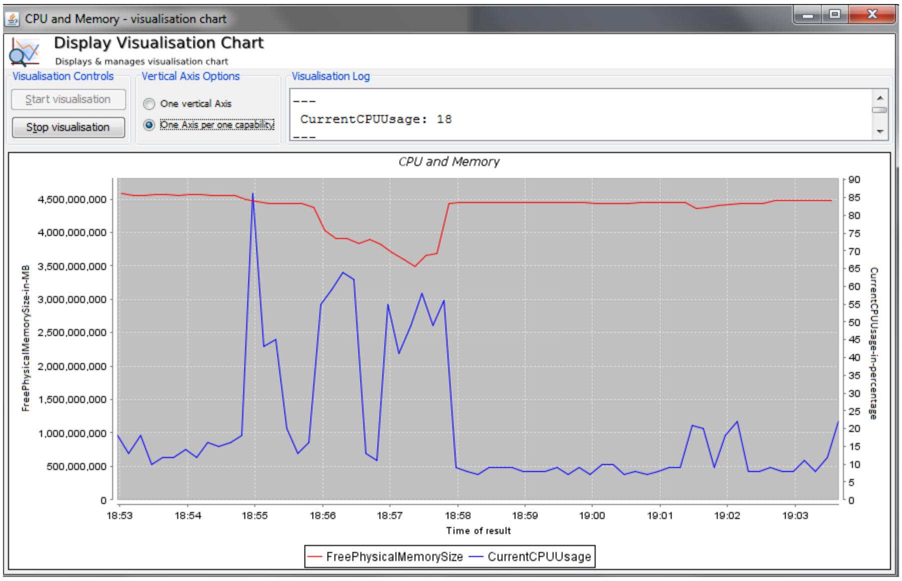
The central point of the GUI role is *Agent Graph*. It is a directed graph where each vertex represents an agent connected to a monitoring group, and each edge shows a monitoring connection created between two agents. It represents a publisher-subscriber dependency between two agents, therefore it is always directed. It is designed to execute a bunch of actions directly from a graph – for instance it is possible to stop an agent, create a monitoring and visualisation.

The *Monitoring Component* allows the user to select measurable capabilities, define a monitoring name or specify a polling interval. In addition, as a result of the creation of a monitoring task, a monitoring link between two agents is established.

The *Rule Component* provides an ability to create, manage and delete the rules. It can be only displayed in the context of the Rule Agent. A detailed description of the Rules is provided in Section 3.4.

Managing of the Persistence Agent can be done through the *Persistence Component*. The main functionality provided by this component is focused on enabling the browsing of the monitoring data in a straightforward way. It is possible to list measurement results and filter them, e.g. by a monitoring name.

The *Visualisation Component* allows the user to create and manage visualisations. It enumerates the available monitoring sessions and allows the user to create a new visualisation or attach to the existing one.



One vertical Axis [✓ One Axis per one capability]

```
---
CurrentCPUUsage: 18
---
```

Figure 3. Example visualisation – two measurements: CPU utilization and memory usage are presented on the same chart

A sample visualisation is presented in Figure 3. It presents an ability to display different monitoring results on one chart. In this example *CPU Usage* and the *Free Physical Memory* is presented at the same time. Due to it, the chart contains two different vertical axes – one per each capability. This component provides more advanced options like a dynamic scale, dynamic rendering, zooming or printing.

3.4 Rule Role

The Rule role introduces automation into the system. This is a very important feature, especially when complex, distributed systems are under monitoring. The user will not be able to capture all the issues, as well as to check all the results in every minute over a long period of time. Carefully designed and implemented rules will introduce a lot of new functionalities without additional cost.

A short definition of the *Rule* states that Rules are used to identify a change in the state of the monitoring and monitored systems. For instance, it can identify that a new Agent is attached. It can be also used to detect that CPU usage is higher than 70 %. This definition is not complete, and will be extended later in this section.

In order to *react* to the changes, a Rule is always connected with an appropriate *Action*. The action is executed by the Rule when the criteria defined by the Rule are met. There are multiple types of actions; most of them will be described later in this section.

The system provides two types of rules: Value Expression Rule and System Event Rule. The first one can be used to monitor the value of a selected capability, for instance to react if the CPU usage is higher than 90 %. It is possible to create a rule condition using Java Script language.

Another type of rules is the System Event Rule which allows to react on the changes occurring in the Monitoring System. For instance, it is possible to monitor whenever an agent is attached or detached to/from the system.

In order to react, AgeMon provides five different actions out of the box (it is also possible to define new actions). The first, simplest action is a *Console Action*. When this action is triggered, the message will be logged to a log file. A more complex action is *Email Action*. When it is triggered, an email message will be sent to a predefined email address.

The next action available in the AgeMon system provides the ability to run an external command. When an *ExecuteScript* action is defined, the user can provide the name of a script and path with the location of the script.

The next two actions are used to directly interact with the monitoring system. They allow for execution of a specified code from the *monitored system* – depending on the action's type, the processor will execute a static method or invoke a method on an MBean.

Typically, a rule is deployed in a single Rule Role. This is the most common situation, but AgeMon provides a second type of rule deployment called the *cooperative* mode which brings the *high availability* functionality. In this mode, the rule is deployed in multiple Rule Agents. Agents are able to synchronize the evaluation and execution of the rules, so the action will be executed only once. The following assumptions for the cooperative mode are listed below:

- All the Rule Roles Agents have a running instance of the Cooperative Rule (**CR**). *Running* means that the rule is executed against the monitoring data on each of the agents.
- When the rule condition is met, only ONE role executes a Cooperative Action (**CA**). The election algorithm will be used to select an agent which can perform a **CA** (the algorithm is similar to the one used in finding the substitute agent).
- If the **CA** fails due to the internal reasons (e.g. not because of the exceptions thrown from the monitored system) a next agent will try to re-execute a **CA**.
- All the roles have to have access to the monitoring data used in a definition of the condition. This means that whenever a rule is deployed, the additional monitoring links need to be created automatically.
- If a new rule agent joins the network, the **CR** will be automatically deployed on it and the corresponding monitoring links will be created.

3.5 Persistence Role

The Persistence Role is used to store the monitoring results in a persistent way. By default, this Role will use an in-memory database. Due to this reason, it is not needed to install any third-party applications in order to enable persistence.

3.6 CLI Role

The CLI Role is the second role used by the user to interact with the Monitoring System (and System Under Monitoring). This role is complementary to the GUI – it will provide additional capabilities for the advanced user, while some of the other features will only be available in the regular GUI. For instance, with the CLI it will not be possible to create a visualisation instance, whereas it is meant to enable creating customized queries used to export the monitoring data.

3.7 Communication in the System

Messaging is the key concept of the system. Agents interact with each other by exchanging the messages. The system uses a state-of-the-art communication library which is built by analogy to a stack. The bottom layer introduces a discovery of all the agents in the same group. Multicast as well as gossip server(s) can be used to detect members of a group. The low-level communication protocol is also implemented in this layer. All the messages are exchanged in a reliable way. The system allows to use point-to-point communication as well as point-to-many. In the second approach, a message is sent with the UDP/multicast protocol. Thanks to that messages are sent in single operation to all agents. This reduces the network usage in contrast to plain TCP protocol. At the same time, additional components built on top of protocol ensure reliability (e.g. failure detection, retransmissions etc.). Due to the discovery protocol, it is possible to start the system in a typical environment without any additional configuration.

The second layer of the communication library brings an agent abstraction which simplifies sending the messages to the agent. The agents' topology is reflected in an object-oriented fashion. An agent object contains methods to send messages as well as to subscribe for possible notifications.

The top layer defines the monitoring capabilities of the system. As mentioned above, the communication between agents is based on exchanging the messages: there are more than 40 types of them, gathered into two groups: foundation messages and regular messages.

The library is highly extensible, it is allowed to define new types of messages as well as corresponding processors. The library can also be used in other systems, since it is not specific to the system under discussion.

3.8 Logging in the System

Having a mechanism for correct logging of the events in the application is an important feature of any modern systems. It can be used to validate the state of the system as well as to track the issues that occurred during runtime.

AgeMon uses the log4j library, by default logging only important messages that are presented on the console. It is possible to enable a very detailed logging and redirect it to files. In order to reduce performance degradation when tracing is enabled, the logging system uses an asynchronous appender. It is also possible to rotate logs. Nevertheless in case a full, detailed logging is enabled, it may have a significant impact on the agent performance. During the conducted tests, when the TRACING level of the logger is enabled, CPU utilization was up to 50 % higher than when the default logging settings are used. Tracing is very detailed – it is possible to see the details of each message that is exchanged as well as all connections that are being made on the TCP/UDP level.

In addition to the regular logging, it is possible to easily integrate the system with more *centric* logging environments that can be used to profile the performance of the system. We have successfully used `logstash` (for pushing data), `elastic` (as data storage) and `kibana` (presentation & BI layer) to store and analyse AgeMon's performance.

4 HIGH AVAILABILITY AND SELF-HEALING

The AgeMon was designed as a self-healing, high availability monitoring system. It is used as a proof-of-concept to verify which techniques can be used to build self-healing monitoring systems. Below we outline the key-features implemented in AgeMon which enable high availability and self-healing.

4.1 Automatic Discovery

The automatic discovery of all the agents in the group has an impact on the high availability. It is not required to have any kind of central registry where all agents have to be registered – therefore there is no single point of failure. The automatic discovery can be also considered as one of the self-healing techniques. Agent, which is detached from a group due to a network issue, is able to automatically re-attach to the group when the issue is resolved.

4.2 Reliable Transport Protocols

The protocols used by the AgeMon system are *reliable*. This means that the transport layer will notify other layers that the message has been *or* has not been delivered successfully to the recipient. The successful delivery means that the recipient sent back the ACK information in order to confirm that the message had been received. The reliability of the protocol provides also additional features like preserving the correct order of messages or enforcing the coherency of messages.

4.3 Network Failures Tolerance

The AgeMon system can operate when there occurred a network failure. The first line of defence is the ability to cache the messages in the designated buffer on the agent side. If the monitoring results cannot be sent to the destination agent, the results will be stored locally and a second phase of resolving the problems will be triggered – finding a substitute agent (it will be described later in this section).

4.4 Lack of Single Point of Failure

The single point of failures is usually the biggest pain point which prevents the system from being highly available. There are two aspects of dealing with single points of failure:

- *avoid* the problem by introducing the correct design decisions. The Agent-based approach guarantees that the system is decentralized. All the agents are homogeneous, except the roles. On the other hand, the roles can be dynamically started in order to resolve the issues in the system.
- add *redundancy* for the components. Whenever it is not possible to avoid the problem, all important components should be duplicated.

4.5 Roles Redundancy

Where it is not possible to avoid the single point of failure, one of the solutions is to enhance redundancy in the system.

AgeMon is by its nature agent-based, and all the agents are the same – the only difference between the agents are the roles which they play. There are multiple ways of achieving the redundancy in the AgeMon System. The first one is quite simple and is presented in Figure 4.

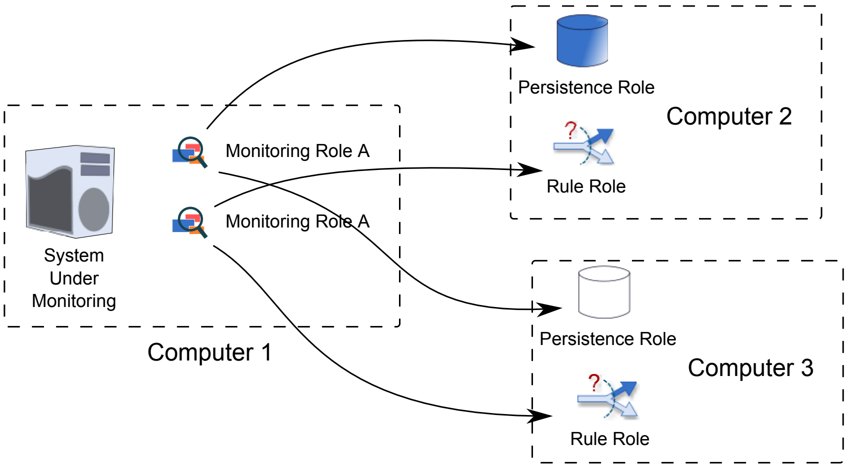


Figure 4. Redundancy of key components in AgeMon system

In this example, AgeMon is used to monitor the OS specific values. All the agents are duplicated – there are two monitoring agents, two persistence roles and two rule roles. The agents are running on different computers in case of a physical failure.

While AgeMon supports duplication of the agents, it has several disadvantages. The complexity of deployment is very high in case of multiple monitoring agents, and requires a lot of redundant connections. It requires a significantly higher network bandwidth. Therefore this type of deployment is not recommended and in most situations it is not needed.

AgeMon provides much more sophisticated solutions which enable high availability deployment – for instance it can automatically respond on changes in the system and add the redundancy 'on-the-fly' by electing the substitute agents. This approach is described in the next subsection.

4.6 Substitute Agents – Failover

The Agents Substitution is a key feature implemented in AgeMon, which enables high availability in the Monitoring System. It is used to elect the substitute agent that will persist the data in the database while the original destination agent is down. Figure 5 depicts an example usage.

In this example (Figure 5 a)), the user wants to monitor two OS capabilities: CPU Usage and Memory Usage. After some period of time *Agent 2* is killed (Fig-

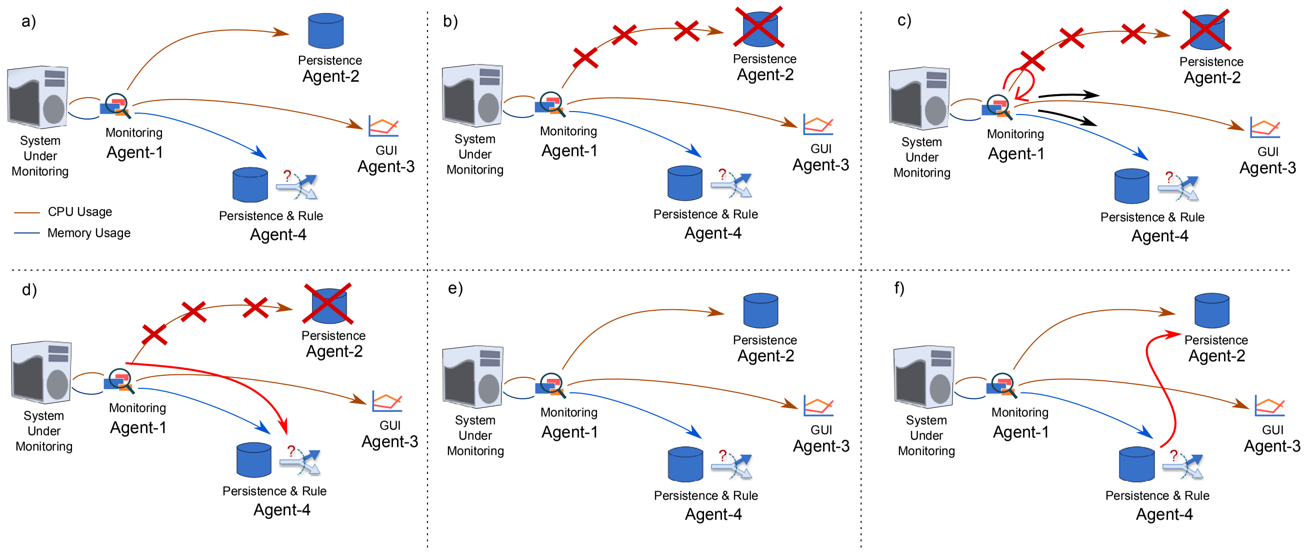


Figure 5. Overview of the election algorithm

ure 5 b)). The system detects such a situation and starts the failover procedure which consists of two steps.

In the first step (Figure 5 c)) the monitoring agent creates a *local buffer* where the monitoring data will be stored until the problem is resolved. This prevents from losing any monitoring data.

The second step is focused on finding the *substitute agent* that can handle the monitoring data. The *election algorithm* is used in order to fulfil this task (it is possible to use different election *policies*). The failover monitoring link will be automatically created and the monitoring data will be sent to the selected substitute agent (Figure 5 d)).

At the same time, the rule agent was configured to monitor the state of the monitoring agent. Since Agent 2 was killed, it sends an email to the administrator. After some time, Administrator was able to free some space, and restart the persistence agent (Figure 5 e)). The failover link will be deleted, and all the new monitoring data will be sent directly to Agent 2.

In the last step of the failover/substitution algorithm the monitoring data stored in the Substitute Agent (Agent-4) is transferred to the original destination agent (Figure 5 f)). After this step, Agent 2 contains the complete and coherent monitoring data.

The substitute agents are an important piece of the AgeMon system. Owing to the election algorithms, the system can provide high availability capabilities. It can also be considered as a self-healing strategy. While this technique does not lead to a complete system recovery, it helps with resolving the problem. The monitoring data will never be dropped if at least one agent can function.

4.7 Advanced Rules – Self-Healing

The rules can be triggered if the state of the Agent Group is changed, for instance, when one of the agents is down. This simple rule can be used to define complex actions and enable *self-healing* of the monitoring system. For instance, the system can detect that one of agents is detached. Based on the log entries, the system can decide that the agent is down due to the system restart, and try to start it again (e.g. by executing a remote command on the server).

This is one of examples – of course, since there is no limitation for external scripts, there are plenty of possible self-healing actions, starting with restarting the network adapter to solve connectivity issues between agents, to ending with removing the logs to gain free space. All these cases can be already allowed for AgeMon.

4.8 High Availability/Self-Healing Strategies – A Summary

The roles performed by the agents have different requirements from the high availability perspective. For instance, we can use redundancy to setup the Monitoring

Role. The Persistence Role can also be setup redundantly, but it will not be efficient from the disk usage perspective. A similar problem is when the Rule Role is concerned – the same rule can be executed on multiple agents in parallel, but the action can be triggered only on one agent.

Due to these reasons for each Role type, a different High Availability strategy should be used:

Monitoring Role – for this type of the role, the best strategy is to combine both agent *redundancy* and *self-healing* actions. The redundancy of the agent's instances will decrease the probability of failure. Even if such unexpected situation occurs, the self-healing rules and actions can restart the problematic agents.

Rule Role – the *cooperative* rules should be used for the high availability environment. These rules can be executed/evaluated in parallel. The AgeMon system will ensure that the action will be performed only by one instance of the agents.

Persistence Role – the approach with *Substitute Agents* is the best when this type of the role is considered. In case of any type of failure (agent failure, connection failure) the AgeMon system will elect the substitute agent which will be used to persist the data. This will prevent from losing any monitoring results.

GUI Role and CLI Role – these types of the roles do not need to have a separate high availability mechanism. They are not storing important data and do not have any automated processing which should be parallelized.

Due to the roles approach used in AgeMon it is possible to develop different high availability algorithms for different requirements. As is evident, each rule has a different nature of operating and requires a different treatment in order to enable high availability. Splitting the functionalities into the roles encapsulates them and provides a good stub for implementing specific high availability algorithms.

5 SYSTEM TESTS

The first part of this section that is dedicated to testing the AgeMon System is focused on High Availability. We will verify if the system can be used if some of its components fail. These types of tests are referred to as destructive tests. We will use them to validate if the self-healing techniques used in the AgeMon system are good enough to ensure High Availability.

In the second part of the section, performance tests are presented. These types of tests are used to determine the responsiveness and stability of the system under different values of load. The performance tests deliver information about the scalability or reliability of the system. They are followed by a description of the tests which were used to verify what is the minimum time in which the system can react to changes in the monitoring system.

5.1 Self-Healing Tests/High Availability Tests

In order to verify the High Availability of the AgeMon, two types of tests were executed. The first group consists of a set of destructive tests. These tests attempt to cause a failure in some components (e.g. random components, critical components) in order to verify the robustness and reliability of the system. The second type of the tests which were executed is the soak test. It is used to verify if the system works correctly under significant load over a significant period of time.

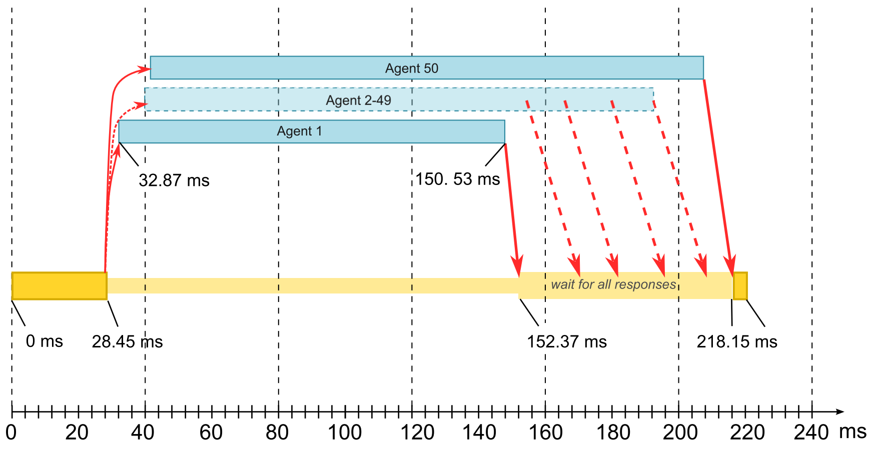


Figure 6. Process of the election of a substitute agent

In the **first test**, a system that consists of 50 agents was under observation. A half of the agents were executing monitoring scenarios, while others were used to persist results. The configuration of the agent allows all agents to perform a persistence role.

The test simulates a failure in the persistence agent. The self-healing mechanism implemented in the AgeMon should detect this situation and try to elect a substitute agent which can be used to persist monitoring results until the original agent is available.

The election procedure (Figure 6) starts when a monitoring agent is not able to send a message to a destination agent. Owing to the Agent Communication Layer, each agent possesses the list of the active agents. If the destination agent is not present, the election procedure is started (the election procedure is also started if there was a failure when sending the message). At the same time all the messages with monitoring results are stored by the agent in a local cache. The messages will not be lost as long as there is a free memory space. A single message typically needs 1–2 kB – it means that it is possible to keep half a million messages with a 1 GB heap size.

At the moment of 28 ms after the failure was detected, a request to start an election process is sent to all the agents in the group. The agent processes the request, and based on the election policy it responds with the information which will be used to elect an agent from the group. The monitoring agents are waiting for all responses (or until a predefined time elapses). An average response time was between 120 and 180 milliseconds. After all the data is collected, the agent elects a substitute agent and advertises this information to all agents. All messages cached in the monitoring agent are sent to the elected agent.

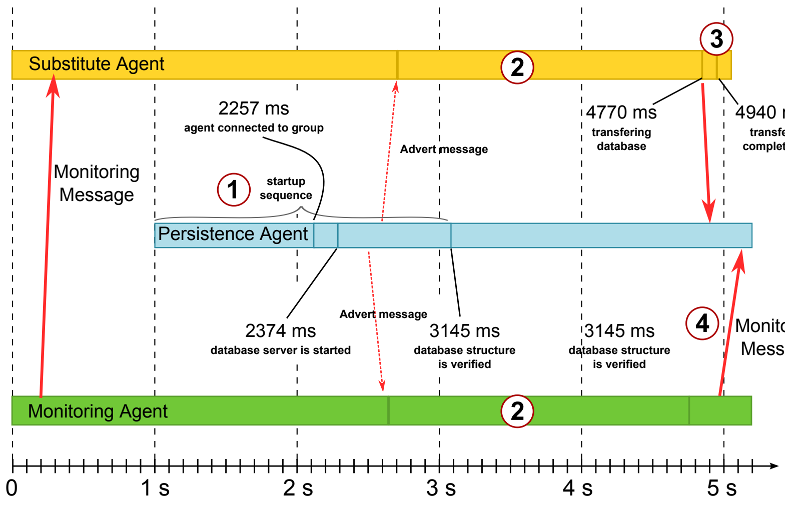


Figure 7. Operations performed after Persistence Agent is recovered

The **second test** scenario covers a situation when Monitoring Agent failed (e.g. due to the machine restart) and needs to be restarted. It presents the capability of the monitoring system to auto-detect failures in the agent group and the flexibility of a rule agent to execute a customized script.

For this scenario (Figure 7), a group of 20 agents is prepared. A half of the agents are performing monitoring, 9 ones are persisting data and one is used as a rule agent. There is one rule deployed in the system – this rule will execute an external script in case one of the agent is detached. The script is used to connect to the remote sever where the Monitoring Agent is installed and will try to execute the restart script.

During the test, Monitoring Agent was killed with `kill -9` command. After about 100 ms, Rule Agent was notified by the Transport Layer about a missing agent. During the next 140 ms the agent was executing a bash script. Monitoring Agent was started after additional 50 ms. Therefore, the total time when the agent was down is **295 ms**. After additional 170 ms all agent's services were started again, and the agent was reconnected to the group.

5.2 Reliability

The total number of the messages sent during all tests is higher than 40 million. There were different tests conducted – we tested the system under regular load, then we performed a stress test of the system. After that the destructive tests were conducted followed by the soak test.

In all these tests, we verified the reliability of the system (together with testing other non-functional requirements like performance or scalability). These tests are very promising – there was no single monitoring message missing. The system could work with reduced performance (e.g. when load was too high) but we did not observe any dropping messages. The tests evidence that transport layer is reliable – all messages were delivered. In case there was a problem with network, the message was not acknowledged – and it was retransmitted again.

There are only two cases when isolated messages can be lost. The first one can occur when there is a failure in a monitoring agent before the message was broadcasted to the other agents. The second situation happens when there is a failure of the persistence agent during message processing (but before it is persisted in the database). The theoretical solution for this problem assumes that the persistence agent should acknowledge to the monitoring agent the fact that the message is persisted. If there is no ACK message, the monitoring agent can try to:

1. retransmit the message, or
2. start an election procedure to find a substitute agent.

The drawback of both solutions is that it could reduce the overall performance of the system (additional synchronization in the monitoring agent, additional message in the communication layer). Therefore, we did not decide to implement it.

Fortunately, the system architecture allows avoiding both situations. For instance, to avoid losing the messages in the monitoring agent we proposed duplication of those agents. The same idea applies to the persistence agents. Of course this type of the deployment should only be considered when persisting all messages is critical. The conducted tests have proved that losing a message is very unlikely to happen.

There was no single message lost in all the tests – all messages (10 M during destructive tests and 32 M during soak tests) were successfully delivered.

5.3 Performance Tests

To test the performance of AgeMon, a sample testing environment was prepared. In this setup, there are multiple monitoring agents (the exact number depends on the specific test scenario) installed on multiple-machines. They are monitoring CPU usage. The monitoring data is sent to one persistence agent – the interval between each send operation (we call *delay*) is defined in a scenario.

By measuring the total time of the test and comparing it with the expected time, it will be possible to calculate overheads. If the overheads are high (cannot be explained by network delays or serialization/deserialization of the messages) it means that the persistence agent was not able to process so many requests.

In all the test scenarios the following servers/computers were used (connected with Gigabit Ethernet):

- 5 × SunOS 5.10, 48 GB of memory, 2 × Intel® Xeon® CPU X5650 @ 2.67 GHz (later referred to as Sun),
- 5 × Linux RedHat 5.5 (Tikanga), 30 GB of memory, 24 × Intel® Xeon® CPU X5650 @ 2.67 GHz (later referred to as Linux),
- 1 × Windows 7 Enterprise, 16 GB of memory, 1 × Intel® Core™CPU i5-3527U @ 2.30 GHz (4 cores) (later referred to as Win7).

During the test some principal performance metrics were gathered like CPU Usage, Memory Usage, Threads Count, and Garbage Collector executions. The system was monitored with a second instance of the AgeMon system running on different ports with a different agent group. This second 'monitoring' system instance was built with multiple monitoring agents and one persistence agent. The monitoring data was extracted through a GUI agent.

Some 12 tests with different numbers of agents and messages were performed in order to measure the differences between 'ideal processing' and the observed performance. We assume here that more than 20 % difference between 'ideal processing' and the actual performance should be considered unacceptable.

The results were confronted with the 'ideal processing'. The ideal processing was calculated based on the number of messages and the delay between sending the messages. It does not include any network delays or time used by the agent to persist the data. Therefore, the ideal processing cannot be reached in a live system. On the other hand, in a high performance system, the results should be close to the ideal ones.

In order to verify if the performance of the Persistence Agent is correlated with the number of agents, different types of configurations were used. Figure 8 presents the results of the tests. The figure depicts the differences between the ideal situation and the observed results (the difference is marked by pink and red colour). We can draw multiple conclusions:

- The performance is directly related to the number of messages to be processed. A huge amount of messages results in longer processing delays of each message.
- Persistence Agent can process 950 messages with acceptable performance. Tests with more than 950 messages show a performance degradation (the difference between 'ideal processing' and the observed performance was more than 20 %).

If AgeMon is used in an environment with a large number of messages (greater than 1 000 messages per second), the performance may be unacceptable. In order to resolve this issue, the user has two possible ways: multiply the number of the persistence agents or deploy a persistence agent in the environment with a more powerful CPU.

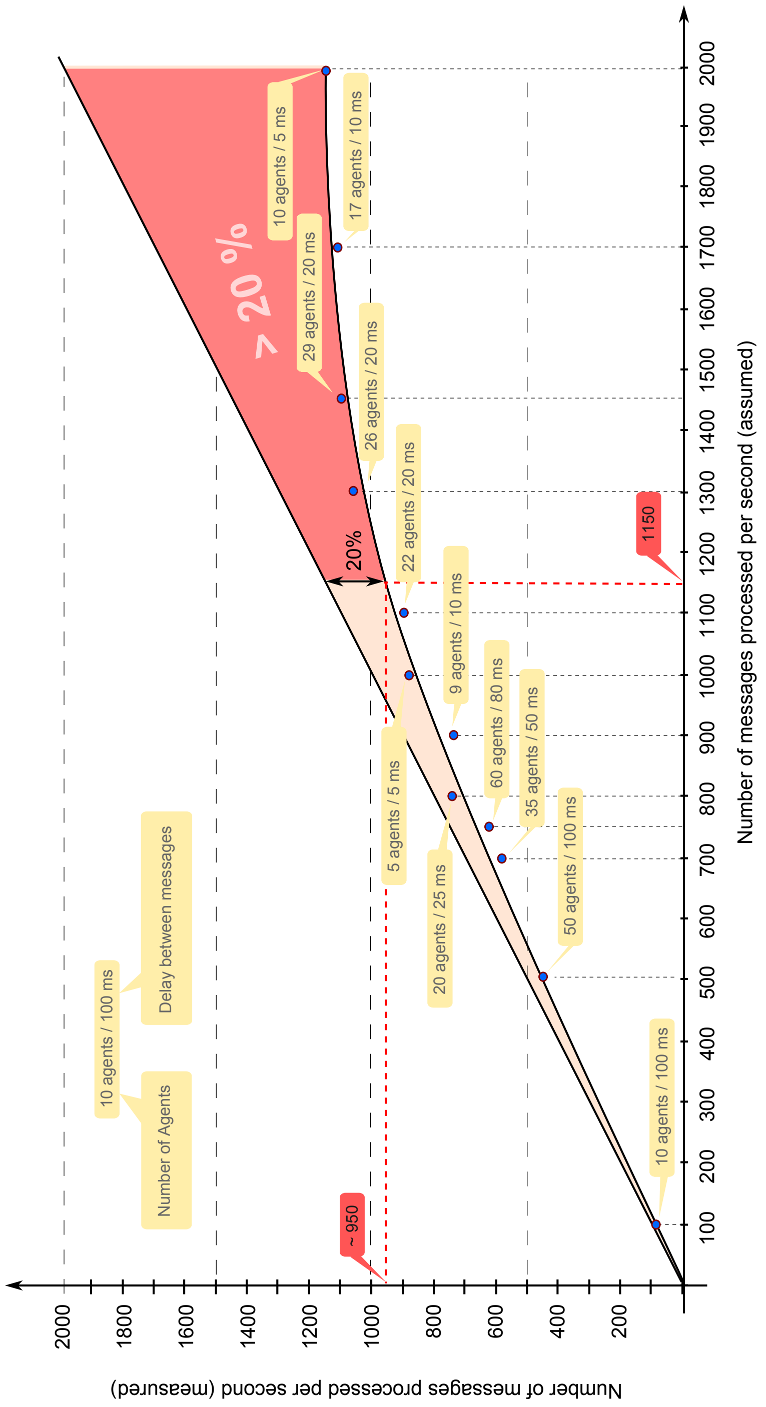


Figure 8. Performance of Persistence Agent

The tests results show that the system can be used to persist a big number of messages per second. One persistence agent can successfully process over 1 000 messages per second – it should be enough for most of the monitoring cases.

In order to verify the *scalability* of AgeMon, some additional tests were performed. In this test scenario a large number of agents were used. The first cycle of the test started with one persistence agent and 20 monitoring agents. The monitoring agents send messages to the persistence agent every 25 ms. In the next cycle the number of persistence agents and monitoring agents is increased. For each cycle, the performance of the system is measured. The results are given in Figure 9.

The AgeMon system can scale horizontally – as described above, the CPU could be regarded as the biggest bottleneck. Therefore, in order to increase the number of messages that can be processed by the system, the user may consider adding new agents running on dedicated physical machines.

System can scale to a greater extent than that presented in Figure 9, but due to the resource limitations we were not able to conduct more extensive tests. In order to estimate if the system can be used in a deployment scenario with a significantly greater number of agents instantiated we measured the overhead of the messages that were not used to exchange the monitoring data.

From the high-level perspective, the agents exchange two types of messages. The first group consists of messages used to transfer monitoring data between agents (in a typical case among the monitoring agents and persistence agents). Another type of messages is used to synchronize, detect failures and keep up-to-date the group of agents. These messages are vital to keep the group of the agents together within a single monitoring system.

Fortunately, the overhead generated by these messages is almost negligible. In our tests, these messages were responsible for 3 % of the network traffic, and 8 % of CPU. Based on that, we can draw a conclusion that the system should be able to handle at least 12 times more agents (up to 1 200) in a single group compared to the maximum number of agents used in the tests.

5.4 Latency in the System

Below we discuss two test scenarios which were executed to measure the latencies in the AgeMon system. The first one is quite interesting – it answers the question: "How long it takes the system to make a decision based on the state of the System Under Monitoring?" The second test is important as well, since it is used to check when the monitoring data is persisted.

The average decision time (Avg_{dt}) is one of the most important aspects of AgeMon. It defines the time which is required to take a decision. This factor describes what the maximum speed of changes in the monitored system is, which can be successfully detected and allows the system to make a correct, healing decision.

If the monitored system is faster than Avg_{dt}, the monitoring system may not be able to make a correct decision. An action made by the system could not be accurate to the prevailing conditions. In the test scenarios the observed decision in the system was performed **36 milliseconds** after a change in the measured capability had been observed.

In the next scenario, the persistence time was measured. The goal was to measure the average time between receiving the monitoring value and storing this information in the monitoring agent. The configuration in this scenario is similar to what was introduced in the previous test. The only difference is that a persistence agent is substituted for the rule agent. In this scenario the time needed to commit will be measured.

The tests show that the Monitoring Agent requires 33 milliseconds to send a message to the Persistence Agent. After additional 11 milliseconds the message is committed to the database by the Persistence Agent. The full process took 47 milliseconds.

The system can operate with millisecond-long latencies. The system requires 36 ms to make a decision and 47 ms to commit a result into the database. Prior to defining the rules, network tests need to be performed (e.g. with the `PING` command) in order to measure the latency.

The biggest contribution to the latency is caused by the message serialization. It is planned to be improved in the next versions of the system (the message can be constructed in a binary form instead of the use of the standard Java Serialization mechanism).

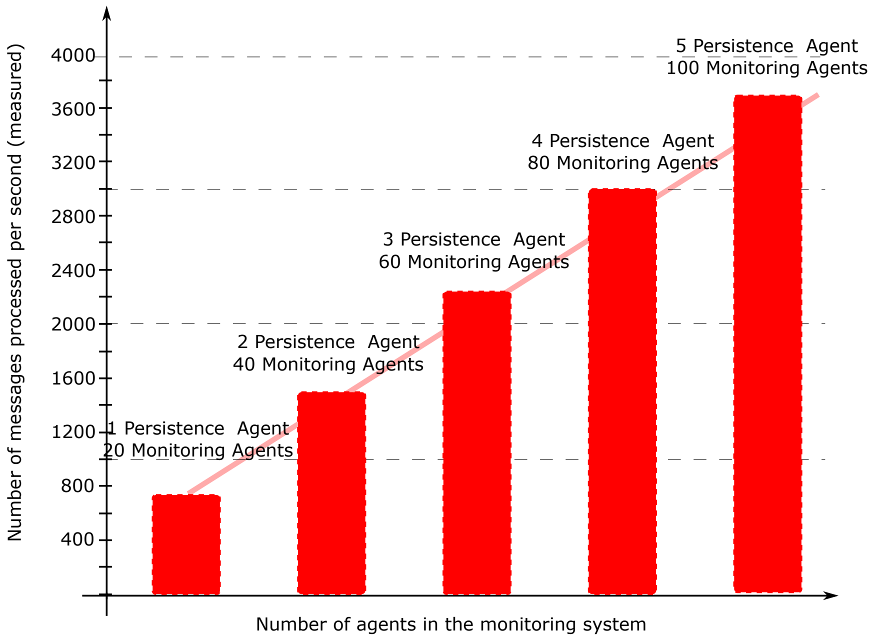


Figure 9. Scalability in the AgeMon system

6 CONCLUSIONS

The main goal for our research was to design a model and evaluate the algorithms and techniques which can be used to build a highly available monitoring system. In the paper a set of approaches has been described. Our work focuses also on the practical verification of the requirements. In order to do this, a new monitoring system called AgeMon was developed. This system uses an agent based, distributed approach. Each agent can perform different roles in the system. A role groups similar functionalities.

Role is an important concept to build a *highly available*, *self-healing system*, since different functionalities need different strategies to solve the problems related to high availability. The Persistence Role in the system uses substitute agents to persist data when one of the agents fails. The Rule Role can cooperate with other roles to provide a reliable way of executing the actions. The Monitoring role can be dynamically restarted in case of a failure.

The model lacks any single point of failure. The communication between agents is based on a reliable and fast protocol and provides auto-discovery. Due to it, there is no need to have a gossip/rendezvous server. Each component in the system could be redundant, but it is not a "must".

The system is a proof of concept for a more generic model. This model can be applied to build high availability systems (it does not apply only to the monitoring systems). Such concepts like roles, communication layer, and agent abstraction can be successfully applied in other systems.

The *monitoring mechanism* used by the AgeMon is very flexible. By default it can be used to monitor OS and any Java application. If such an application uses JMX to provide monitoring information it can be automatically used by the AgeMon. For any other type of monitoring, an appropriate adapter needs to be developed. In order to implement such an adapter, a single interface needs to be implemented.

An important part of the monitoring system is providing **visualisation** of the results. Charts in AgeMon support a single or multiple sources. Owing to this it is possible to compare two different measurements (e.g. CPU or memory) on a single chart. It is possible to zoom, scroll and print charts.

The system supports multiple types of *rules* and *actions.* AgeMon provides built-in rules like "agent attached", "agent detached" and rules based on the monitored value. It is also possible to create a custom rule by providing an implementation of a particular interface in Java. Similarly, some number of built-in actions is available (console, email, execute script, execute static call) while other actions can be implemented by the user.

The system is fully *distributed.* The system supports different types of deployment – for instance it is possible that a large number of agents are installed on a single computer. It is also possible to have one agent per physical machine. Agents can be distributed across a local network. The distribution over WAN or Internet is also supported. In this case the tunnelling and gossip servers need to be used.

The Persistence Role provides the *persistency* of monitoring results. AgeMon provides an in-memory, relational database out of the box (HSQL). If needed, other SQL or NoSQL databases [31] can be used.

The system can be *extended* and *integrated* with the monitored application or other monitoring systems in a straightforward way.

The use of AgeMon is very facile. It does not require any configuration or sophisticated *deployment*. The system is built inside one file (a jar file) and no installation is required. In the default configuration (default agent group, in memory database) no changes to the configuration is needed – the user can start agents which will automatically discover themselves and build the monitoring system.

To summarize, it is possible to build a high-available monitoring system. The designed self-healing approach can be very helpful in achieving this goal.

Novel concepts introduced. The AgeMon system is one of the first designed and implemented monitoring systems with self-healing capabilities. It is also a highly available system; it does not have a single point of failure and supports the redundancy of all its components.

The model used to build the system, which is based on roles, can be used to build other self-healing systems. This approach can be applied to other types of applications, so it is not limited only to the monitoring systems. The self-healing strategies defined in the model can be successfully used for other types of the components.

One of the most important achievements of our work is the evaluation of a number of HA and self-healing strategies. Different problems require different solutions, the role-based approach helps with identifying similar functionalities with the same healing strategies.

Additionally, the communication model designed for the system can be adopted and used in many other applications with distributed components. It supports auto-discovery, reliable and efficient communication. The model is built with three different layers – a physical communication layer, an agent layer, and a monitoring layer. The monitoring layer can be easily adapted to support other types of messages, specific to a system under design.

Acknowledgment

The Author is indebted to several people, first of all to Dr. Piotr Pęgiel for valuable help and to Mr. Paweł Koperek for inspiring discussions. The research presented in this paper received financial support from AGH University of Science and Technology Statutory Project.

REFERENCES

[1] Perry, J. S.: Java Management Extensions: Managing Java Applications with JMX. O'Reilly, 2002. ISBN 0-596-00245-9.

[2] Fleury, M.—Lindfors, J.: JMX: Managing J2EE with Java Management Extensions. Sams Publishing, 2002. ISBN 0-672-32288-9.

[3] Laprie, J. C. (Ed.): Dependability. Basic Concepts and Terminology. Springer-Verlag, New York, 1992, doi: 10.1007/978-3-7091-9170-5.

[4] Gray, J.: Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7., Tandem Computers, 1985, pp. 17–19.

[5] Rentschler, M.—Kehrer, S.—Zangl, C. P.: System Self Diagnosis for Industrial Devices. Proceedings of 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2013), September 10–13, 2013, doi: 10.1109/ETFA.2013.6648019.

[6] The Complex Adaptive Systems (CAS) Group: Multi-Agent System. `http://wiki.cas-group.net/index.php?title=Multi-Agent_System`, 2011.

[7] The Intelligent Software Agents Lab. `http://www.cs.cmu.edu/softagents/intro.html`.

[8] Ghosh, D.—Sharman, R.—Rao, H. R.—Upadhyaya, S.: Self-Healing Systems – Survey and Synthesis. Decision Support Systems, Vol. 42, 2007, pp. 2164–2185, doi: 10.1016/j.dss.2006.06.011.

[9] George, S.—Evans, D.—Marchette, S.: A Biological Programming Model for Self-Healing. First ACM Workshop on Survivable and Self-Regenerative Systems (SSRS '03), 2003, pp. 72–81, doi: 10.1145/1036921.1036929.

[10] Aldrich, J.—Sazawal, V.—Chambers, C.—Nokin, D.: Architecture-Centric Programming for Adaptive Systems. Proceedings of 1st Workshop on Self-Healing Systems (WOSS '02), 2002, pp. 93–95, doi: 10.1145/582128.582146.

[11] Dabrowski, C.—Mills, K. L.: Understanding Self-Healing in Service Discovery Systems. Proceedings 1st Workshop on Self-Healing Systems (WOSS '02), 2002, pp. 15–20, doi: 10.1145/582128.582132.

[12] IBM. The IBM Autonomic Computing Initiative: `http://www-03.ibm.com/systems/z/os/zos/features/sysmgmt/autonomic/index.html`.

[13] Raz, O.—Koopman, P.—Shaw, M.: Enabling Automatic Adaptation in Systems with Under-Specific Elements. Proceedings 1st Workshop on Self-Healing Systems (WOSS '02), 2002, pp. 55–60, doi: 10.1145/582128.582139.

[14] Blair, G. S.—Coulson, G.—Blair, L.—Duran-Limon, H.—Grace, P.—Moreira, R.—Parlavantzas, N.: Reflection, Self-Awareness and Self-Healing in OpenORB. Proceedings 1st Workshop on Self-Healing Systems (WOSS '02), 2002, pp. 9–14, doi: 10.1145/582128.582131.

[15] Georgiadis, I.—Magee, J.—Kramer, J.: Self-Organizing Software Architectures for Distributed Systems. Proceedings 1st Workshop on Self-Healing Systems (WOSS '02), 2002, pp. 33–38, doi: 10.1145/582128.582135.

[16] de Lemos, R.—Fiadeiro, J. L.: An Architectural Support for Selfadaptive Software for Treating Faults. Proceedings 1st Workshop on Self-Healing Systems (WOSS '02), 2002, pp. 39–42, doi: 10.1145/582128.582136.

[17] Weygant, P. S.: Clusters for High Availability: A Primer of HP Solutions. Prentice Hall, 336 pp., 2001. ISBN 978-0-13089-355-0.

[18] Richards, M.: The Secret to Bulding Highly Available Systems. No Fluff Just Stuff, Vol. 2, 2011, No. 5. `http://wmrichards.com/ha.pdf`.

[19] Goel, A. L.: Software Reliability Models: Assumptions, Limitations, and Applicability. IEEE Transactions on Software Engineering, Vol. 12, 1985, pp. 1411–1423, doi: 10.1109/TSE.1985.232177.

[20] Longbottom, C.: Make a High-Performance Computing and High-Availability Datacentre. Computer Weekly, 2013. `http://www.computerweekly.com/feature/Making-your-datacentre-fit-for-high-performance-computing-and-high-availability`.

[21] Oracle: Java Management Extensions (JMX) Technology. `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

[22] Funika, W.—Godowski, P.—Pegiel, P.— Król, D.: Semantic-Oriented Performance Monitoring of Distributed Applications. Computing and Informatics, Vol. 31, 2012, No. 2, pp. 427–446.

[23] Nagios Web Site: `https://www.nagios.org/`.

[24] Ganglia Web Site: `http://ganglia.sourceforge.net/`.

[25] Shehory, O.—Martinez, J.—Andrzejak, A.—Cappiello, C.—Funika, W.—Kondo D.—Mariani, L.—Satzger, B.—Schmid, M.: Self-Healing and Recovery Methods and Their Classification. In: Andrzejak, A., Geihs, K., Shehory, O., Wilkes, J. (Eds.): Proceedings Dagstuhl Seminar 09201 "Self-Healing and Self-Adaptive Systems", May 10–15, 2009, Dagstuhl, Germany, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 2009. ISSN 1862-4405.

[26] Ribler, R. L.—Vetter, J. S.—Simitci, H.—Reed, D. A.: Autopilot: Adaptive Control of Distributed Applications. Proceedings of the Seventh International Symposium on High Performance Distributed Computing, July 1998, doi: 10.1109/HPDC.1998.709970.

[27] Baliś, B.—Bubak, M..—Łabno, B.: GEMINI: Generic Monitoring Infrastructure for Grid Resources and Applications. Proceedings of the Cracow '06 Grid Workshop "The Knowledge-Based Workflow System for Grid Applications", ACC Cyfronet AGH, Krakow, 2007, pp. 60–73.

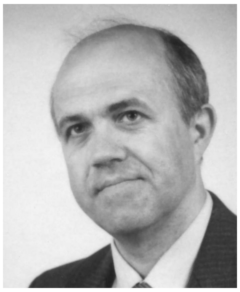
[28] Fahringer, T.—Seragiotto, C.: Automatic Search for Performance Problems in Parallel and Distributed Programs by Using Multi-Experiment Analysis. In: Sahni, S., Prasanna, V. K., Shukla, U. (Eds): High Performance Computing – HiPC 2002. Springer Verlag, Lecture Notes in Computer Science, Vol. 2552, 2002, pp. 151–162.

[29] Fahringer, T.—Jugravu, A.—Pllana, S.—Prodan, R.—Seragiotto, C. Jr.—Truong, H.-L.: ASKALON: A Tool Set for Cluster and Grid Computing. Concurrency and Computation: Practice and Experience, Vol. 17, 2005, No. 2-4, pp. 143–169, Wiley, Inc., 2005.

[30] Fahringer, T.—Seragiotto, C.: Performance Analysis for Distributed and Parallel Java Programs. IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005) 2005, `http://dps.uibk.ac.at/local/conferences/ccgrid/2005/pdfs/115.pdf`.

[31] Gąsiorowska, K.—Trybała, D.—Funika, W.: Storage of Monitoring Data in NoSQL Database. Procedia Cracow Grid Workshop '16, Krakow, ACC CYFRONET AGH, Krakow, 2016, pp. 39–40. ISBN: 978-83-61433-20-0.

[32] Żmuda, D.—Psiuk, M.—Zieliński, K.: Dynamic Monitoring Framework for the SOA Execution Environment. Proceedings of International Conference on Computational Science (ICCS 2010). Procedia Computer Science, Vol. 1, 2010, No. 1, pp. 125–133.



Włodzimierz Funika works at the Institute of Computer Science of the AGH University of Science and Technology in Krakow (Poland). His main research interests are in distributed programming, tools construction, performance analysis and visualization, machine learning. Involved in EU Cross-Grid, Core-GRID, K-WfGrid, ViroLab, GREDIA, UrbanFlood, MAPPER, VPH-Share projects as well as in Polish-wide projects PL-Grid, PLGrid PLUS, PLGrid Core, and PLGrid NG.