PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL

FACULTY OF INFORMATICS

GRADUATE PROGRAM IN COMPUTER SCIENCE

Fault Supervision for Multi Robotics Systems

Felipe de Fraga Roman

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at Pontifícia Universidade Católica do Rio Grande do Sul.

Advisor: Prof. Dr. Alexandre de Morais Amory

Porto Alegre

2015

Dados Internacionais de Catalogação na Publicação (CIP)

R758f Roman, Felipe de Fraga

Fault supervision for multi robotics systems / Felipe de Fraga Roman. – Porto Alegre, 2015.

79 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Dr. Alexandre de Morais Amory.

1. Informática. 2. Robótica. 4. Tolerância a Falhas (Informática). I. Amory, Alexandre de Morais. II. Título.

CDD 004.36

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada *"Fault Supervision for Multi Robotics Systems"* apresentada por Felipe de Fraga Roman como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 25/03/2015 pela Comissão Examinadora:

Prof. Dr. Alexandre de Morais Amory – Orientador — PPGCC/PUCRS

Prof. Dr. Felipe Rech Meneguzzi – PPGCC/PUCRS

Prof. Dr. Aurélio Tergolina Salton – PPGEE/PUCRS

Homologada em...../...../....., conforme Ata No. ..... pela Comissão Coordenadora.

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320–3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

FAULT SUPERVISION FOR MULTI ROBOTICS SYSTEMS

ABSTRACT

As robotics becomes more common and people start to use it in routine tasks, dependability becomes more and more relevant to create trustworthy solutions. A commonly used approach to provide reliability and availability is the use of multi robots instead of a single robot. However, in case of a large teams of robots (tens or more), determining the system status can be a challenge.

This work presents a runtime monitoring solution for Multi Robotic Systems. It integrates Nagios IT Monitoring tool and ROS robotic middleware. One of the potential advantages of this approach is that the use of a consolidated IT infrastructure tool enables the reuse of several relevant features developed to monitor large datacenters. Another important advantage of that this solution does not require additional software at the robot side.

The experimental results demonstrate that the proposed monitoring system has a small performance impact on the robot and the monitoring server can easily support hundreds or even thousands of monitored robots.

Keywords

Multi robot system, fault monitoring.

SUPERVISÃO DE FALHAS PARA SISTEMAS MULTI-ROBÔS

RESUMO

À medida que a robótica se torna mais comum e as pessoas começam a utilizá-la em suas tarefas de rotina, dependabilidade torna-se cada vez mais importante para a construção de uma solução digna de confiança. Uma abordagem comum de prover confiabilidade e disponibilidade é o uso de multi robôs ao invés de um único robô devido a sua redundância intrísica. Entretanto, no caso de um grande time de robôs (dezenas ou mais), uma tarefa aparentemente simples como a determinação do status do sistema pode se tornar um desafio.

Este trabalho apresenta uma ferramenta de monitoramento de sistemas multi robôs em tempo de execução. Esta solução integra a ferramenta de monitoramento de TI Nagios com o *middleware* robótico ROS sem a necessidade de instalação de software adicional no robô. O uso de uma ferramenta de TI consolidada permite o reuso de diversas funcionalidades relevantes já empregadas amplamente no monitoramento de datacenters.

Os resultados experimentais demonstram que a solução proposta tem um baixo impacto no desempenho do robô e o servidor de monitoramento pode facilmente monitorar centenas ou até milhares de robôs ao mesmo tempo.

Palavras-chave:

Sistemas multi robôs, monitoramento de falhas.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTING

LIST OF ABBREVIATIONS AND ACRONYMS

API – Application Protocol Interface

CLI - Command Line Interface

DCIM - Data Center Infrastructure Management

GUI – Graphic User Interface

HTTP - Hypertext Transfer Protocol

MRS - Multiple Robots Systems

NRPE - Nagios Remote Plug-in Executor

PUCRS – Pontifical Catholic University of Rio Grande do Sul

ROS - Robot Operating System

SNMP - Simple Network Management Protocol

SRS - Single-Robot Systems

SSH – Secure Shell

TCP/IP - Transmission Control Protocol and Internet Protocol

XDR - External Data Representation

XML - Extensible Markup Language

XML-RPC - Remote Procedure Call protocol which uses XML

YAML - is a recursive acronym for "YAML Ain't Markup Language"

TABLE OF CONTENTS

1 INTRODUCTION

Robotics is becoming commonplace and people start to use more mobile robots to help and to accomplish a variety of ordinary tasks such as vacuum cleaning, pool cleaning, and lawn mowing, among others. In addition, robots are also typically used to execute dangerous tasks, unhealthy tasks, go to remote places, among other possibly critical applications [GDU2010].

In this work robots are classified in two different types: stationary robots and mobile robots. Stationary robots are simpler than mobile ones because they are fixed at some controlled environment. It is common to use the stationary robots on industry to automate repetitive tasks. Also, this kind of robot is built for a very specific application. Typical applications of stationary industrial robots include casting, painting, welding, assembly, materials handling, product inspection, and testing. All these tasks can be performed with more accuracy and speed compared to humans.

Mobile robotics, on the other hand, is the research area that studies the control of autonomous or semi-autonomous vehicles [GDU2010] and [RSI2004]. Currently, there are some commercial applications for mobile service robots, such as goods transportation, surveillance, inspection, cleaning, and household tasks. Robotics has been evolving fast in terms of new functionalities and becoming affordable, increasing its use in several aspects of society [PAR2010]. This fact increased the development rate of new and more complex robotic applications [PAR2010], which require more complex software stack [ABA2008]. Despite these improvements, autonomous mobile robots have not yet made much impact upon industrial and domestic applications, mainly due to the lack of dependability, robustness, reliability and flexibility in real environments. This requires more research to enable the design of more efficient and robust robotic applications.

According to [LEP2008] one cost-effective way to provide effectiveness and robustness to robotic system is to use multi-robots instead of a single robot. Due to the application of parallelism, multi-robot systems (MRS) have some advantages over single-robots systems, such as greater task completion speed. Furthermore, improvements on robustness and reliability can be achieved in MRS through the implementation of fault-tolerant systems. For instance, when one member of the team fails, another can take over his work and continue that the task. An MRS of cheaper and simpler robots can typically

provide more reliability than a more expensive and complex single robot [LEP2008]. On the other hand, MRS also present more complex challenges compared to single robot systems. Because they are collective systems, MRS are more complex to manage and coordinate, since they require increased communication capabilities in order to coordinate all robots. Collectivity is also a factor on troubleshooting, because it is harder to determine the global state of the system.

MRS can be classified as homogeneous or heterogeneous [SVE2005]. Homogeneous MRS are systems where all robots have the same specification (hardware and software configuration). Heterogeneous MRS can have different kind of robots in the same system. In Homogeneous MRS it is easier to replace faulty robots. On the other hand heterogeneous MRS can employ different kind of specialized robots to perform different tasks.

Robotic systems can also be classified according to their autonomy level, i.e. its ability to decide how to accomplish a task based on its perception of the environment *[RHB2007]*. There are robots with no autonomy at all, called tele-operated, and semi-autonomous robot. A robot with some level of autonomy can be called an agent. Mobile robotics will become commonplace in the society if it can be cost-effective and dependable. Currently the cost-effectiveness of robotics is evolving since computers and electronics are more accessible. On the other hand, current single mobile robots lack effectiveness and dependability. MRS is naturally more robust than single robots due its intrinsic redundancy, but it increases the software complexity due to its distributed nature.

According to *[LEP2012]* even MRS designed to be robust will face unexpected faults from a very large range of possibilities. Detecting the sources of faults is the very first step towards a fault tolerant MRS. The large number of robots, the large number of possible faults in each robot, and a dynamic environment make the fault monitoring a complex and mandatory task for MRS with reliability constraints.

The *goal of this work* is to provide means to easily monitor faults at a team of heterogeneous robotic agents. The contribution of this work is a proof of concept that traditional infrastructure networking monitoring tool can also be used combined with robotics. Detect and isolate the defective robot is a first step to achieve robustness, toward an adaptive MRS that can execute the desired tasks even with the presence of faults. With more dependable robotic systems, more applications can be created to serve the society.

In order to achieve the proposed goal, the present work describes the integration between a traditional infrastructure networking monitoring tool and a robotics middleware. The advantage of combining pre-existing consolidated tools include a reduced development cost/time and the possibility to leverage desirable characteristics of well established software, such as network scalability, software stability, and software extensibility.

This work is organized as follows. The section 2 presents theoretical background regarding MRS, autonomous agents, and dependability. Section 3 describes the state of the art in terms of individual robot fault detection and MRS fault detection. Section 4 presents the developed architecture and the section 5 describes the implementation of the architecture and the experiments performed with it. Section 6 discusses the results obtained with the experiments. Section 7 concludes of this research.

2 THEORETICAL BACKGROUND

This section presents a theoretical background of the main concepts used in this research plan.

2.1 Autonomous Agents

Functional programs or traditional software work basically receive an input, process data and produces some output based on the received input *[RHB2007]*. However there are other kinds of programs that do not work on this traditional approach. This different kind of software maintains an ongoing interaction with their environment, they do not compute some function based on the input and return an output. Some example of these programs includes computer operational systems, process control systems and others. Even more complex software that these two previously approaches are the systems called agents system, an agent is a reactive system that contains autonomy in order to take actions determined by himself to accomplish their goals. These different systems are called agents because these systems are active, they are able to figure out one plan to actively pursue their goals. *[RHB2007]*.

2.1.1 Characteristics of Agents

Agents are systems situated in some environment. Some typical examples are the system stock exchange agents, these systems are developed to observe the stock market and, based on this information, take actions. The agent has the capability to perceive its environment through its sensors and it is able to cause some effects on the environment via its actuators as illustrated in Figure 1.

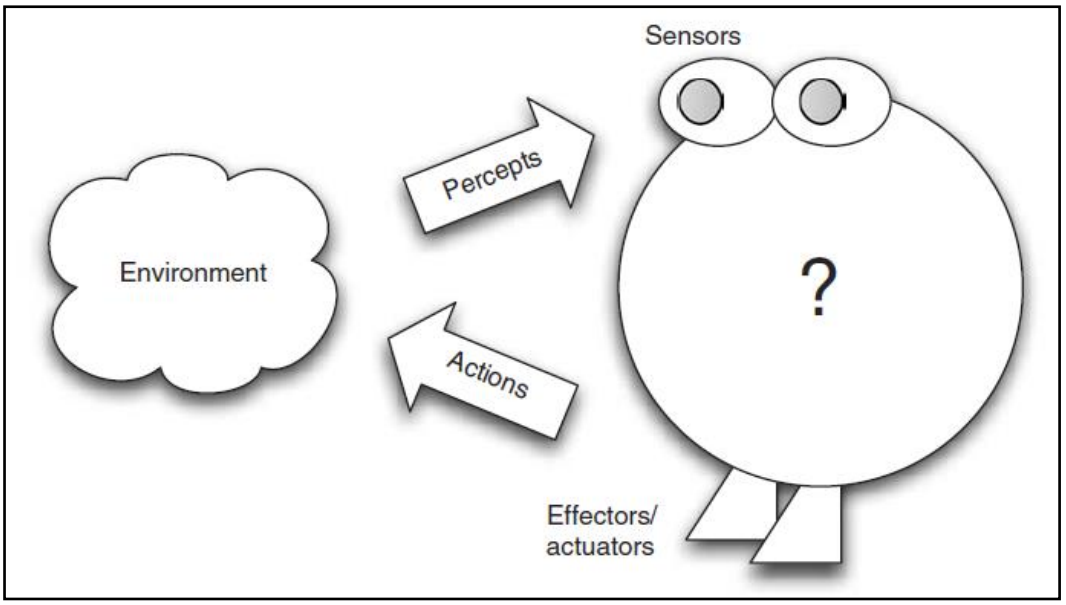


Figure 1 - Agent interaction with the environment *[RHB2007]*

According to *[RHB2007]* the environment occupied by an agent could be either physical or virtual (in case of software/simulation environment). For software agent works for virtual environment and robotics works for physical environment. The agents can take actions that will affect the environment, but they cannot completely control it. For example, a lawn-mower robot may not be able to finish its work because of obstacles on the ground. The real environment is dynamic and cannot be controlled so even the highly tested robots will face some unforeseen situations and fail.

Important characteristics of agents include [RHB2007]:

- Autonomy: the capability to operate independently at some level. Agents must be able to formulate a plan and execute it in order to achieve a goal.
- Proactiveness: the ability to exhibit goal-directed behaviour. An agent should actively try to initiate work that will lead it to achieve its goals. It should need direction to do so.
- Reactiveness: the ability to detect and adapt to unexpected changes.
- Social Ability: the ability to cooperate and coordinate efforts with other agents in order to achieve goals.

2.2 Dependability

The dependability of a computer system is the ability to deliver service that can be trusted *[BLU2004]*. There are three concepts that describe the notion of dependability. The Figure 2 demonstrates these concepts.

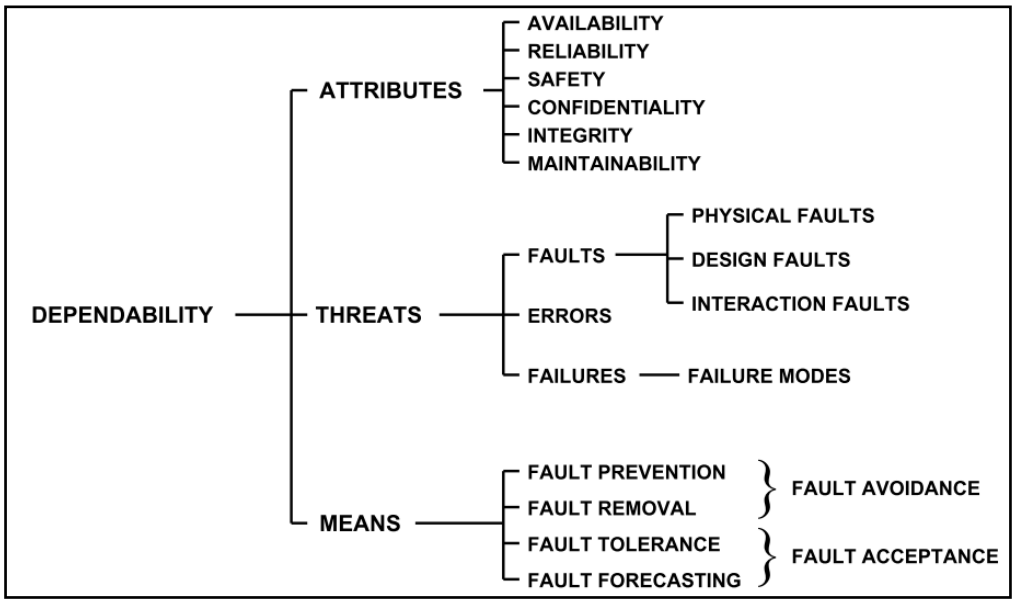


Figure 2 - The dependability concepts *[BLU2004]*

The dependability attributes could be described as *[BLU2004]*:

- **Availability:** Be available during a period of time and deliver a correct service during this time.
- **Reliability:** Continuous delivery of correct service during a period of time.
- **Safety:** Do not cause catastrophic consequences on the users and the environment.
- **Confidentiality:** Does not disclose unauthorized information.
- **Integrity:** Absence of improper state alterations.
- **Maintainability:** Ability to perform repairs and modifications of the system.

2.2.1 Threats

In this section, we present the taxonomy of threats that may affect an autonomous system. They consist of failures, errors and faults. System failures are events that deviate from the delivery of correct service. An error is an unexpected event that may cause a failure. A failure, in turn, occurs when an error reaches the service interface. Finally, a fault

is the cause of an error. If a fault produces an error it is active. On the other hand, faults that does not produce any errors are called dormant.

A system can fail in different ways. There are three different taxonomies for faults *[BLU2004],* as we show in the Figure 3.

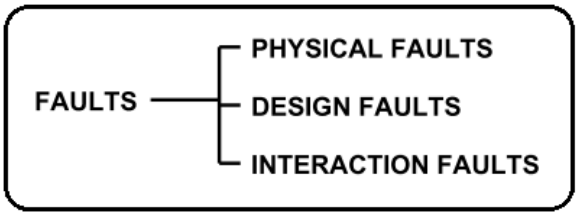


Figure 3 - A fault taxonomy *[*AAV2001*]*

2.2.1.1 Physical Faults

Physical faults are faults due to adverse physical phenomena. For example, a hardware sensor that does not work as expected, returning a non-valid value. A common way to detect this kind of problems is comparing the output of two independent identical units, like a sensor.

2.2.1.2 Design Faults

Design faults are faults unintentionally caused by man during the development of the system. This kind of faults could be either hardware or software faults. Redundant elements are a common way to detect and avoid this kind of faults.

2.2.1.3 Interaction Faults

Interaction faults are faults resulting from the interaction with other systems or users. There is a distinction between accidental faults and malicious interaction faults. An operator mistake is an example of an accidental fault and an intentional attack is a example of malicious fault.

2.2.2 Means

For these three categories of faults mentioned before there are different ways to prevent these faults. These approaches to prevent the faults are called *means* in this diagram on the Figure 4.

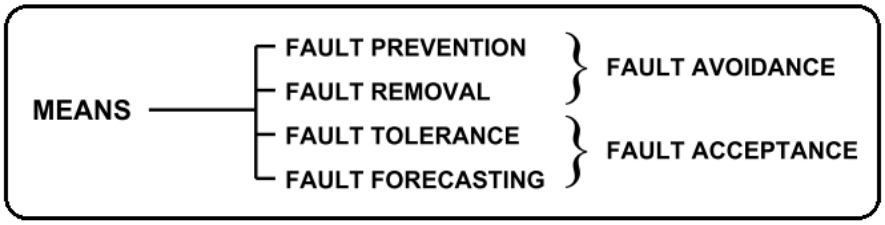


Figure 4 - Means - Fault remove techniques *[BLU2004]*

2.2.2.1 Fault Prevention

It is a way to prevent the occurrence or introduction of a fault. Fault prevention can be considered as a fault avoidance system.

2.2.2.2 Fault Removal

It is a way to reduce the number or to reduce the severity of a fault. Fault removal can be considered as a fault avoidance system. Both Fault Prevention and Removal are the attempt to develop a system without faults.

2.2.2.3 Fault Tolerance

It is a way to continue delivering the correct service even when a fault occurs. Fault Tolerance implements the concept of fault acceptance, which attempts to reduce the consequence of a fault. The main difference between fault tolerance and maintenance is that maintenance requires the participation of an external agent and fault tolerance not.

2.2.2.4 Fault Forecasting

Is a way to estimate the future incidence or the consequences of faults. Fault forecasting also implements the same concept of fault acceptance, i.e., an attempt to reduce or estimate the consequence of a fault.

The development of a dependable computing system usually combines different techniques. This work is focused on the Fault Tolerance technique, knowing that fault is almost inevitably. Fault tolerance concepts through the redundancy of multiple robotics or redundant sensors is a good approach to keep the system working as expected, even after faults occur.

2.2.3 Fault Tolerance

Fault tolerance mechanisms typically consist of an error detection and error recovering mechanisms *[LUS2004]*, as illustrated in Figure 5.

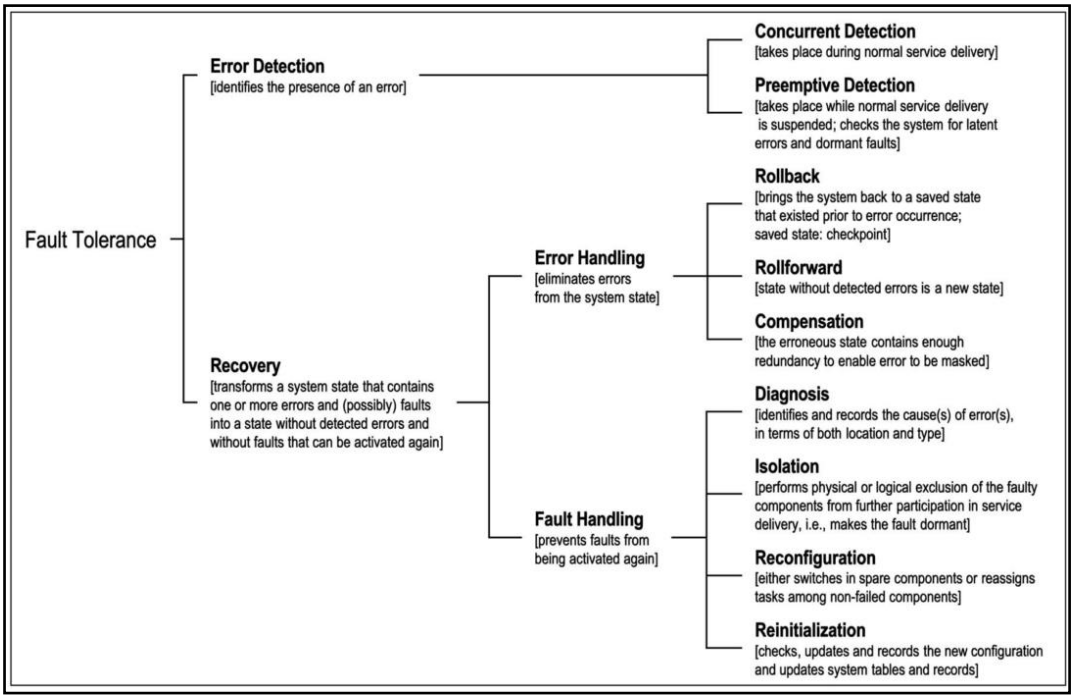


Figure 5 – Fault Tolerance Techniques *[AAV2004]*

2.2.3.1 Error Detection

Error detection originates from an error signal from the system. There are two classes of error detection:

1. Concurrent Error Detection: the error detection works at the same time of the service delivery
2. Preemptive Error Detection: check for error while the service delivery is suspended. Also check for dormant faults.

In this work the focus is on the *concurrent error detection* system that enables the service delivery and fault tolerance at the same time.

2.2.3.2 Error Recovery

Recovery *[BLU2004]* is the process that transforms a system from a state that contains faults and errors to a state that can be activated again without presence of any error or fault. Error recovery eliminates errors in three forms:

- Rollback: Return the system to a previous state where the system can be activated again. The previous saved state is called a checkpoint or safe point. Rollback is the most popular approach to recovery a system, however it is time and resource consuming.
- Rollforward: Put the system in a state where there are no errors or faults. This is a new state not previously recorded. Restart the system is a possible solution for this approach. Note that rollback and rollforward are not mutually exclusive. Usually rollback is the first attempted and then rollforward is a second option.
- Error Compensation: The erroneous state contains enough redundancy to handle the fault situation and enable error elimination. A common approach for error compensation is the fault masking. This approach requires three or more identical or similar components to be used implementing a vote system where the majority is chosen.

These three techniques eliminate errors from the system state. Rollback and rollforward are invoked on demand. Compensation can be applied either on demand or systematically, at pre-scheduled events, independently of the presence of errors.

2.2.3.3 Fault handling

Summon *[ROG2006]*, Fault handling prevent fault from being activated again. There are four techniques of fault handling as explained below:

- Diagnosis: Identifies the root cause of error in terms of location and type.
- Isolation: Perform exclusion of the faulty components from further participation in service delivery. The exclusion could be both logical and physical. For physical exclusion the fault component must have a spare component for take over the tasks.
- Reconfiguration: Set up a new configuration avoiding failed components (when it is possible).
- Reinitialization: Checks, updates and records the new configuration and updates system tables and records.

2.3 Multiple Robots Systems (MRS)

Multiple Robot Systems are systems comprised of several robots. This kind of system show some advantages over Single-Robot Systems (SRS). One of these advantages is an increased task completion speed through parallelism. MRS can also perform better in tasks that are inherently distributed in space, time, or functionality. Furthermore, the cost of multiple single-specialization robots can be lower than that of a unique all-capable robot. Finally, the use of multiple robots can eliminate single points of failure, which increase the robustness and reliability of the system through redundancy [LEP2008].

However, there are some drawbacks to MRS. For instance, determining how to manage the whole system is usually more complex than in a SRS. The lack of centralized control is one of the reasons for increased complexity of MRS [VGO2004]. Also, MRS require increased communication to coordinate all the robots in the system. Increasing the number of robots can lead to higher levels of interference between themselves, depending on the used communication device and protocol. Additionally, each individual robot in the MRS should be able to work even when the whole system state is unknown [MJM1995].

2.4 Dependable Multiple Robotic Systems

Summon *[LEP2012]* defines reliability in robotics as the probability of a determined system delivery the correct service without failure during a period of time. Different measures of reliability can be given in robotics. For example, an individual component, or an individual robot, or even a MRS can be measured. MRS should avoid as much as possible to have a single point of failure. Instead, the system must be distributed and able to work as a single. Because the large number of individual components/robots, the MRS could be fault tolerant to an uncertain environment. Also, the MRS known as swarm robots can properly handle a single robot failure. According to *[MOH2009],* there is a difference between MRS and swarm. Swarm is a new approach that takes inspiration from social insects to coordinate multi-robot systems.

2.4.1 Reliability in Robotics

Robotics is a research area with a vast amount of literature, even though only a limited part of this effort addresses reliability in robotics *[MLL1998].* Also the analysis to explore the reasons of how the robots fail is not very common in the literature *[JCA2003]*. Centralized approaches to online diagnosis MRS do not scale well basically for two different reasons: complexity of the solutions and the need of communicate each individual to a central diagnoser *[DAI2007]*.

Computers use unreliable components and, for this reason, they improve their reliability using techniques like error control codes, duplication with comparison, triplication with voting, and diagnostics to locate failed components. Similar reliability techniques can be applicable for robotics.

One of the main reasons why mobile robots fail is because the real environment cannot be completely mapped and it is naturally dynamic. Because of the dynamic environment, fault-tolerant mobile robots have to be able to handle and even learn from new situations several times. Because of this complex scenario, there are several approaches to implement reliability in robotics. Section 2.4.2 introduces some of these techniques and explains dependability in MRS.

2.4.2 Reliability in Multiple Robotics Systems

Multiple Robots Systems (MRS) need to be reliable as a whole *[LEP2012]*. For these reasons there are some questions to be addressed:

- How to detect when robots have failed?
- How to diagnose robots failures?
- How to respond to these failures?

Instead of single-robots systems (SRS) that are designed to be robust as a single, multiple robots systems (MRS) are design to be fault tolerant, it means, continue working even after a fault occurs. MRS are designed to take advantage of the collective to accomplish the work as a team, it means, they need to be able to communicate between them and a healthy robot could take over a task from a robot in a faulty state.

The main reason of MRS is to achieve significant level of reliability through the redundancy or multiple robots. The key motivation is that several robots faults can be

overcome by the redundancy system. In order to achieve this level of reliability the whole system must be developed with these faults in mind. Internal and external reasons can drive the MRS to a fault state. A software design defect is an internal reason that could lead a robot to a fault state. On the other hand, an unexpected environment changes driving the robot to a fault state is an example of external problem. Usually problems caused by external reasons are more difficult to handle or avoid than the internals.

These are some of the challenges of achieve reliability in MRS:

- Individual robot failure: The total number of individual components parts in a system is directed related with the probability of a fault occurs *[JCA2005]*. In Carlson and Murphy observed many different causes of failures leading to low reliability of robots operated by humans. This study also showed that custom designed components are less reliable than mass-produced components such as power supply and sensors.
- Local perspective: Each one of the robots maintains only a local perspective and is not able to see the system as a whole. In order to keep the entire system fault tolerant, the system should be distributed and not centralized. It allows the system to be more fault tolerant and also brings scalability to the MRS.
- Interference: The existence of MRS sharing the same physical environment can cause interference and contention. These issues must be addressed to enable MRS application.
- Software errors: As all complex software systems, the MRS software can also contain bugs that raise faults. Because of the complexity these software, defects/bugs could be difficult to detect and to fix.
- Communication failures: In MRS the communication between the individual robot is a requirement to enable the whole system works as expected. According to *[RCA1993],* all individual robot have to be able to work even when the communication with others are not available.

3 STATE OF THE ART

According to *[LEP2012]* there are several possibile faults in robotic systems, such as: robot sensors faults, uncertain environment models, limited power, and computation limits.

In order to address these complex faulty scenarios there are some tools developed that intend to help engineers and developers to handle these problems. Robot middlewares are one of these tools developed to abstract part of the complexity of these problems.

Several robot middlewares *[BRG2009]*, *[BRG2010]*, *[MAK2007]* address the fault detection problem. However, their approach is limited to monitoring a single robot at a time, which makes it difficult to observe the system as a whole. Also, most of those approaches are driven by the capabilities of robotics middleware and not by the needs of the robotics research field.

3.1 Individual Robot Fault Detection

According to *[MHA2003]* the most popular method of fault detection in robot systems is comparing sensor values with a pre-determined range of acceptable values (*i.e,* thresholds). Other well-known fault detection method is creating a vote system based on different redundant components *[RCA2003]*. If an individual component is in a faulty state, it will vote differently than the majority. This individual component could be ignored and the others values are used instead.

Logging is a fault detection/monitoring technique where data is collected during runtime to be analyzed later (*i.e.,* off-line fault detection). During the normal runtime, all necessary data is collected and stored in a collector device. One disadvantage of this technique is that a huge amount of data could be generated and it cannot be used to issues online. Usually Logging needs another tool to monitor the and trigger clean-up actions *[LOT2011]*. Logging could be used for SRS or for MRS.

3.2 Multiple Robots Fault Monitoring

Fault detection in MRS *[MEN2010]* is more complex due to the distributed nature of these systems. A networked control system is a requirement to connect all robots in MRS, which adds another layer of possible fault scenarios. Furthermore, the network itself is subject to faults, performance deterioration or operation interruption.

According to *[MEN2010],* several different methods and techniques to deal with these problems can be found in the literature. However, these methods usually follow a centralized approach. This introduces an undesirable central point of failure in MRS. A technique that could be used to monitor MRS is the Distributed Artificial Intelligence (DAI). This methodology is based on the creation of a system of multiple supervision agents that are able to communicate directly with each other in order to perform monitoring tasks. Summon et al. *[CHR2009]* states that one of the most important advantages of swarm robotic systems is redundancy. If one robot fail, another robot can try to make repairs or even take over taks assigned to the failed robot. The solution proposed in this work is creating a completely decentralized algorithm to detect non-operational robots in a swarm robotic system. Every robot flashes and their neighbors can detect the flashs and start to flash in synchrony. Robots in a faulty state do not flash periodically and can be detected by others. This innovative approach does not use conventional networking communication to perform monitoring tasks. The advantage of this approach over others is that it does not generate network traffic and it does not depend on a conventional network.

The work [KBL2006] proposes common metrics to evaluate the effectiveness of fault-tolerance solutions. Effectiveness is measured by identifying the influence of fault-tolerance towards overall system performance. According to this work only a few fault tolerance solutions are designed to consider the distributed and decentralized nature of MRS. An appropriate fault tolerant controller that implements fault detection and diagnosis systems is necessary for monitoring MRS.

RoSHA (Multi-Robot Self-Healing Architecture) *[RSH2013]* is an architecture proposal that offers self-healing capabilities for MRS. The authors argued that the architecture should be resource efficient and generates minimum impact on the system. Scalability is another important requirement. The self-healing add-on should be independent from the size of a MRS or from its robot distribution. Beside these envisioned features of a self-healing architecture, humans should be still able to oversee and control the system. There are five key characteristics of the RoSHA architecture: resource-efficient, high degree of configurability, human controllability, extensibility, and modularity.

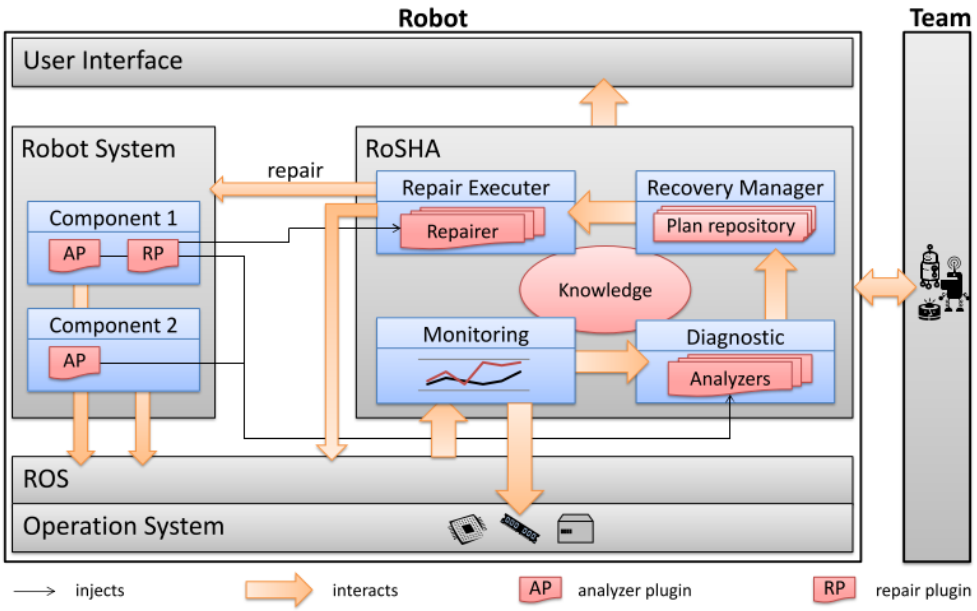


Figure 6 – Overview of the RoSHA architecture ***[RSH2013]***

Figure 6 shows the RoSHA architecture is divided in 4 components. The monitoring component collects information about the current system state. The diagnostic component uses the collected information to identify failures and their root causes. Detected faults are reported to the recovery manager. This component selects a recovery plan from a set of predefined policies to recover from failure. The execution component provides a set of generic repair actions.

The integration of the self-healing add-on in an already existing MRS is essential in the sense of practical usage, in order to foster real-world applications and to increase the commercial use. This work is a very advanced proposal on how to handle the MRS dependability challenges, however this work presents only a proposal on how to address a possible solution and do not contain experiment or any artifact that this proposal was already implemented or intend to be in the future.

Kaminka et al. [KAM2002] present an approach to monitor multi-agent systems by observing their actions by 'hearing' the routine communication among these agents. The results show that the proposed approach has a monitoring performance comparable to a human expert. There is no evaluation on the computing performance overhead on the agent and the network bandwidth overhead imposed by the communication overhearing among the agents. In addition, the authors say that the so called *report-based monitoring*

requires modification on the robot’s software plans and it generates major network bandwidth usage.

4 DEVELOPED ARCHITECTURE

This chapter presents in Section 4.1 the used tools initially studied until the actually used tools (Nagios and ROS) were selected. Section 4.2 presents the proposed monitoring environment and the adaptations required for the integration.

4.1 Techniques and Tools Analyzed

This section compares the two main types of tools used in this research: IT infrastructure monitoring and robotics middleware.

4.1.1 IT Infrastructure Monitoring

The goal of a Data Center Infrastrucutre Management or DCIM is to provide an overview of the monitored server status. DCIM tools allow the administrators to log and analyze status information related to datacenter servers *[COL2012]*. There are several DCIM tools for commercial use, open-source and free software licenses. Some solutions support the development of extensions or plugins. These plugins are used to add capabilities to the monitoring tool. The remainder of this section introduces well-known IT infrastructure monitoring tools.

Ganglia *[GAN2013]* is a "scalable distributed monitoring system" developed for cluster based systems. It provides an overview of your entire clustered system. Some of the main features about Ganglia are distributed design for clusters, use of technologies such as XML, XDR for compact, portable data transport, and data storage and visualization. The algorithms were developed to high performance work with concurrency.

Spiceworks *[SPI2013]* is a free network/system monitoring tool. This tool uses the SNMP protocol since it has low impact (minimal overhead) on the network communication with monitoring tasks. Pre-defined alerts can be configured to monitor the system status. The administrator is also able to select each of these alerts and see more detailed information about the node.

Zabbix *[ZAB2013]* is another network monitoring tool which offers a web interface console with different views and mappings. The MySQL database is used to store historical information. It is developed in C and the web interface is developed in PHP. Some of the protocols supported are SNMP, TCP and ICMP.

Nagios is a well-known IT infrastructure monitoring according to *[NAG2013]*. This monitoring system was developed to support scalability and flexibility. Nagios provides information about the entire IT infrastructure, allowing detecting, send alerts and also repairing problems. Nagios supports the development of extensions or plugins to add capabilities to the tool. Nagios is an open source application that monitors virtually any kind of device for problems and reports the results. Nagios was designed for Unix-based systems.

4.1.2 Nagios

This section details Nagios features which motivated us to select it for this research.

Nagios provides a plataform for executing specific checks on the entire monitored system customized by devices. Pratically any kind of device information can be monitored, for instance the use of memory, free space on disks, cpu load, the number of processes, and many other cutomized information *[NAG2005]*. Nagios provides an easy web interface for graphical view the entire system and simple navigation into nodes showing detailed monitoring information.

According to *[NAG2005],* every test performed by Nagios is executed by an external program called plugin. A set of plugins are distributed with Nagios and can be loaded as required. For example, there is a generic plugin to test TCP connections called *check_tcp* plugin. This plugin can be used to determine if a service is reachable through the network.

Some information about monitored servers may not available through network services. For instance, there is no network protocol for checking free capacity on a hard drive. In this case Nagios is able to access the server via a remote shell (SSH for instance) and capture this information.

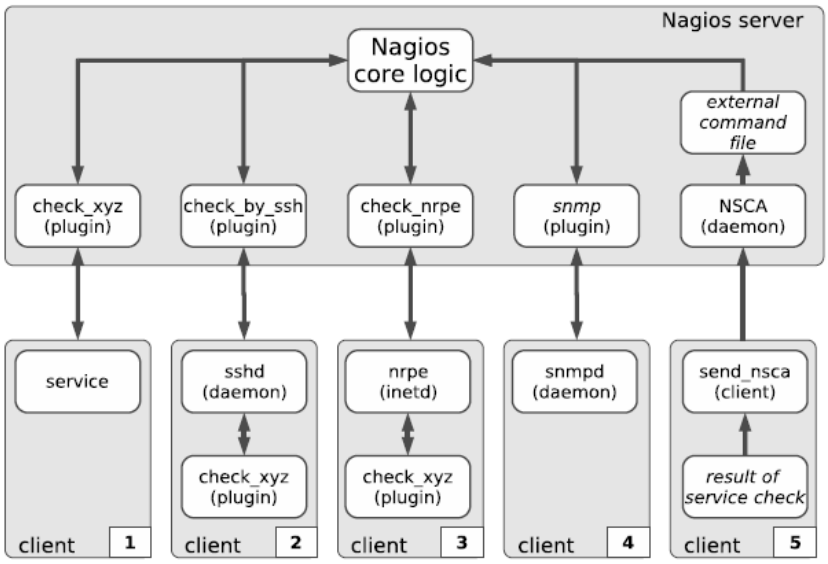


Figure 7 – Nagios allows different tests methods *[NAG2005]*

Figure 7 shows an overview of different test methods supported by Nagios. The upper box contains all the components that run directly on the Nagios server. Nagios has a flexible design that allows the development of extensions to communicate and monitor almost any kind of system through a Nagios plugin. A Nagios plugin is a small piece of software that must be developed following the Nagios plugin specification in order to support Nagios API. Plugins used for host and service checks are separate and independent programs that can also be used outside of Nagios.

In order for Nagios to use an external program, it must follow Nagios plugin rules. First, the return status generated by the plugin must return OK, Warning, Critical or Unknown status *[NAG2005]*. Listing 1 demonstrates the execution of the *check_tcp* plugin. It shows Nagios monitoring if the ROS service is active on port 11311, on the target host 192.168.1.3.

```
nagios@linux:nagios/libexec$ ./check_tcp -H 192.168.1.3 -p 11311

TCP OK - 0,061 second response time on port 5631 | time=0,060744s;0,
000000;0,000000;0,000000;10,000000
```

Listing 1 – Nagios example checking remote TCP port *[NAG2005]*

The ROS core service uses TCP port 11311 by default, so this simple example could determine if ROS is up and running on a specific host/robot. It is important to notice that this example only checks ROS core service status, without collecting any information from robot sensors.

The second rule is the use of '-' to separate status code from the detailed textual status. Listing 1 also presents this part of the Nagios status message.

A Nagios plugin can be a simple bash script developed to execute steps and print the formatted output on the standard output. Listing 2 shows a simple Nagios plugin source code.

```
#!/bin/bash

NAGCHK="/usr/local/nagios/libexec/check_nagios"

PARAMS="-e 60 -F /var/nagios/nagios.log -C /usr/local/nagios/bin/nagios"

INFO=`$NAGCHK $PARAMS`

STATUS=$?

case $STATUS in

0) echo "OK : " $INFO

;;

*) echo "ERROR : " $INFO | \

/usr/bin/mailx -s "Nagios Error" nagios-admin@example.com

;;

Esac
```

Listing 2 – Nagios simple plugin source code example *[NAG2005]*

According to *[NAG2005],* Nagios also supports the executation of a plugin via SSH on a remote host. Nagios administrator needs an account on the target system in order to connect and to execute the plugin. The Nagios Remote Plugin Executor (NRPE) is another method to execute plugins remotely. This plugin can be useful for indirectly testing hosts/services that are not reachable from the Nagios server network, as illustrated in Figure 8.

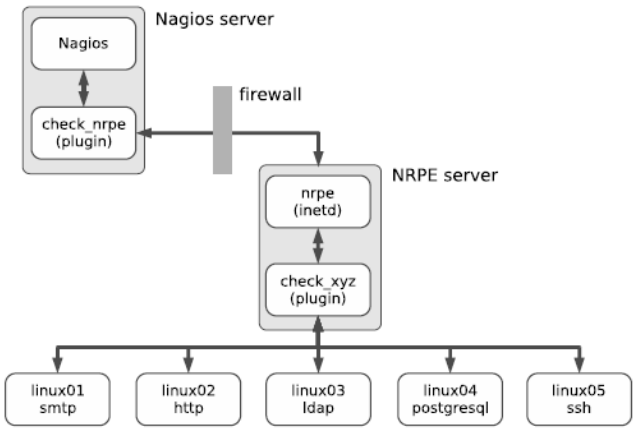


Figure 8 – Nagios executing a remote check using NRPE *[NAG2005]*

Nagios also has a complete Notification System that could be configured to perform notification via email/SMS or any other communication protocol according with pre-defined configurations/rules *[NAG2005]*. The Figure 9 shows the Notification System details.

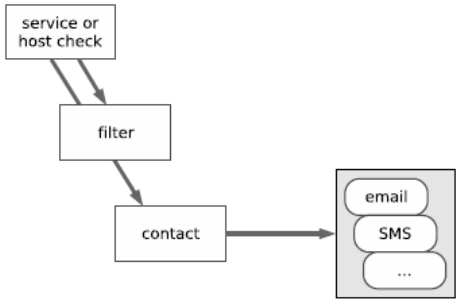


Figure 9 – Nagios Notification System overview *[NAG2005]*

Scalability is another characteristic of Nagios. Several noncentral Nagios instances could be executed and configured to send their results to a Nagios Central Server using the Nagios Service Check Acceptor *[NAG2005],* as illustrated in Figure 10.

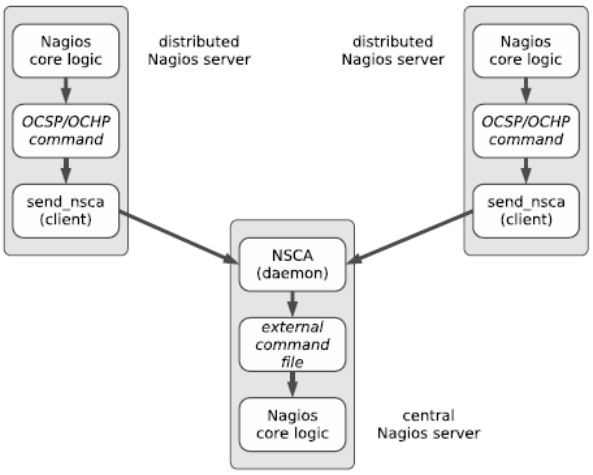


Figure 10 – Distributed monitoring with Nagios *[NAG2005]*

According to *[NAG2005],* Nagios has several reports that can display graphics of performance data collected from the hosts (e.g. time series dynamic charts). It is available through third-party software that could be configured to work integrated to Nagios.

4.1.3 Robotic Middleware

According to *[ELK2012],* a robotic middleware is a layer between the robot operating system and software applications, as illustrated in Figure 11. The middleware layer is designed to allow reuse of software and reduce costs of development.

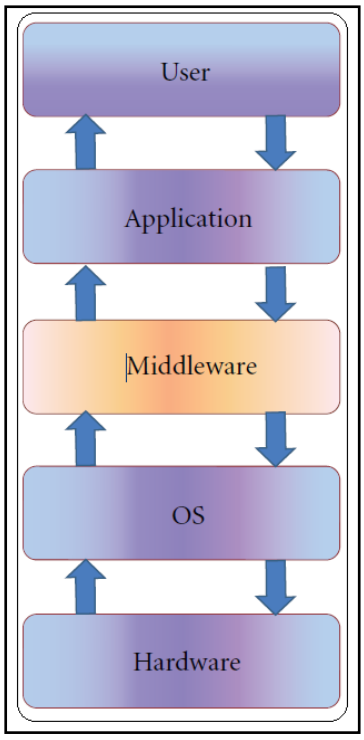


Figure 11 - Middleware layer *[ELK2012]*

Modern robots are considered complex distributed systems consisting of a number of integrated hardware (such as the embedded computer and specific robotics sensors) and software modules. All robot's modules (hardware and software) work together to achieve their goals *[MOH2008]*. This section describes some of these existent solutions and briefly explains some criteria used to select one.

Miro *[SEN2001]* and *[HUT2002]* is a robot middleware developed by University of Ulm, Germany. Miro is designed and implemented by applying object oriented design. According to [MIR2013] the core components have been object oriented developed using a multi-platform framework supporting network and real time communication.

Orca *[AMA2006]* is a middleware framework for developing component-based robotics. It is designed to support a wide range of applications. The main goal of Orca is to enable software reuse in robotics. According to [ORC2013] it provides ways to develop nodes that can be composed to create complex robotic systems.

According to *[SAH2006]* and *[SKJ2006],* UPnP middleware was developed under a Universal Plug and Play (UPnP) architecture. One of main features of the UPnP architecture is the peer-to-peer network connectivity *[UPN2013]. UPnP also supports standard protocols like TCP/IP, HTTP and XML.*

Robot Operating System (ROS) is robot middleware designed to reduce the cost of development for large-scale robots systems. According to *[MQU2009]* the ROS main characteristics are:

• Peer to peer communication to reduce traffic in the network;

• Tools-based: micro kernel designed instead of monolithic kernel;

• Multi-lingual support;

•Thin: software development libraries with no dependencies on ROS;

• Free and open-source under BSD license;

• Organized in packages in order to build large systems.

ROS modular design allows managing the complexity of sophisticated robotic applications. ROS also promotes code reuse since a single software module can be easily used on different applications and robots. Moreover, ROS middleware provides tools for fault monitoring and diagnosis *[*LOT2011*]*. These tools are useful for development and monitoring one specific robot each time and not an entire MRS. For this reason this solution addresses only part of the overall problem of runtime monitoring because they allow to check the status of one component/module at time.

4.1.4 ROS Concepts

ROS *[ROS2014]* is organized in three levels: filesystem, computation graph and community. The first two, which are most relevant for this work, are described next.

ROS package is the most important concept of the Filesystem level. The package is the small unit of software in ROS and it includes processes (called nodes), libraries and all required configuration files. Metapackages represent a group packages.

The structure of the communication on ROS is represented by Messages. Message descriptions stored into msg files define the structure of the message. Listing 3 presents an example of a message file *Message.msg* with only one field of string type.

```
Message.msg

string input
```

Listing 3 – ROS Message type format

Service communication is also defined in a file and provides the interface for the nodes to interact with a service. Listing 4 shows an example of a Service file that declares a single String attribute for the request and another String attribute for the response.

```
Monitor.srv

        string input

        ---

        string output
```

Listing 4 – ROS Server message format

The Computation Graph abstracts network communication in ROS. It is a graph comprised of n*odes* and *topics*. Nodes are processes that perform computation and communicate with each other by passing messages to a given topic. A topic is the structure that receives the message sent by a node. A node sends a message by *publishing* it to a topic and can also read messages by *subscribing* the topic. In the context of this work, ROS suggests the use of /diagnostic topic to publish this kind of diagnostic data and /diagnostic_agg topic for grouped diagnostic information *[ROD2014]*.

4.1.5 Fault Reporting using ROS Style - ROS Diagnostics

The Diagnostics stack is the ROS software responsible for analyzing and reporting the system state. It consists of libraries and tools for collecting, publishing, and visualizing monitoring information. This tool-chain is built around standardized interfaces, named the /*diagnostic* topic for monitoring information. Gathered status data is published continuously on the diagnostic topics [ROD2014] [GP22014].

Listing 5 shows the diagnostic Status message format (diagnostic_msgs/DiagnosticStatus). The first field is a byte value that accepts one of the possible levels of operations (0, 1, 2 or 3). Second field is a string to identify which component is reporting the diagnostic. A message description could also add some more details about the diagnostic. The hardware_id field defines a unique hardware identificantion (in case of robots that contains more than one component with the same name for redundancy purposes). The last field is an array that could store any extra details for the diagnostic. For instance, a low battery message could store the remaining operational time.

```
# This message holds the status of an individual component of the robot.

# Possible levels of operations
byte OK=0
byte WARN=1
byte ERROR=2
byte STALE=3

byte level # level of operation enumerated above
string name # a description of the test/component reporting
string message # a description of the status
string hardware_id # a hardware unique string
KeyValue[] values # an array of values associated with the status
```

Listing 5 – ROS Diagnostic Status message format

4.1.6 Diagnostics Aggregator

According to [ROD2014] ROS also distributes a built-in diagnostic_aggregator package. It is designed to subscribe the /diagnostic topic, read the raw published data, reorganize all information based on pre defined rules and publish the generated result in the /diagnostic_aggregator topic.

The publisher default interval is 1 Hz but this value can be configured by the user on the YAML rule file. The diagnostic.yaml rule file also defines groups for aggregating the information according to the type of data. For example robots with more than one battery could aggregate all batteries statuses on a Battery group. For instance: My Robot/Actuators/Motor Group 1/Motor 1 means that Motor 1 belongs to Motor Group 1 and Motor Group 1 belongs to Actuators and so on. The diagnostic aggregator summarizes the least relevant systems states and emphasizes the most critical ones

Another tool built-in on ROS is the robot_monitor tool. This is a GUI tool that displays all results published on the */diagnostic_agg* topic in a hierarchical format. It groups the statuses in terms of their conditions (Ok, Warning, and Critical) and it displays the most urgent statuses (in Critical condition) first with red color to help the user to focus on the most important issues.

Figure 12 – RQT Monitor main Window *[ROD2014]*

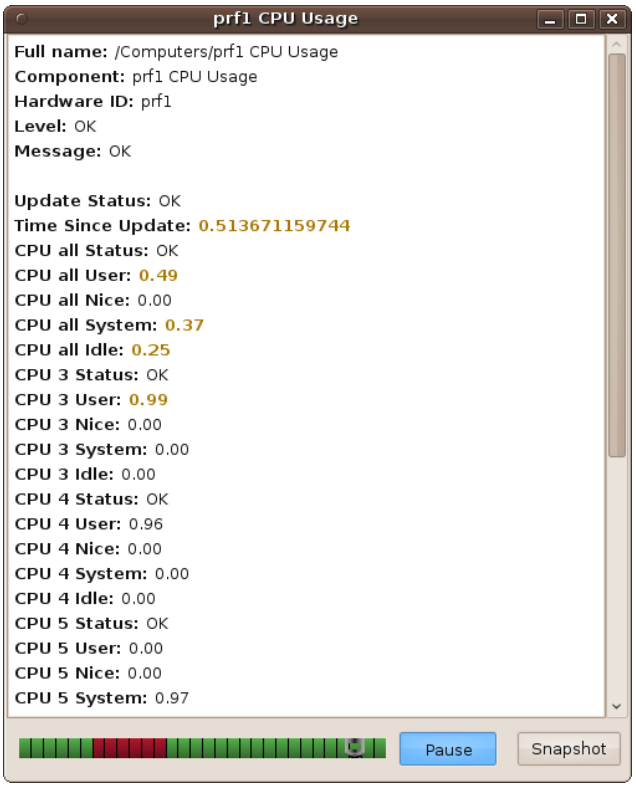


Figure 13 – RQT Monitor status viewers *[ROD2014]*

This GUI is divided in three boxes. The first box shows only the errors in a list, the second one shows the warnings. The third contains a tree with all items organized in a list view. The colored line in the bottom of the screen is a timeline where each timeframe represents an update. A detailed window can be opened by clicking in the desired item, as illustrated in Figure 13.

4.1.7 Turtlebot Kobuki

This section presents the iClebo Kobuki *[KBK2013]* mobile base used in this research. According to *[KBK2013],* iClebo Kobuki is a mobile base designed for research in robotics. Figure 14 shows Turtlebot mounted over a Kobuki mobile base. It provides sensors and actuators, as summarized next.

Figure 14 – Turtlebot Kobuki *[KBK2013]*

Functional Specification

- Maximum translational velocity: 70 cm/s
- Maximum rotational velocity: 180 deg/s (>110 deg/s gyro performance will degrade)
- Payload: 5 kg (hard floor), 4 kg (carpet)
- Cliff: will not drive off a cliff with a depth greater than 5cm
- Threshold Climbing: climbs thresholds of 12 mm or lower
- Rug Climbing: climbs rugs of 12 mm or lower
- Expected Operating Time: 3/7 hours (small/large battery)
- Expected Charging Time: 1.5/2.6 hours (small/large battery)
- Docking: within a 2mx5m area in front of the docking station

Hardware Specification

- PC Connection: USB or via RX/TX pins on the parallel port
- Motor Overload Detection: disables power on detecting high current (>3A)
- Odometry: 52 ticks/enc rev, 2578.33 ticks/wheel rev, 11.7 ticks/mm
- Gyro: factory calibrated, 1 axis (110 deg/s)
- Bumpers: left, center, right
- Cliff sensors: left, center, right

- Wheel drop sensor: left, right
- Power connectors: 5V/1A, 12V/1.5A, 12V/5A
- Expansion pins: 3.3V/1A, 5V/1A, 4 x analog in, 4 x digital in, 4 x digital out
- Audio : several programmable beep sequences
- Programmable LED: 2 x two-coloured LED
- State LED: 1 x two coloured LED [Green - high, Orange - low, Green & Blinking - charging]
- Buttons: 3 x touch buttons
- Battery: Lithium-Ion, 14.8V, 2200 mAh (4S1P - small), 4400 mAh (4S2P - large)
- Firmware upgradeable: via usb
- Sensor Data Rate: 50Hz
- Recharging Adapter: Input: 100-240V AC, 50/60Hz, 1.5A max; Output: 19V DC, 3.16A
- Netbook recharging connector (only enabled when robot is recharging): 19V/2.1A DC
- Docking IR Receiver: left, centre, right
- Diameter : 351.5mm / Height : 124.8mm / Weight : 2.35kg (4S1P - small)

iClebo Kobuki *[KBK2013]* provides C++ drivers for Linux and ROS compatibility. This robot already implements the diagnostics information necessary to perform real time monitoring and to integrate it to the IT Monitoring tool. Kobuki driver provides status information about the Watchdog, Battery, Cliff Sensor and others. Listing 6 is one example of the Kobuki's diagnostic raw data.

```
mobile_base_nodelet_manager: Watchdog: No Signal
mobile_base_nodelet_manager: Analog Input: [4095, 4095, 4095, 4095]
mobile_base_nodelet_manager: Battery: Healthy
mobile_base_nodelet_manager: Cliff Sensor: All right
mobile_base_nodelet_manager: Digital Input: [0, 0, 0, 0]
mobile_base_nodelet_manager: Gyro Sensor: Heading: -19.92 degrees
mobile_base_nodelet_manager: Motor Current: All right
mobile_base_nodelet_manager: Motor State: Motors Enabled
mobile_base_nodelet_manager: Wall Sensor: All right
mobile_base_nodelet_manager: Wheel Drop: All right
```

Listing 6 – Kobuki diagnostic raw data

The watchdog sensors detect when the Kobuki is connected to the computer via USB, in this example there is no signal from the robot. Analog input represents the status of the analog sensors present in the robot. Battery shows the robot's battery status. The Cliff sensor detects if the robot is in a flat surface or uphill. Digital input are digital buttons controlled via software. The gyro sensor gets the current robot orientation. Motor Current monitors the electrical current in the motor and reports its status (in terms of OK, Warning or Error). Motor State informs whether the motor is enabled or disabled. Wall sensor detects when the robot hits an obstacle. The wheel drop sensors detects if one of the wheels is not properly in contact with the surface.

4.2 Proposed Multi Robot Monitoring Architecture

This section describes the proposed Multi Robot Monitoring architecture, with a focus on the development approach.

The proposed software architecture, illustrated in Figure 15, creates a connection between Nagios and the ROS diagnostics aggregator topic. This connection is responsibility of the ROS Diagnostics Nagios Plugin. Most of the robots manufacturers, like *Turtlebot Kobuki,* provide drivers that are compatible with ROS *diagnostics* and *diagnostics aggregator* topics. The ROS Diagnostics Nagios plugin connects to the robot's ROS diagnostic aggregator node through ROS APIs, gets the requested information and prints the output on the standard output format required by Nagios engine.

ROS core is a service application listening on a specific TCP/IP port and waiting for subscriber connections. All communication process is executed using XML-RPC protocol. Figure 15 shows the architecture.

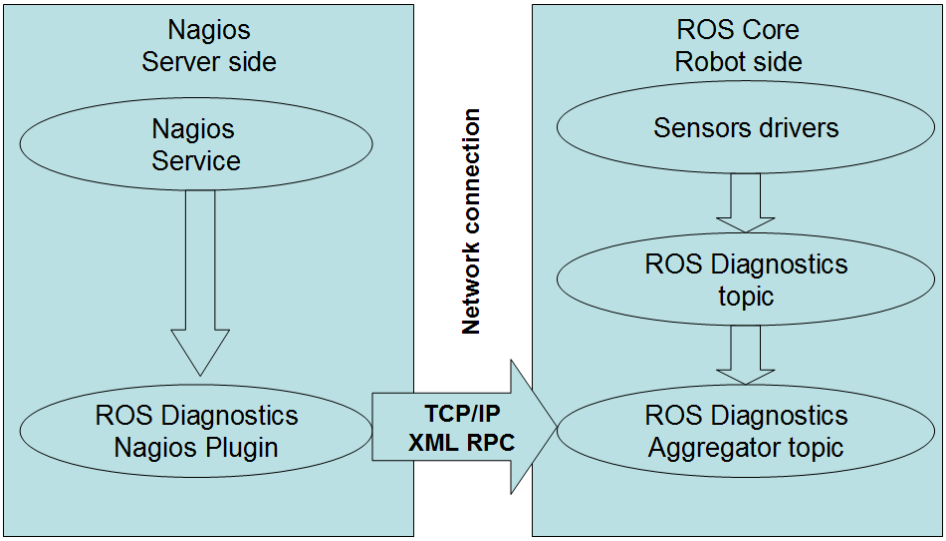


Figure 15 - Software Architecture

Nagios remotely connects to the robot and gets the required information without the need of any additional software running at the robot side. The only requirement is the diagnostic aggregator topic, which is already present in most ROS compatible robot platforms.

A python ROS Diagnostics Nagios plugin was developed to access the robot's ROS core topic via XML-RPC. This plugin subscribes to the diagnostic aggregator topic of each monitored robot and parses the information to the Nagios output format. The ROS Diagnostics Nagios plugin source code is presented on the Appendix.

The plugin supports any number of sensors and arbitrary sensor names. This enables the plugin to monitor most robots. Listing 7 shows the plugin syntax.

```
$ ./ros-diagnostics_agg.py --help

Usage: ros-diagnostics_agg.py [options]

Options:

  -h, --help            show this help message and exit

  -H HOST, --host=HOST  Define the target host

  -N NAME, --name=NAME  Define the sensor name

$ ./ros-diagnostics_agg.py -H <host>

OK - OK Sensor(s) list: /Camera, /Camera/Cam1, /Laser, /Laser/Laser1, /Laser/Laser2, /Motor, /Motor/Motor1, /Motor/Motor2, /Motor/Motor3, /Power, /Power/Laptop Battery, /Power/Robot Battery, /Temp, /Temp/Sensor1, /Temp/Sensor2
```

Listing 7 – ROS Diagnostic plugin syntax

This example shows a robot overall status, which is described as OK. Next it lists the name of topics which carry the status of different parts of the robot. The robot overall status and its topics can have 3 possible states: Ok, Critical and Warning. The robot overall status assumes the most severe status of all monitored topics. The Listing 8 shows topics reporting statuses with different states.

```
CRITICAL - CRITICAL sensor(s) list: /Camera, /Camera/Cam1, WARNING sensor(s) list: /Power, /Power/Laptop, OK sensor(s) list: /Laser, /Laser/Laser1, /Laser/Laser2, /Motor, /Motor/Motor1, /Motor/Motor2, /Motor/Motor3, /Power, /Power/Robot Battery, /Temp, /Temp/Sensor1, /Temp/Sensor2
```

Listing 8 - ROS Diagnostic plugin output

The plugin also has the ability to monitor only specific sensors status. Listing 9 shows, for example, the same plugin used to monitor only batteries status.

```
$ ./ros-diagnostics_agg.py -H <host> -N battery

OK - OK Sensor(s) list: /Power, /Power/Laptop Battery, /Power/Robot Battery
```

Listing 9 - ROS Diagnostic plugin monitoring only Battery

In this case all other sensors not containing battery in the name are ignored by the plugin.

Filtering by name allows more configuration flexiblity. This is achieved by enabling Nagios to monitor specific sensors independently. For example, Nagios can be configured to monitor the Motor status every five minutes and the temperature sensor every thirty seconds.

The monitoring system is completely independent of the robot application. It means that if the monitor server stops, only the monitor system will stop to working. The robot application will continue working exactly the same way.

On Nagios configuration each monitored hosts has a configuration file on `/etc/nagios/conf.d/hostname.cfg`. This file contains all the informaton about the host. Basic information of the host includes hostname, alias, IP address, and which checks must be executed by Nagios. Listing 10 shows an example of this file.

```
define host{
        host_name       host_name
        alias           alias
        display_name    display_name
        address         address
}
```

Listing 10 – Nagios define host syntax

The configuration file defines the specific rules for the host it describes. This means that every different host/robot monitored by Nagios could have specific configurations. For example, robots with different number/kind of sensors could be defined properly on this file. Also the polling interval of each sensor could be defined setting the *check_interval* parameter. This flexibility allows Nagios to support heterogeneous MRS.

Once the host is added on Nagios it will be displayed on Nagios portal, which also shows the last known status of the host.

lsa-vm-simulator-10_32_177_43	Current Load	OK	2014-11-07 21:40:06	0d 0h 1m 2s	1/4	OK - load average: 0.09, 0.12, 0.17
	Current Users	OK	2014-11-07 21:40:49	0d 0h 0m 19s	1/4	USERS OK - 4 users currently logged in
	Disk Space	PENDING	N/A	0d 0h 2m 27s+	1/4	Service check scheduled for Fri Nov 7 21:41:32 BRST 2014
	Robot / Laptop battery	PENDING	N/A	0d 0h 2m 27s+	1/4	Service check scheduled for Fri Nov 7 21:42:15 BRST 2014
	Robot Camera	PENDING	N/A	0d 0h 2m 27s+	1/4	Service check scheduled for Fri Nov 7 21:42:58 BRST 2014
	Robot Lasers	PENDING	N/A	0d 0h 2m 27s+	1/4	Service check scheduled for Fri Nov 7 21:43:41 BRST 2014
	Robot Motors	OK	2014-11-07 21:39:38	0d 0h 1m 30s	1/4	OK - OK sensor(s) list: /Motor, /Motor/Motor 1, /Motor/Motor 2, /Motor/Motor 3, /Motor/Motor 4, /Motor/Motor 5
	Robot all sensors	CRITICAL	2014-11-07 21:40:21	0d 0h 0m 47s	1/4	CRITICAL - CRITICAL sensor(s) list: /Camera, /Camera/Cam 1, WARNING sensor(s) list: /Power, /Power/Robot Battery, OK sensor(s) list: /Laser, /Laser/Laser 1, /Laser/Laser 2, /Motor, /Motor/Motor 1, /Motor/Motor 2, /Motor/Motor 3, /Motor/Motor 4, /Motor/Motor 5, /Power/Laptop Battery, /Temp, /Temp/Sensor 1, /Temp/Sensor 2
	Total Processes	PENDING	N/A	0d 0h 2m 27s+	1/4	Service check scheduled for Fri Nov 7 21:41:03 BRST 2014

Figure 16 – Nagios monitoring a host status

On Figure 16, the left column of the table contains the name of the monitored host. The second column lists all the checks configured for this specific host. The following columns display, respectively, the status of the check, date/time of the last check, the number of attempts to execute the check, and any description received from the check plugin output.

5 EXPERIMENTAL ENVIRONMENT

To validate both the proposed architecture and its implementation, two experiments werer performed: a scalability experiment with up to a hundred homogeneous virtual robots and an experiment with one real robot.

5.1 Server Scalability Experiment

The experiment described in this section involves a simulated scenario with up to a hundred simultaneous virtual robots. Its goal is to allow an evaluation of the scalability of the monitoring server.

Figure 17 illustrates the architecture of the scalability experiment. The Database server in the left side was created to collect performance data (cpu load, memory usage and network bandwidth) during the experiment. This database is not actually a requirement for the monitoring solution, its purpose is only to gather data regarding the experiment.

Nagios server runs the monitoring tool. All required plugins for Nagios are installed on server side. This enables the implemented solution to work without requiring any software installed robot-side. All computers (servers and robots) need to be in the same local IP network or in a VPN. Details of this setup are presented in the following sections.

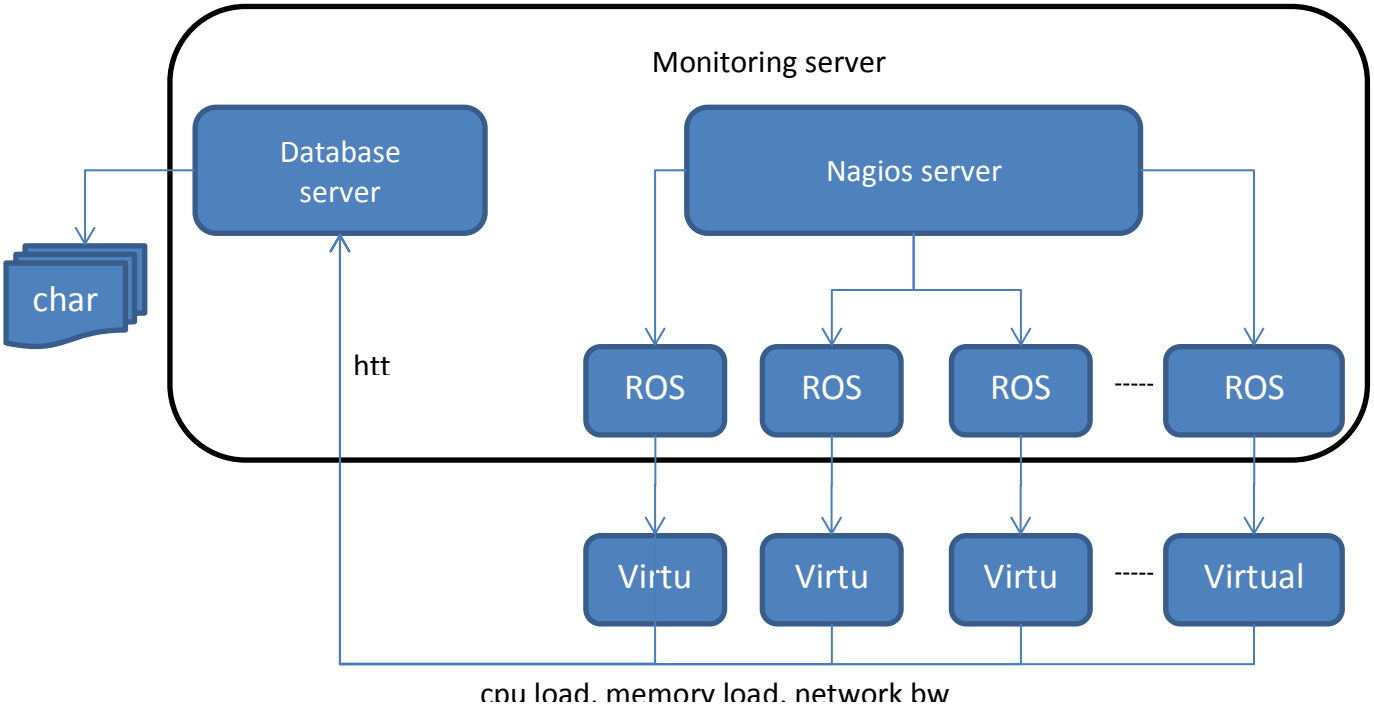


Figure 17 – Schematic view of the Server scalability experiment.

5.1.1 The Virtual Robot Setup

A virtual robot is application was developed for this experiment, using the Python programming language. It generates diagnostics data in the same format implemented by ROS diagnostics compliant robots. This application reads a set of configurations and, based on these configurations, publishes data on the *ROS diagnostics* topic.

Listing 11 shows the configuration file. It determines the total number of sensor to be simulated and also the number of sensors in the warning and error state that will be published. See section 7.6 on Appendix for its source code implementation details.

```
##################################################################################
# Configuration
##################################################################################

##################################################################################
# Simulation config
total_errors = 0
total_warnings = 0
##################################################################################
# Refresh interval in seconds
refresh_interval = 300
##################################################################################
# Robot configuration
##################################################################################
##################################################################################
# Motors config
number_of_motors = 3
##################################################################################
# Temperature config
number_of_sensors = 2
##################################################################################
# Laser config
number_of_lasers = 2
##################################################################################
# Cameras config
number_of_cameras = 1
##################################################################################
# Battery config
# The intial battery level - 100 is considered full charge
initial_battery_level = 100
# Error battery level for diagnostics
error_battery_level = 10
# Warn battery level for diagnostics
warn_battery_level = 20
```

Listing 11 – Virtual robot configuration parameters

All virtual robot diagnostic information is combined in a diagnostic aggregator node, which defines rules to parse the raw diagnostic information and categorize it into a more readable and meaningful way on the */diagnostics_agg* topic.

After the simulator node is running, all simulated data is published on the diagnostic topic at a frequency of 1Hz. There is a file that defines the rules that are interpreted by diagnostic_agg node in order to read all diagnostic raw data, to compile the information according to the groups defined in this file, and to publish the final result in the

/diagnostic_agg topic. Listing 12 shows the rules file definition in a YAML format, which defines the publication rate and analyser roles.

```
pub_rate: 1.0 # Optional
base_path: '' # Optional, prepended to all diagnostic output
analyzers:
  power:
    type: GenericAnalyzer
    path: 'Power'
    timeout: 5.0
    contains: ['Battery']
  motor:
    type: GenericAnalyzer
    path: 'Motor'
    timeout: 5.0
    contains: ['Motor']
  temp:
    type: GenericAnalyzer
    path: 'Temp'
    timeout: 5.0
    contains: ['Sensor']
  laser:
    type: GenericAnalyzer
    path: 'Laser'
    timeout: 5.0
    contains: ['Laser']
  camera:
    type: GenericAnalyzer
    path: 'Camera'
    timeout: 5.0
    contains: ['Cam']
```

Listing 12 – ROS Diagnostic aggregator diagnostics.yaml configuration

ROS provides a launcher application that reads a launch file and runs all nodes and applications defined on this file. Listing 13 provides an example of a launch file.

```
<launch>
   <arg name="battery_runtime" default="60"/>
   <node kg="rbx2_utils" name="simulator3" type="simulator.py" output="screen" clear_params="true">
   </node>
   <!-- Load diagnostics -->
   <node pkg="diagnostic_aggregator" type="aggregator_node" name="diag_agg" >
          <rosparam command="load" file="/home/lsa/catkin_ws/src/simulator/ utils/diagnostics.yaml"
/>
   </node>
   <node pkg="rqt_robot_monitor" type="rqt_robot_monitor" name="rqt_robot_monitor" />
</launch>
```

Listing 13 – ROS laucher file syntax

The launch file in Listing 13 directs the launcher to execute the simulator node, the aggregator_node and the rqt_robot_monitor. This monitor provides a Graphic User Interface to display all information generated by the simulator.

Figure 18 illustrates the rqt_robot_monitor capturing the diagnostic information generated by the developed application. This figure shows a list of the monitored devices and their statuses. The panel in the left shows the detailed information of the selected device.

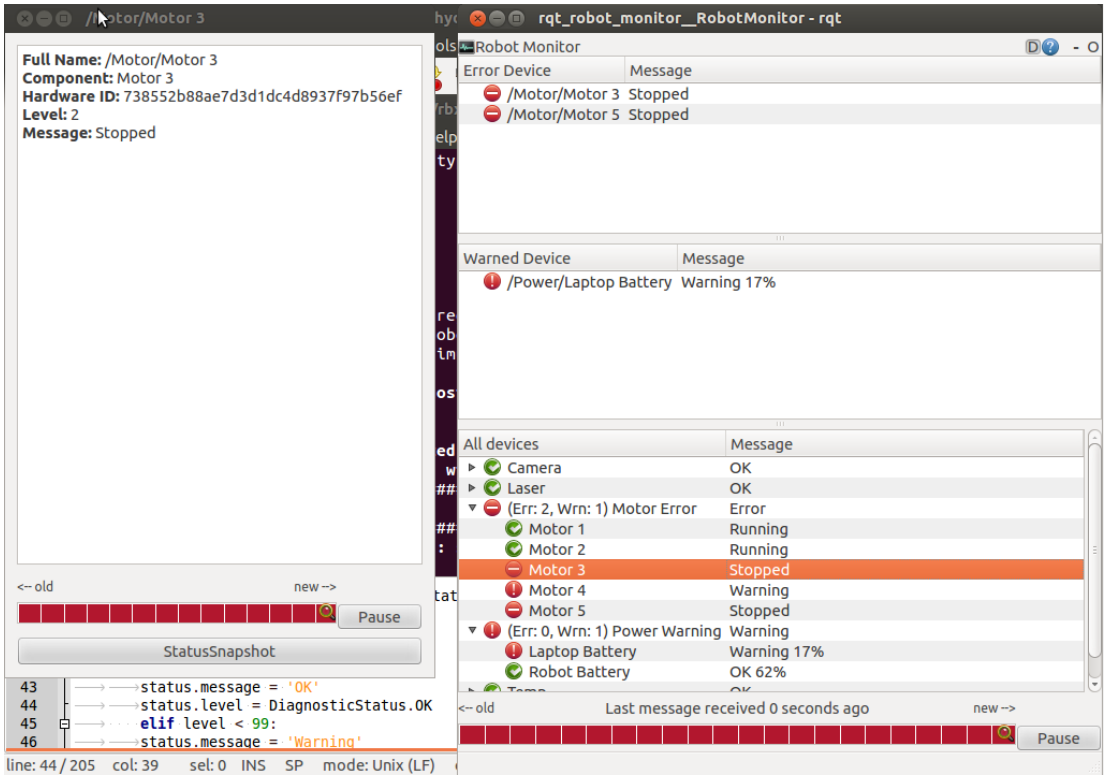


Figure 18 – RQT Screenshot of the Simulator running on a Virtual Machine

Each virtual robot runs on a virtual machine with 256 Mbytes of RAM and 1 processor. These machines execute Ubuntu 12.04 running ROS hydro. The VM image of a robot is configured to automatically start all the necessary services to run the simulation.

When a new instance of the virtual robot VM starts up, it automatically sends its IP address to the Nagios server, in order to be register itself in the monitoring server. This behaviour was implemented to ease the management of a large number of virtual machines. It is not a requirement for the proposed monitoring solution, since the robots can be registered by the operator.

A script was created to collect performance data in a VM during runtime and send this information to the database server. This is a bash script that runs on the guest OS and gets the CPU usage from `/proc` filesystem, the memory usage from *free* unix command, and the bandwidth transmitted from `/sys/class/net`. This information is sent every minute. The script runs both in Nagion Server and in the robots.

5.1.2 Monitoring Server

The Nagios monitoring server runs on a virtual machine created with 2 Gbytes of RAM and 2 processors running Ubuntu 13.10. It runs Nagios server, MySQL database

server, Apache HTTP server configured with PHP module enabled, and the proposed ROS plug-in used to collect ROS diagnostic information from the remote robots.

Each host that should be monitored by Nagios must be added on Nagios configurations files. A new file must be created on `/etc/nagios3/conf.d/` following the format presented in Listing 14.

```
# A simple configuration file for monitoring the local host
# This can serve as an example for configuring other servers;
# Custom services specific to this host are added here, but services
# defined in nagios2-common_services.cfg may also apply.
define host{
        use                     generic-host            ; Name of host template to use
        host_name               localhost
        alias                   localhost
        address                 127.0.0.1
        }
# Define a service to check the disk space of the root partition
# on the local machine.  Warning if < 20% free, critical if
# < 10% free space on partition.
define service{
        use                             generic-service         ; Name of service template to use
        host_name                       localhost
        service_description             Disk Space
        check_command                   check_all_disks!20%!10%
        }
# Define a service to check the number of currently logged in
# users on the local machine.  Warning if > 20 users, critical
# if > 50 users.
define service{
        use                             generic-service         ; Name of service template to use
        host_name                       localhost
        service_description             Current Users
        check_command                   check_users!20!50
        }
# Define a service to check the number of currently running procs
# on the local machine.  Warning if > 250 processes, critical if
# > 400 processes.
define service{
        use                             generic-service         ; Name of service template to use
        host_name                       localhost
        service_description             Total Processes
                    check_command                   check_procs!250!400
        }
# Define a service to check the load on the local machine.
define service{
        use                             generic-service         ; Name of service template to use
        host_name                       localhost
        service_description             Current Load
          check_command                check_load!5.0!4.0!3.0!10.0!6.0!4.0
        }
```

Listing 14- Nagios host configuration file example

Because of the large number of virtual robots to be added on Nagios for the scalability experiment, a script was developed to connect to the Database server and to automatically add a new host on Nagios. This is done by adding a new host on the `/etc/hostname` in Nagios server. The script, showed on Listing 15, is necessary because robots must be reachable by Hostname (ROS requirement).

```
#!/bin/bash
clear
echo ============================================================
```

```
echo "Update VM list"
echo "Require root login"
echo ========================================================================


echo ========================================================================
read -p "Apache IP address: " -e -i hostname.com.br host
echo ========================================================================

echo ========================================================================
echo "Restore /etc/hosts from /etc/hosts.orig"
cp /etc/hosts.orig /etc/hosts
echo ========================================================================

echo ========================================================================
echo "Download VMs list"
cd ~
wget -c "$host/mestrado/apache/dump.php" --
echo ========================================================================

echo ========================================================================
echo "Configure VMs list"
cat dump.php
cat dump.php >> /etc/hosts
rm /root/dump.php
echo ========================================================================

echo ========================================================================
echo "Download Nagios CFG files from VMs"
cd ~
rm -rf temp
mkdir temp
cd temp
wget -r --no-parent http://$host/mestrado/apache/cfg
cd $host
cd mestrado
cd apache
cd cfg
cp *.cfg /etc/nagios3/conf.d
cd /etc/nagios3/conf.d
chmod 755 *
cd ~
/etc/init.d/nagios3 restart
echo ========================================================================
```

Listing 15 – Nagios automated configuration hosts

After all hosts are correctly configured on Nagios, the Nagios web portal can display their updated statuses. Figure 19 is a screenshot of the Nagios running and monitoring robots.

Figure 19 - Nagios web Portal

5.1.3 Database Server

As previously defined, the database server collects the data that allows performance measurement in this experiment. There are two different servers configured on its Virtual Machine: The first one is a MySQL database server and an Apache HTTP Server.

The first one is a database that stores all VMs contact information and their status information. The data model for this information contains two tables. The VM table is described in Table 1 and the statuses table is decribed in Table 2.

Field name	Data type	Descrition
id	int(11)	A unique ID to identify the Virtual Machine.
hostname	varchar(255)	The hostname of the Virtual Machine. ROS requires a valid hostname configured to allow roscore remote connections.
ip	varchar(255)	The Virtual Machine IP address.
date	date	The date of the last VM contact.
time	time	The time of the last VM contact.

Table 1 - Database server table vms structure

Field name	Data type	Descrition
id	int(11)	A unique ID to identify the row.
hostname	varchar(255)	The VM hostname that send the status.
ip	varchar(255)	The VM current IP address.
date	date	The date of the status.
time	time	The time of the status.
CPU	Float	The current % of the CPU use.
Memory	Falta o tipo	The current % of the memory use.
Bandwidth	Falta o tipo	Total bytes of the Bandwidth transmitted (send and received).
Comment	Long text	An addition field used to facilitate group and organize all received data.

Table 2 - Database server table statuses structure

The second service running on this machine is the Apache HTTP server configured with PHP module enabled. It hosts PHP scripts that receive data from VMs through HTTP POST requests and handle it properly.

The add.php script receives a HTTP POST containing the information specified on the VM table and updates the database. If it is a new VM, it inserts a new row in the table. If it is an existent VM, it updates its entry.

The add script also facilitates the configuration robots on Nagios. It does so by outputting host entries that can be appended to the `/etc/hostname` file on the Nagios server. This ouput is presented in Listing 16. This configuration allows the Nagios machine to contact all VMs without a DNS server configured on the network.

```
IP                  Hostname
10.32.168.80        lsa-vm-simulator-10_32_168_80
10.32.168.81        lsa-vm-simulator-10_32_168_81
10.32.177.29        lsa-vm-simulator-10_32_177_29
10.32.177.31        lsa-vm-simulator-10_32_177_31
```

Listing 16 – Generated /etc/hostname file

The second step of this script is to automatically generate a Nagios configuration file on the `/etc/nagios3/conf` directory. This configuration file, showed in Listing 17, defines all information that should be monitored by Nagios for each specific host. When this file is copied to its directory and the Nagios service is restarted, robots are added on Nagios.

```
# A simple configuration file for monitoring the local host
# This can serve as an example for configuring other servers;
# Custom services specific to this host are added here, but services
# defined in nagios2-common_services.cfg may also apply.
define host{
        use                     generic-host            ; Name of host template to use
        host_name               lsa-vm-simulator-10_32_177_31
        alias                   lsa-vm-simulator-10_32_177_31
        address                 10.32.177.31
        }
# Define a service to check the robot sensors status
# All sensors
define service{
        use                             generic-service         ; Name of service template to use
        host_name                       lsa-vm-simulator-10_32_177_31
        service_description             Robot all sensors
        check_command                   check-all
        }
# Battery
define service{
        use                             generic-service         ; Name of service template to use
        host_name                       lsa-vm-simulator-10_32_177_31
        service_description             Robot / Laptop battery
        check_command                   check-battery
        }
```

Listing 17 – Generated Nagios host configuration file

The status.php is a script created to measure the VMs runtime information, this script receives a HTTP post containing all the statuses information and stores the data in the database. As previously stated, all the VMs in this experiment run this script every minute. This is done through a crontab task.

5.2 Experiment with Real Robot

The experiment with real robot uses a Kobuki-based Turtlebot robot equipped with a Microsoft Kinect as a depth sensor to avoid obstacles. Two Core i7 with 8GB memory laptops were used to run the application. The robot is programmed to perform autonomous navigation with collision avoidance.

The robot executes the ROS kobuki_node[1] to access/control the mobile base, the freenect_stack[2] to access the Kinect, and ROS move_base[3] navigation system. The kobuki_node locally produces diagnostic information from the mobile base in real-time.

The monitoring solution was used to periodically collect diagnostic data during the robot's operation. The implementation of the architecture was the same as the one described in the previous experiment. Due to that, the same server infrastructure was used.

[1] http://wiki.ros.org/kobuki_node
[2] http://wiki.ros.org/freenect_stack
[3] http://wiki.ros.org/move_base

6 EXPERIMENTAL RESULTS

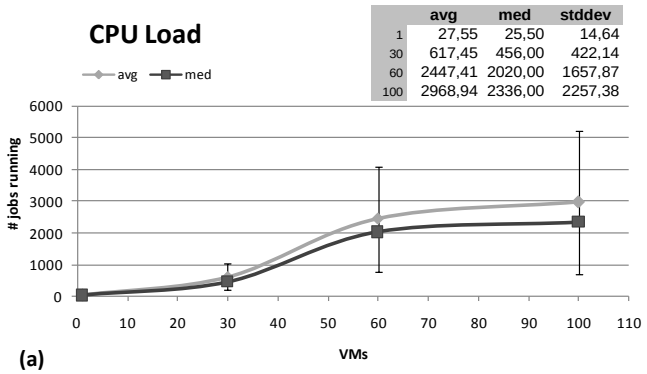
This section discusses the results obtained from the executed experiments.

6.1 Server Scalability

This experiment evaluates the scalability of the monitoring server as the number of monitored virtual robots increases up to 108. The monitoring server and the virtual robot configurations are described in Chapter 5.

We evaluate the Nagios server latency, which measures the delay to execute each check. Zero latency means that the server executed the checks at the exact time they were scheduled for execution. The acceptable check latency is typically lower than 15 seconds [NAG2005]. Larger latency indicates saturation of the Nagios server, meaning it is executing more checks than supported by the server computer. The latency was evaluated with 1, 30, 60, and 100 virtual robots connected to the monitoring server. The latency results range from 0.1 to 1 second for 1 to 60 virtual robots. With 100 virtual robots the average latency is 1.9 seconds, reaching the maximum value of 18.9 seconds in a single measurement.

Figure 20 shows the monitoring server's performance while the number of monitored virtual robots increases from 1, 30, 60, 100 instances. The performance parameters evaluated are CPU load, memory load, and network bandwidth. These parameters are collected during 100 minutes in 5 minutes intervals. For each of the 3 parameters, their average, median, and standard deviation are presented in the following charts. The same data is also displayed in tabular format in the top right corner of each chart.



	avg	med	stddev
1	27,55	25,50	14,64
30	617,45	456,00	422,14
60	2447,41	2020,00	1657,87
100	2968,94	2336,00	2257,38

(a)

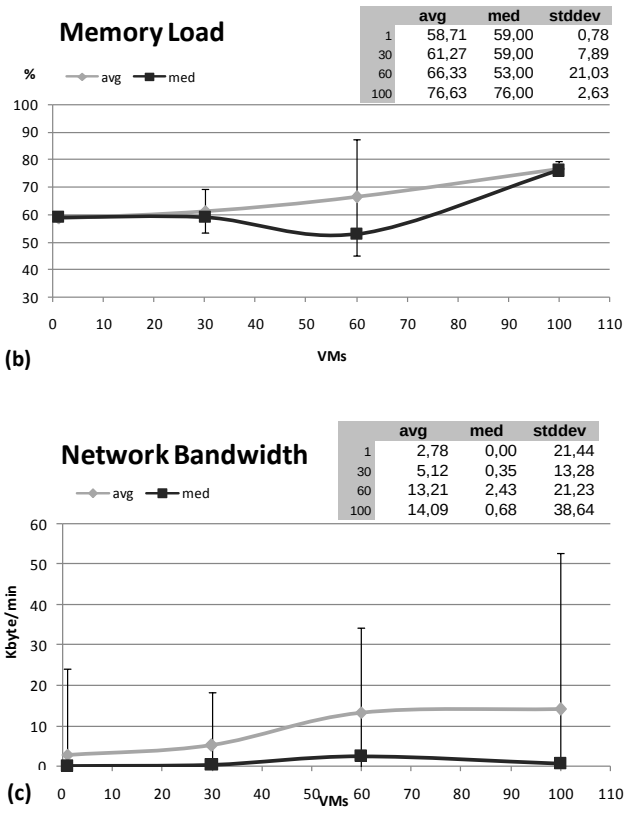
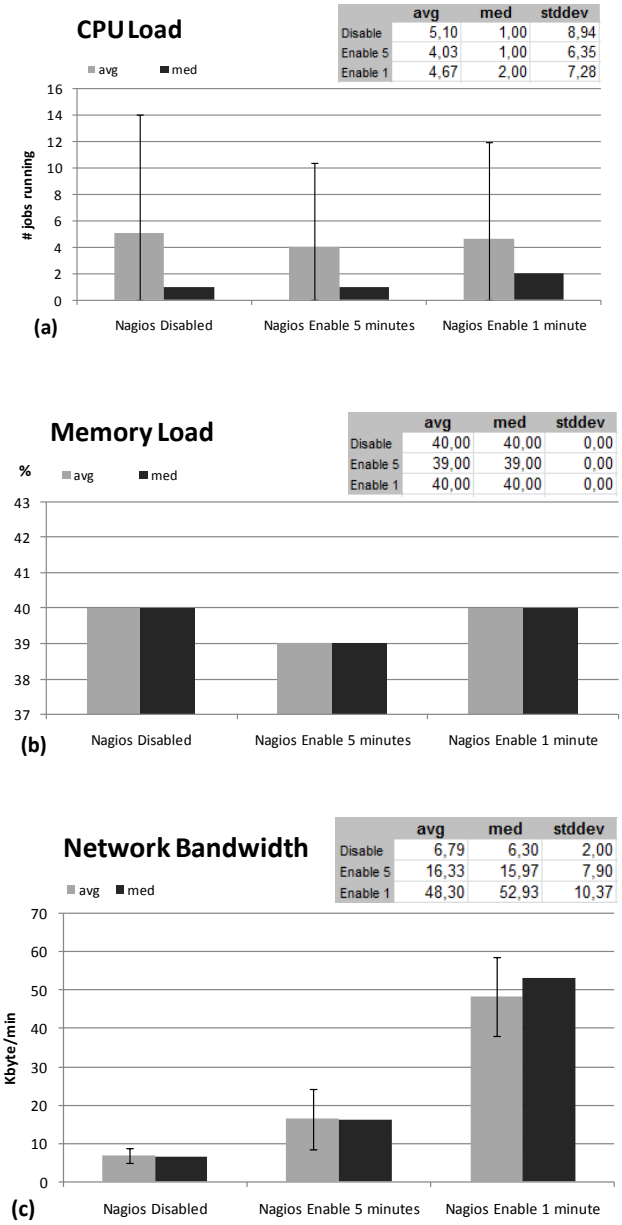


Figure 20 – CPU load (a) memory load (b) and network bandwidth (c) used at the monitoring server as the number of virtual robots increases

One can observe in Figure 20**Erro! Fonte de referência não encontrada.**(b) that the server's average memory load increased from about 60% to 80% as the number of virtual robot increased. The network bandwidth in Figure 20(c) increased from about 5 to 15 Kbytes/min. The server latency result with 100 virtual robots corroborates with CPU load results observed in Figure 20(a). One can observe that the CPU load is high (both *avg* and *stddev*), causing fluctuations in the latency measurement. These latency and CPU load results indicate that 100 robots are starting to saturate the minimalist server described in Section 5.1.2. On the other hand, it is reported that Nagios can monitor thousands of computers *[NMP2014]* when a normal fully-configured server is used instead of VMs. If we assume one computer per robot, it means that thousands of robots could be monitored.

6.2 Virtual Robot Performance

This experiment collects performance data at the robot side, in this case, from a virtual robot VM. Performance data (CPU load, memory load, network bandwidth) was collected during 60 hours with different monitoring frequencies (no monitoring, every 5 minutes, and every 1 minute). The observed results are illustrated in Figure 21, where the average, median, and standard deviation are presented in each chart.



	avg	med	stddev
Disable	5,10	1,00	8,94
Enable 5	4,03	1,00	6,35
Enable 1	4,67	2,00	7,28

	avg	med	stddev
Disable	40,00	40,00	0,00
Enable 5	39,00	39,00	0,00
Enable 1	40,00	40,00	0,00

	avg	med	stddev
Disable	6,79	6,30	2,00
Enable 5	16,33	15,97	7,90
Enable 1	48,30	52,93	10,37

Figure 21 – CPU load (a) memory load (b) and network bandwidth (c) used at the monitored robot

These results show that the monitoring process has a small impact on the monitored VMs, even considering the minimal resources allocated for each VM, as described in Chapter 5.

6.3 Real Robot Performance

This experiment evaluates the impact of the monitoring system on a real robot performing a common mobile robot software application for autonomous navigation. The robot's hardware and software configuration are described in Section 5.2.

Table 3 shows the performance data (CPU load, memory load, network bandwidth) collected during 30 minutes of navigation. First, data is collected when the monitoring server is off, then the same navigation task is repeated with the monitoring server on. As mentioned before, the proposed system required no additional software or specific configuration on the real robot.

Item	Monitor Status	Average	Median	Std dev
CPU Load	Monitor Off	123,83	125,50	23,28
	Monitor On	136,00	140,00	27,21
	% of change	9,83%	13,36%	16,86%
Memory Load	Monitor Off	42,00	42,00	0,00
	Monitor On	41,84	42,00	0,37
	% of change	-0,38%	0,00%	N/A
Network BW	Monitor Off	19,00	12,46	14,76
	Monitor On	26,56	21,80	19,16
	% of change	39,76%	74,89%	29,80%

Table 3 - CPU load (a) memory load (b) and network bandwidth (c) at the real robot with the monitoring on (every 5 min) and off

6.4 Limitations and Future Work

There are limitations in the usage of Nagios and ROS. One important limitation is that Nagios was orignally developed to monitor servers instead of robots. Due to this, Nagios requires a restart when a new host/robot is added or removed. This operation takes few seconds, depending on how many hosts are configured. This, however, is an implementation issue, not a limitation of the monitoring architecture.

Another important feature that was not explored in this work is Nagios support of asynchronous operations, which consists in listening for traps or interruptions instead of polling hosts. This feature would allow robots to instantly send a Nagios alerts and updates.

Nagios has a very flexible configuration stack, allowing for example the configuration of different polling intervals for each robot or for each sensor of an individual robot. This feature allows closer monitoring for more sensitive statuses. Another important feature not

explored by this work is Configuration Templates. This feature could be useful to monitor large sets of heterogeneous robots. Different templates could be defined for each kind of robot on a heterogeneous MRS. These templates can also be used to quickly add or remove robots on Nagios whitout redefining sensor rules.

Because of the simplicity of Nagios protocol (reads the standard output in a specific pre determined format) it is possible to implement a Nagios plugin to theoretically monitor any kind of device/robot on different robots middleware. The same instance of Nagios server could run multiple developed plugins allowing, for example, monitoring a MRS running different kind of robot middlewares.

As future work we intend to use the proposed approach as a supervisory system to monitor and to log events of one or multiple industrial robots. In the near future we intend to use this monitoring information to create a self-healing and autonomic MRS.

7 CONCLUSION

This work presented a lightweight and easy to configure monitoring infrastructure to monitor the status of a large number of different robots during runtime. The proposed approach integrates consolidated tools for IT monitoring (Nagios) and robotics (ROS), which proved to be very efficient for the robotic domain.

In terms of usability, the proposed approach requires no modification or additional software on the robot side. On the server side the proposed ROS plug-in must be installed and the server must be able to access the robots via their IP addresses. The source code for this plugin is available in the Appendix and can be used for any ROS compliant robot.

Experimental results show that it is possible to monitor 100 robots even with a minimally configured Nagios Server. It has been reported *[NMP2014]* that a fully configured Nagios Server could monitor thousands of computers. The results also show that the monitoring related overhead on real robots is negligible compared to the resources required to perform an autonomous navigation task.

Given the obtained results, it is possible to conclude this research achieves the goal of providing means to monitor faults in a MRS. The results also show that the proposed approach is easy to implement, has minimal impact on the robots, and provides a high usability overview of the MRS status.

REFERENCES

[AAV2001] A. Avizienis, J.-C. Laprie, B. Randell et al., Fundamental concepts of dependability. University of Newcastle upon Tyne, Computing Science, 2001.

[AAV2004] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” Dependable and Secure Computing, IEEE on, vol. 1, no. 1, pp. 11–33, 2004.

[ABA2008] A. Basu, M. Gallien, C. Lesire, and T. Nguyen, [Online] “Incremental component-based construction and verification of a robotic system,” ECAI, 2008.

[AMA2006] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for Robotics," In International Conference on Intelligent Robots and Systems (IROS), pp. 163-168, 2006.

[BLU2004] B. Lussier, R. Chatila, F. Ingrand, M.-O. Killijian, and D. Powell, “On fault tolerance and robustness in autonomous systems”, in Proceedings of the 3rdI ARP-IEEE/RASEURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments, 2004.

[BRG2009] B. Davide, P. Scandurra. "Component-based robotic engineering (part i)". Robotics & Automation Magazine, IEEE 16.4 (2009): 84-96, 2009.

[BRG2010] Brugali, Davide, and A. Shakhimardanov. "Component-based robotic engineering (part ii)." Robotics & Automation Magazine, IEEE 100-112, 2010.

[CHR2009] Christensen, A. Lyhne, R. O'Grady, and M. Dorigo. "From fireflies to fault-tolerant swarms of robots." Evolutionary Computation, IEEE on 13.4 (2009): 754-766, 2009.

[COL2012] C. Dave. "Data center infrastructure management." Data Center Knowledge, 2012.

[DAI2007] Daigle, Matthew J., Xenofon D. Koutsoukos, and Gautam Biswas. "Distributed diagnosis in formations of mobile robots." Robotics, IEEE 353-369, 2007.

[ELK2012] E. Ayssam, T. Sobh. "Robotics middleware: a comprehensive literature survey and attribute-based bibliography." Journal of Robotics, 2012.

[GAN2013] Ganglia Website. [Online] Available from: http://ganglia.sourceforge.net, 2013.

[GDU2010] G. Dudek and M. Jenkin, Computational principles of mobile robotics. Cambridge university press, 2010.

[GP12014] G. Patrick. “ROS By Example HYDRO - Volume 1”. P. 214, 2014.

[GP22014] G. Patrick. “ROS By Example HYDRO - Volume 2. Packages and Programs for Advanced Robot Behaviors”. 2014.

[HUT2002] H. Utz, S. Sablatng, S. Enderle, G. Kraetzschmar, "Miro- Middleware for Mobile Robot Applications," IEEE 493-497, 2002.

[JCA2003] J. Carlson and R. R. Murphy, “Reliability analysis of mobile robots,” in Robotics and Automation, 2003. Proceedings. ICRA’03. IEEE 274–281, 2003.

[JCA2005] J. Carlson and R. R. Murphy, “How ugvs physically fail in the field,” Robotics, IEEE Transactions on, vol. 21, no. 3, pp. 423–437, 2005.

[KAM2002] G. Kaminka, D. V. Pynadath, and M. Tambe, “Monitoring teams by overhearing: A multi-agent plan-recognition approach,” Journal of Artificial Intelligence Research, no. 17, pp. 83–135, 2002.

[KBK2013] iClebo Kobuki web site. [Online] Available from: http://kobuki.yujinrobot.com, 2013.

[KBL2006] Kannan, Balajee, and Lynne E. Parker. "Fault-tolerance based metrics for evaluating system performance in multi-robot teams." Proceedings of Performance Metrics for Intelligent Systems Workshop, 2006.

[LEP2008] L. E. Parker, “Multiple mobile robot systems,” Springer Handbook of Robotics, pp. 921–941, 2008.

[LEP2012] L. E. Parker, “Reliability and fault tolerance in collective robot systems,” Handbook on Collective Robotics: Fundamentals and Challenges, 2012.

[LOT2011] Lotz, Alex, Andreas Steck, and Christian Schlegel. "Runtime monitoring of robotics software components: Increasing robustness of service robotic

systems." Advanced Robotics (ICAR), 2011 15th International Conference on. IEEE, 2011.

[MAK2007] Makarenko, Alexei, Alex Brooks, and Tobias Kaupp. "On the benefits of making robotic software frameworks thin." International Conference on Intelligent Robots and Systems. Vol. 2, 2007.

[MEN2010] Mendes, Mário JGC, and J. da Costa. "A multi-agent approach to a networked fault detection system." IEEE. Control and Fault-Tolerant Systems (SysTol), 2010.

[MHA2003] M. Hashimoto, H. Kawashima, and F. Oba, “A multi-model based fault detection and diagnosis of internal sensors for mobile robot,” in Intelligent Robots and Systems, 2003. Proceedings. 2003 IEEE/RSJ International Conference on, vol. 4. IEEE, pp. 3787–3792, 2003.

[MIR2013] Miro – Middleware for Robots Website. [Online] Available from: http://www.ohloh.net/p/miro-middleware, 2013.

[MJM1995] M. J. Mataric, M. Nilsson, and K. Simsarin, “Cooperative multi-robot box-pushing,” in Intelligent Robots and Systems 95.’Human Robot Interaction and Cooperative Robots’, Proceedings. 1995 IEEE/RSJ International Conference on, vol. 3. IEEE, pp. 556–561, 1995.

[MLL1998] M. L. Leuschen, I. D. Walker, and J. R. Cavallaro, “Robot reliability through fuzzy markov models” in Reliability and Maintainability Symposium, 1998. Proceedings., Annual. IEEE, pp. 209–214, 1998.

[MOH2008] Mohamed, Nader, Jameela Al-Jaroodi, and Imad Jawhar. “Middleware for robotics: A survey Robotics, Automation and Mechatronics”, 2008 IEEE Conference on IEEE, 2008.

[MOH2009] Mohan, Yogeswaran, and S. G. Ponnambalam. "An extensive review of research in swarm robotics." Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on. IEEE, 2009.

[MQU2009] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in ICRA workshop on open source software, vol. 3, no. 3.2, 2009.

[NAG2005] Barth, Wolfgang. Nagios - System and Network Monitorin, 2005.

[NAG2013] Nagios, “Nagios - the industry standard in it infrastructure monitoring, 2013. Available from: http://nagios.org,” [Online]. Available: http://nagios.org, 2013.

[NMP2014] Maximizing Performance In Nagios. [Online] Available from: http://assets.nagios.com/downloads/nagiosxi/docs/Maximizing-Performance-In-Nagios-

[ORC2013] Orca: Components for Robotics Website. [Online] Available from http://orca-robotics.sourceforge.net/. 2013.

[PAR2010] P. A. R. Fagundes, “Plataforma de controlo e simulacao robotica,” 2010.

[RCA1993] R. C. Arkin, T. Balch, and E. Nitz, “Communication of behavorial state in multi-agent retrieval tasks,” in Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on. IEEE, pp. 588–594, 1993.

[RCA2003] R. Canham, A. H. Jackson, and A. Tyrrell, “Robot error detection using an artificial immune system,” in Evolvable Hardware, 2003. Proceedings. NASA/DoD Conference on. IEEE, pp. 199–207, 2003.

[RHB2007] R. H. Bordini, J. F. H¨ubner, and M. Wooldridge, Programming multi-agent systems in AgentSpeakusingJason. Wiley. com, vol. 8, 2007.

[ROD2014] Diagnostic System for Robots Running ROS. [Online] Available from: http://www.ros.org/reps/rep-0107.html, 2014.

[ROG2006] L. D. Rogério. Adaptability and Fault Tolerance. University of Kent, UK. 2006.

[ROS2014] ROS Website. [Online] Available from: http://www.ros.org, 2014.

[RSH2013] Kirchner, Dominik, Stefan Niemczyk, and Kurt Geihs. "RoSHA: A Multi-Robot Self-Healing Architecture⋆." 17th RoboCup International Symposium, Eindhoven, Netherlands. 2013.

[RSI2004] R. Siegwart and I. R. Nourbakhsh, Introduction to Autonomous Mobile Robots. The MIT press, 2004.

[SAH2006] S. Ahn, J. Lee, K. Lim, H. Ko, Y. Kwon, and H. Kim, "Requirements to UPnP for Robot Middleware," in Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), Oct, 2006.

[SEN2001] S. Enderle, H. Utz, S. Sablatng, S. Simon, G. Kraetzschmar, and G. Palm, "Miro: Middleware for autonomous mobile robots," IFAC Conference on Telematics Applications in Automation and Robotics, 2001.

[SKJ2006] S. Ahn, K. Lim, J. Lee, H. Ko, Y. Kwon and H. Kim, "UPnP Robot Middleware for Ubiquitous Robot Control," The 3rd International Conference on Ubiquitous Robots and Ambient Intelligence (URAI 2006), 2006.

[SPI2013] Spirceworks Website. [Online] Available from: http://www.spiceworks.com/ 2013.

[SVE2005] S. Verret, “Current state of the art in multirobot systems,” Defence Research and Development Canada-Suffield, 2005.

[UPN2013] UPnP Website. [Online] Available from: http://www.upnp.org, 2013.

[VGO2004] V. Goldmanand S. Zilberstein, “Decentralized control of cooperative systems: Categorization and complexity analysis,” J. Artif. Intell. Res.(JAIR), vol. 22, pp. 143–174, 2004.

[ZAB2013] Zabbix Website. [Online] Available from: http://www.zabbix.com, 2013.

APPENDIX

7.1 Nagios installation steps

The detailed steps to download, install and configured Nagios could be accessed on *[NAG2005].* Running Linux Ubuntu 13.04 the apt-get package management system can install and configure in a few steps:

Executed as root:

Install packages Listing 18:

```
# sudo apt-get install -y nagios3
```

Listing 18 – Nagios instalation command on Ubuntu

This command will download, install and pre-configured all Nagios requirements for work like Apache HTTP server, MySQL database and libraries.

Set admin password, Listing 19:

```
# sudo htpasswd -c /etc/nagios3/htpasswd.users nagiosadmin
```

Listing 19 – Nagios set administrator password

Start the service, see Listing 20:

```
# sudo /etc/init.d/nagios start
```

Listing 20 – Start Nagios service

7.2 Nagios configuration steps

On Ubunto 13.04 the initial verion of Nagios comes with a defect that generate a alert, this is a filled bug *#615848 – "You can either give the nagios user permission to that file or just ignore the file during check".* Search for this bug on the Nagios website for more details.

7.3 ROS configuration Steps

A detailed page on how to install and configure ROS on Ubuntu could be found at:

http://wiki.ros.org/hydro/Installation/Ubuntu

Another source explains step by step detailed on how to install and configure ROS is *[GP12014]* and *[GP22014]*.

After ROS installation steps is completed successfully, configure environment variables Listing 21:

```
source /opt/ros/<distro>/setup.bash
```

Listing 21 – Configure ROS environment

Create a ROS Workspace, see Listing 22:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ~/catkin_ws/
catkin_make
source devel/setup.bash
```

Listing 22 – ROS create workspace

7.4 Kobuki Turtlebot installation

On Ubuntu 12.04 execute the steps, see Listing 23:

```
apt-get install ros-hydro-turtlebot*
apt-get install ros-hydro-kobuki*
run only at first time
rosrun kobuki_ftdi create_udev_rules
```

Listing 23 – Kobuki Turtlebot installation steps

Steps to execute Kobuki, see Listing 24:

```
# Connect the robot on the USB and run:
roslaunch turtlebot_bringup minimal.launch
# Keyboard robot control:
roslaunch turtlebot_teleop keyboard_teleop.launch
# GUI
rosrun rqt_runtime_monitor rqt_runtime_monitor
```

Listing 24 – Steps to run Kobuki Turtlebot

7.5 ROS Diagnostics Nagios plugin source code

The source code of the ROS Diagnostics Nagios plugin developed in Python to integrate Nagios with ROS Diagnostics.

```
#!/usr/bin/env python

import sys
sys.path.append("/opt/ros/hydro/lib/python2.7/dist-packages")
```

```
import os
os.environ['PATH'] = "/opt/ros/hydro/bin:" + os.environ['PATH']

from optparse import OptionParser

import rospy
import rosnode
import os
import roslib
import sys
roslib.load_manifest('linux_hardware')
from linux_hardware.msg import LaptopChargeStatus
from diagnostic_msgs.msg import DiagnosticStatus, DiagnosticArray, KeyValue

# Exit statuses recognized by Nagios
UNKNOWN = -1
OK = 0
WARNING = 1
CRITICAL = 2
STALE = 3
COUNT_sensors = 0

# TEMPLATE FOR READING PARAMETERS FROM COMMANDLINE
parser = OptionParser()
parser.add_option("-H", "--host", dest="host", default='localhost', help="Define the target host")
parser.add_option("-N", "--name", dest="name", default='all', help="Define the sensor name")
(options, args) = parser.parse_args()

# Set turtlebot ROS Master URI
os.environ['ROS_MASTER_URI'] = 'http://' + options.host  + ':11311'

total_level = None
OK_sensors = None
WARNING_sensors = None
CRITICAL_sensors = None
STALE_sensors = None


def callback(data):
    global total_level
    global OK_sensors
    global WARNING_sensors
    global CRITICAL_sensors
    global STALE_sensors
    global COUNT_sensors

    ready = False

    while not ready:
        for current in data.status:
          # Debug all information
          #from pprint import pprint
          #pprint(options.host)

          # Filter if name is received

          if (options.name == "all" or options.name in current.name):

              # Calculate the total level
              if current.level >= total_level:
                total_level = current.level

              # Parse current.name string and keep only the part after the :
              parse_name = current.name

              # Count how many sensors were found
              COUNT_sensors = COUNT_sensors + 1

              # Create CRITICAL sensors list
              if current.level == CRITICAL:
                  if CRITICAL_sensors != None:
                    CRITICAL_sensors = str(CRITICAL_sensors) + str(parse_name) + str(', ')
                  else:
                    CRITICAL_sensors = str(parse_name) + str(', ')

              # Create WARNING sensors list
```

```
                if current.level == WARNING:
                    if WARNING_sensors != None:
                      WARNING_sensors = str(WARNING_sensors) + str(parse_name) + str(', ')
                    else:
                      WARNING_sensors = str(parse_name) + str(', ')

                # Create OK sensors list
                if current.level == OK:
                    if OK_sensors != None:
                      OK_sensors = str(OK_sensors) + str(parse_name) + str(', ')
                    else:
                      OK_sensors = str(parse_name) + str(', ')

                # Create STALE sensors list
                if current.level == STALE:
                    if STALE_sensors != None:
                      STALE_sensors = str(STALE_sensors) + str(parse_name) + str(', ')
                    else:
                      STALE_sensors = str(parse_name) + str(', ')

            ready = True

    time = rospy.get_time()
    #percentage = int(float(percentage))
    rospy.signal_shutdown(0)

def listener():
    rospy.init_node('ros_diagnostics', anonymous=True,  disable_signals=True)
    rospy.Subscriber("diagnostics_agg", DiagnosticArray , callback)
    rospy.spin()

def myhook():

    description = None

    if STALE_sensors != None:
        if (description != None):
          description = str(description) + str("STAKE sensor(s) list: " + str(STALE_sensors))
        else:
          description = str("STALE sensor(s) list: " + str(STALE_sensors))

    if CRITICAL_sensors != None:
        if (description != None):
          description = str(description) + str("CRITICAL sensor(s) list: " + str(CRITICAL_sensors))
        else:
          description = "CRITICAL sensor(s) list: " + str(CRITICAL_sensors)

    if WARNING_sensors != None:
        if (description != None):
          description = str(description) + str("WARNING sensor(s) list: " + str(WARNING_sensors))
        else:
          description = str("WARNING sensor(s) list: " + str(WARNING_sensors))

    if OK_sensors != None:
        if (description != None):
          description = str(description) + str("OK sensor(s) list: " + str(OK_sensors))
        else:
          description = str("OK sensor(s) list: " + str(OK_sensors))

    # Remove last comma
    if description != None:
      description = description[:-2]

    if COUNT_sensors == 0:
      print "CRITICAL - %s" % (description)
      exiting(CRITICAL)
    if total_level >= CRITICAL:
      print "CRITICAL - %s" % (description)
      exiting(CRITICAL)
    elif total_level == WARNING:
      print "WARNING - %s" % (description)
      exiting(WARNING)
    else:
      print "OK - %s" % (description)
      exiting(OK)
```

```
def exiting(value):
    try:
         sys.stdout.flush()
         os._exit(value)
    except:
        pass

if __name__ == '__main__':
    try:
        master = rospy.get_master()
        master.getPid()
    except Exception:
        print "UNKNOWN - Roscore not available"
         exiting(UNKNOWN)

    try:
        if len(sys.argv) < 1:
            print "usage %s -N <name of sensor>" % (sys.argv[0])
             exiting(UNKNOWN)
        rospy.on_shutdown(myhook)
        listener()
    except rospy.ROSInterruptException:
        exit
```

Listing 25 - ROS Diagnostics Nagios plugin source code

7.6 Virtual Robot source code

This source code is a ROS node developed in Python to generate randon diagnostic information based on pre-defined rules and publish the generated information on ROS diagnostics topic.

The ROS Diagnostic aggregator node (distributed with ROS) reads this information and based on a XML rules file group and publish the information on the ROS Diagnostic agg topic. Once the information is published on diagnostic_agg topic it is avaiable to be accessed from Nagios through the ROS Diagnostics Nagios plugin.

```
#!/usr/bin/env python

import roslib; # roslib.load_manifest('pr2_motors_analyzer')

import rospy, random, md5
from diagnostic_msgs.msg import DiagnosticArray, DiagnosticStatus, KeyValue

##########################################################################
# Configuration
##########################################################################

##########################################################################
# Simulation config
total_errors = 0
total_warnings = 0

##########################################################################
# Refresh interval in seconds
refresh_interval = 300


##########################################################################
# Robot configuration
##########################################################################

##########################################################################
# Motors config
number_of_motors = 3

##########################################################################
```

```
# Temperature config
number_of_sensors = 2

##########################################################################
# Laser config
number_of_lasers = 2


##########################################################################
# Cameras config
number_of_cameras = 1


##########################################################################
# Initial runtime setup
##########################################################################

##########################################################################
# Battery config
# The intial battery level - 100 is considered full charge
initial_battery_level = 100

# Error battery level for diagnostics
error_battery_level = 10

# Warn battery level for diagnostics
warn_battery_level = 20

# Runtime error control
current_errors = 0
current_warnings = 0

def motor(msg):
          global current_errors
          global current_warnings
          for cont in range(1,number_of_motors + 1):
                    status = DiagnosticStatus()
                    status.name = "Motor " + str(cont)
                    status.hardware_id = md5.new(str(status.name)).hexdigest()
                    random.seed()
                    level = random.randint(0, 100)
                    #print level
                    if level < 95:
                              status.message = 'Running'
                              status.level = DiagnosticStatus.OK
                    elif level < 99:
                              status.message = 'Warning'
                              status.level = DiagnosticStatus.WARN
                              current_warnings += 1
                    else:
                              status.message = 'Stopped'
                              status.level = DiagnosticStatus.ERROR
                              current_errors += 1
                    msg.status.append(status)
          return msg

def camera(msg):
          global current_errors
          global current_warnings
          for cont in range(1,number_of_cameras + 1):
                    status = DiagnosticStatus()
                    status.name = "Cam " + str(cont)
                    status.hardware_id = md5.new(str(status.name)).hexdigest()
                    random.seed()
                    level = random.randint(0, 100)
                    #print level
                    if level < 95:
                              status.message = 'OK'
                              status.level = DiagnosticStatus.OK
                    elif level < 99:
                              status.message = 'Warning'
                              status.level = DiagnosticStatus.WARN
                              current_warnings += 1
                    else:
                              status.message = 'Error'
                              status.level = DiagnosticStatus.ERROR
```

```
                        current_errors += 1
                msg.status.append(status)
        return msg


def temperature(msg):
        global current_errors
        global current_warnings
        for cont in range(1,number_of_sensors + 1):
                status = DiagnosticStatus()
                status.name = "Sensor " + str(cont)
                status.hardware_id = md5.new(str(status.name)).hexdigest()
                random.seed()
                level = random.randint(0, 100)
                #print level
                if level < 95:
                        temp = random.randint(15, 50)
                        status.message = str(temp) + ' degrees'
                        status.level = DiagnosticStatus.OK
                elif level < 99:
                        temp = random.randint(51, 80)
                        status.message = str(temp) + ' degrees'
                        status.level = DiagnosticStatus.WARN
                        current_warnings += 1
                else:
                        temp = random.randint(80, 99)
                        status.message = str(temp) + ' degrees'
                        status.level = DiagnosticStatus.ERROR
                        current_errors += 1
                msg.status.append(status)
        return msg

def laser(msg):
        global current_errors
        global current_warnings
        for cont in range(1,number_of_lasers + 1):
                status = DiagnosticStatus()
                status.name = "Laser " + str(cont)
                status.hardware_id = md5.new(str(status.name)).hexdigest()
                random.seed()
                level = random.randint(0, 100)
                #print level
                if level < 95:
                        status.message = 'Normal'
                        status.level = DiagnosticStatus.OK
                elif level < 99:
                        status.message = 'Warning'
                        status.level = DiagnosticStatus.WARN
                        current_warnings += 1
                else:
                        status.message = 'Error'
                        status.level = DiagnosticStatus.ERROR
                        current_errors += 1
                msg.status.append(status)
        return msg


def battery(msg):
        global current_errors
        global current_warnings
        # Initialize the diagnostics status
        status = DiagnosticStatus()
        status.name = "Robot Battery"

        current_battery_level = random.randint(1, 100)

        if current_battery_level < error_battery_level:
                status.message = "Low " + str(current_battery_level) + "%"
                status.level = DiagnosticStatus.ERROR
                current_errors += 1
        elif current_battery_level < warn_battery_level:
                status.message = "Warning " + str(current_battery_level) + "%"
                status.level = DiagnosticStatus.WARN
                current_warnings += 1
        else:
                status.message = "OK " + str(current_battery_level) + "%"
```

```
                    status.level = DiagnosticStatus.OK

          # Add the raw battery level to the diagnostics message
          status.values.append(KeyValue("Level", str(current_battery_level)))
          msg.status.append(status)
          return msg

def laptop_battery(msg):
          global current_errors
          global current_warnings
          # Initialize the diagnostics status
          status = DiagnosticStatus()
          status.name = "Laptop Battery"

          current_battery_level = random.randint(1, 100)

          if current_battery_level < error_battery_level:
                    status.message = "Low " + str(current_battery_level) + "%"
                    status.level = DiagnosticStatus.ERROR
                    current_errors += 1
          elif current_battery_level < warn_battery_level:
                    status.message = "Warning " + str(current_battery_level) + "%"
                    status.level = DiagnosticStatus.WARN
                    current_warnings += 1
          else:
                    status.message = "OK " + str(current_battery_level) + "%"
                    status.level = DiagnosticStatus.OK

          # Add the raw battery level to the diagnostics message
          status.values.append(KeyValue("Level", str(current_battery_level)))
          msg.status.append(status)
          return msg

def generate_values_internal(msg):
          global current_errors
          global current_warnings

          msg.header.stamp = rospy.Time.now()
          msg = motor(msg)
          msg = battery(msg)
          msg = temperature(msg)
          msg = laser(msg)
          msg = camera(msg)
          msg = laptop_battery(msg)
          return msg

def generate_values():
          global total_errors
          global total_warnings
          global current_errors
          global current_warnings

          msg = DiagnosticArray()
          loop2 = 0

          while True:
                    loop2 = loop2 + 1
                    current_errors = 0
                    current_warnings = 0
                    generate_values_internal(msg)
                    print '################################################'
                    print 'Generate_values runtime count ' + str(loop2)
                    #print 'total_errors = ' + str(total_errors)
                    #print 'current_errors = ' + str(current_errors)
                    #print 'total_warnings = ' + str(total_warnings)
                    #print 'current_warnings = ' + str(current_warnings)
                    print '################################################'

                    # Avoid stuck the simulator
                    if (loop2 > 100000):
                              break

                    if (total_warnings == current_warnings):
                              if (total_errors == current_errors):
                                        #print "match"
                                        break
```

```
          return msg

def update_config():
          global total_errors
          global total_warnings
          global refresh_interval
          global number_of_motors
          global number_of_sensors
          global number_of_lasers
          global number_of_cameras
          total_errors = 0
          total_warnings = 0
          return False


if __name__ == '__main__':
          # Create initial values
          rospy.init_node('simulator3')
          pub = rospy.Publisher('/diagnostics', DiagnosticArray)
          loop = 0
          my_rate = rospy.Rate(1)
          update_config()

          while not rospy.is_shutdown():
                    #print "loop = " + str(loop)
                    #print "refresh_interval = " + str(refresh_interval)
                    if ((loop % refresh_interval) == 0):
                              print '################################################'
                              print 'Refresh interval ' + str(refresh_interval) + ' seconds...'
                              msg = generate_values()
                              #print msg
                              print '################################################'

                    # Check if the config was changed, if yes force reload sensor values
                    if (update_config() == True):
                              print '################################################'
                              print 'Simulator configuration updated sucessfully'
                              print '################################################'
                              msg = generate_values()

                    pub.publish(msg)
                    loop = loop + 1
                    my_rate.sleep()
```

Listing 26 – Virtual Robot source code