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

Research working in progress report requirement for obtaining the degree of Master in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Dr. Alexandre de Morais Amory

Porto Alegre

2014

INDEX

Image Index

Table Index

LIST OF ABBREVIATIONS AND ACRONYMS

API – Application Protocol Interface

CLI - Command Line Interface

DCIM - Data Center Infrastructure Management

HTTP - Hypertext Transfer Protocol

MRS - Multiple Robots Systems

NRPE - Nagios Remote Plug-in Executor

P2P – Peer to Peer

POC - Proof of concept

PUCRS – Pontifical Catholic University of Rio Grande do Sul

ROS - Robot Operating System

SRS - Single-Robot Systems

TCP/IP - Transmission Control Protocol and Internet Protocol

XML - Extensible Markup Language

1 Introduction

Robotics started at the beginning of XX century with the needs to improve the productivity and quality of manufacture products. Nowadays robotics becomes more common and people start to use more robotics to help and accomplish a variety of tasks such as robot arm, automated guided vehicles, unmanned aerial vehicles, humanoid robots and others. Robotics is typically used to execute dangerous tasks, unhealthy tasks, places where it is not possible to have human control, remote areas where is too hard or too expansive to send a human, or even for tasks that demands a too big workload that a human is not able to accomplish.

Basically there are two different kinds of robots: stationary robots and mobile robots. Stationary robots are simpler than mobile because they are fixed at some controlled environment. It's 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 handles the control of autonomous vehicle or semi-autonomous vehicle *[GDU2010]* and *[RSI2004]*. Currently, there are few commercial applications of mobile service robots: goods transportation, surveillance, inspection, cleaning or household robots, lawn mowing, pool cleaning are just some examples. 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.

One cost-effective way to provide effectiveness and robustness to robotic system is to use multi-robots instead of a single robot. Multi-robot systems (MRS) have some

advantages over single-robots systems, these advantages include increased of speed for task completion through parallelism and also can increased of robustness and reliability. MRS can implement 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 system. For instance, MRS are more complex to manage and coordinate the collective system, require increased communication capabilities in order to coordinate all robots, and they are more complex to determine its global state and to debug the system due to its distributed nature.

MRS can be classified as homogeneous or heterogeneous *[SVE2005]*. Homogeneous MRS means that all members of the team have the same specification (hardware and software configuration). Heterogeneous MRS can have different kind of robots in the same team. Homogeneous MRS is easier to replace a faulty robot. On other hand, the advantage of heterogeneous MRS is to support different kind of specialized and simpler robots, compared to a robot that does several different tasks.

Robotic systems can also be classified according to its 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 (fully autonomous robots are currently not feasible for reasonably complex applications). A robot with some level of autonomy can be called an agent. Multi-agents systems are commonly used to implement MRS with autonomy.

1.1 Motivations and Goals

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 are naturally more robust than single robots due its intrinsic redundancy, but it increases the software complexity due to its distributed nature. The *goal of this work* is to provide means to easily monitor faults at a team of heterogeneous robotic agents. The detection and isolation the defective agent is a first step toward an adaptive MRS which can execute the desired task even in the presence of faults.

With more dependable robotic systems, more applications can be created to serve the society.

1.2 Organization

The Section 2 presents the theoretical background necessary to understand this work, such as, the autonomous agents concepts, dependability concepts, and MRS. Section 3 describes the state of the art in terms of individual robots fault detection and MRS fault detection. Section 0 specifies the research proposal, its activities, and schedule.

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 receiving 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 maintain 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 percept its environment through its sensors and it is able to cause some effects on the environment via its actuators. See the *Image 1 - Agent interaction with the environment [RHB2007]*.

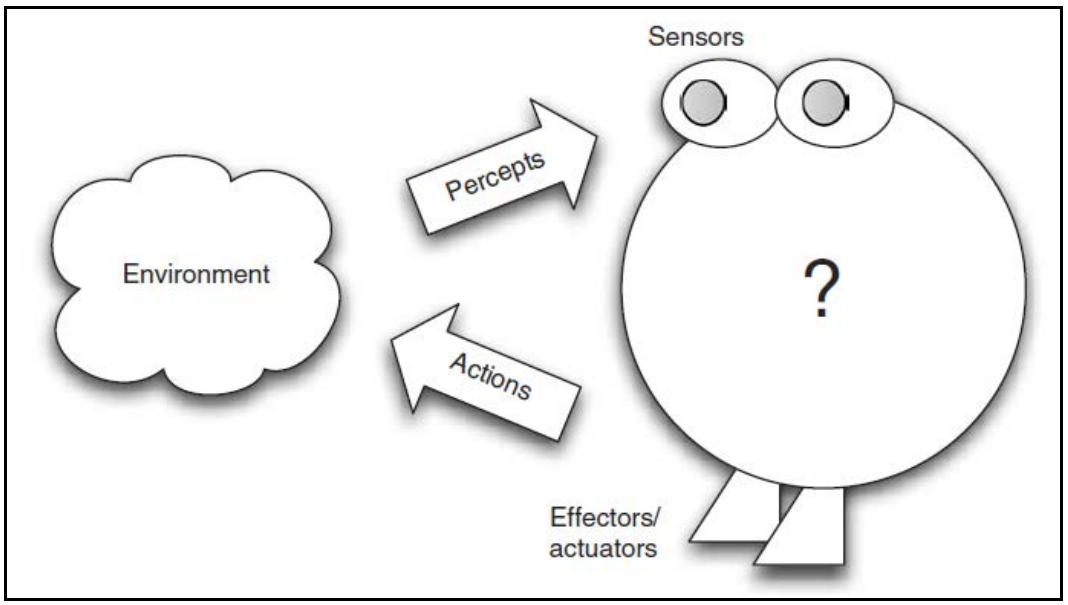

Image 1 - Agent interaction with the environment *[RHB2007]*

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, in general, completely control the environment. For example, a robot built to lawn-mower could stuck on a hole and not be able to finish its work. The real environment is dynamic and cannot be controlled so even the highly tested robots will face some unforeseen situations and fail.

There are some important features expected from agents:

- Autonomy: For agents autonomy means that agents have capacity to operate independently. They are able to figure out and execute a determined plan to achieve their goal.
- Pro-activeness: When an agent has been delegated to do a particular goal, the agent needs be able to act according to his goal-directed behavior.
- Re-activeness: Be responsive to the environment changes.
- Social Ability: Instead of simple exchange of bytes and messages, for agents, social ability means to be able to cooperate and to coordinate efforts in order to achieve their goals.

2.1.2 Multi-Agent Systems

Agents inhabit an environment that others agents occupy and each one of these agents have an impact in this environment. It is possible that one agent has control of only part of its environment, but often there are overlapping between the impacts of different agents into the environment, generating more complex scenarios. The *Image 2 - Typical structure of a multi-agent system [RHB2007]* shows a multi-agent system interacting in the same environment.

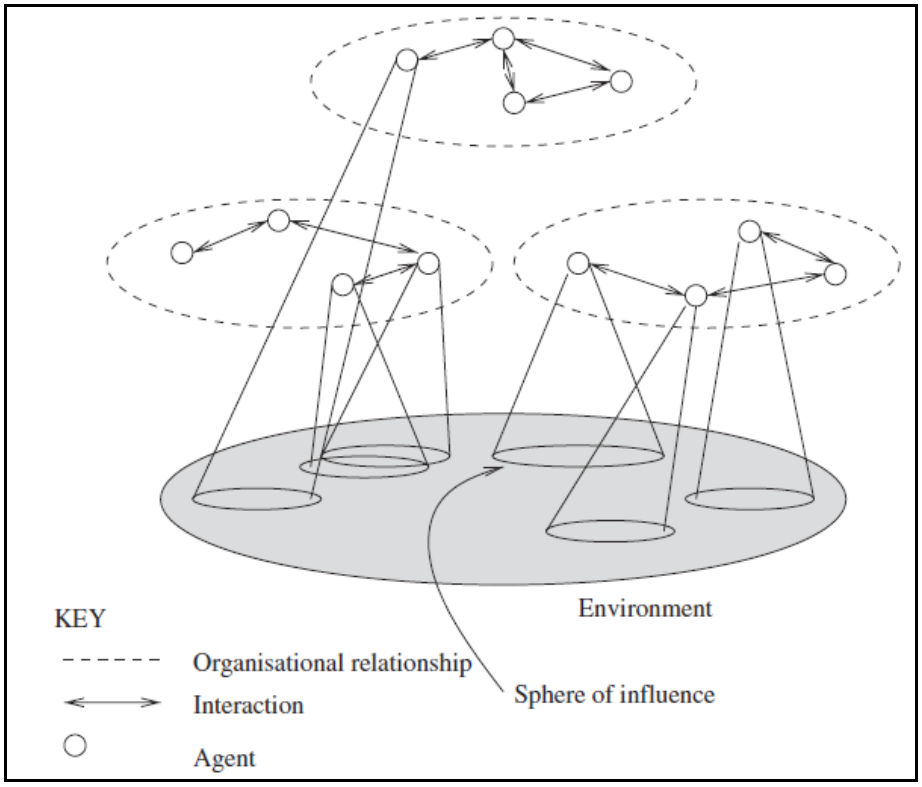

Image 2 - Typical structure of a multi-agent system *[RHB2007]*

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 Image 3 demonstrates these concepts.

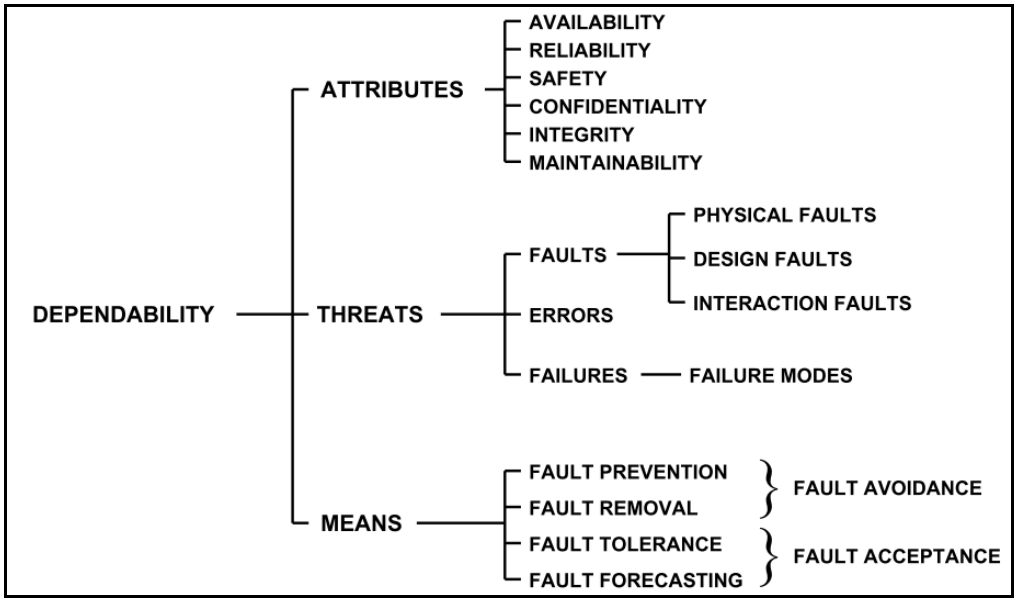

Image 3 - The dependability concepts *[BLU2004]*

2.2.1 Attributes

The dependability attributes could be classified as:

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

2.2.2 Threats

In this section, we present the taxonomy of threats that may affect a system during its entire life. The life cycle of a system consists of two phases: development and use *[AAV2001]*. The development phase contain all activities from the initial concept presentation, passing by the development itself until the final test phases that shows that the system is ready to deliver the service to the user. During this phase of development, defects or bugs could be introduced by the lack of knowledge of the development team, complexity of the system or even for malicious objectives.

The threats in a system consist in failures, errors and faults. System failures are an event that deviates the delivery of correct service. An error is the part of the system state that may cause a failure. A failure occurs when an error reaches the service interface. A fault is the cause of error. A fault is active when it produces an error, otherwise, it is a dormant fault. A system can fail in different ways. There are three different taxonomies for faults *[BLU2004]* as we show in the Image 4.

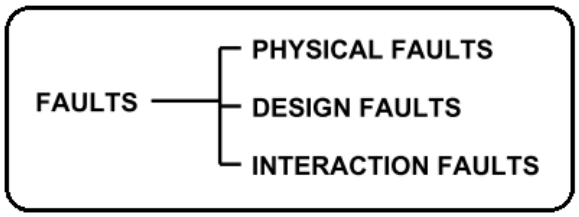

Image 4 - A fault taxonomy *[BLU2004]*

2.2.2.1 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.

2.2.2.2 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.

2.2.2.3 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.

2.2.3 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 below:

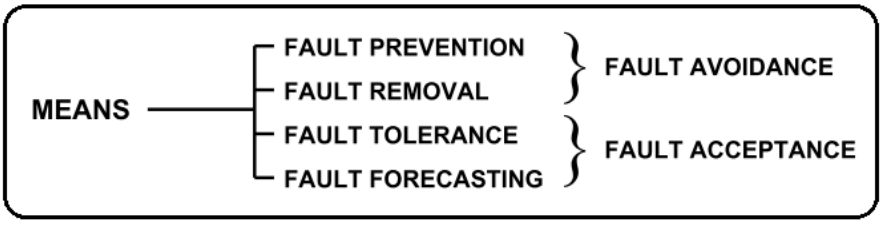

Image 5 - Means - Fault remove techniques *[BLU2004]*

2.2.3.1 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.

2.2.3.2 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.

2.2.3.3 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. This work focuses on fault tolerance mechanisms.

2.2.3.4 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.

2.2.4 Fault Tolerance

Fault tolerance mechanisms typically consist of an error detection and error recovering mechanisms *[LUS2004]*, as illustrated in *Image 6 – Fault Tolerance Techniques [LUS2004]*.

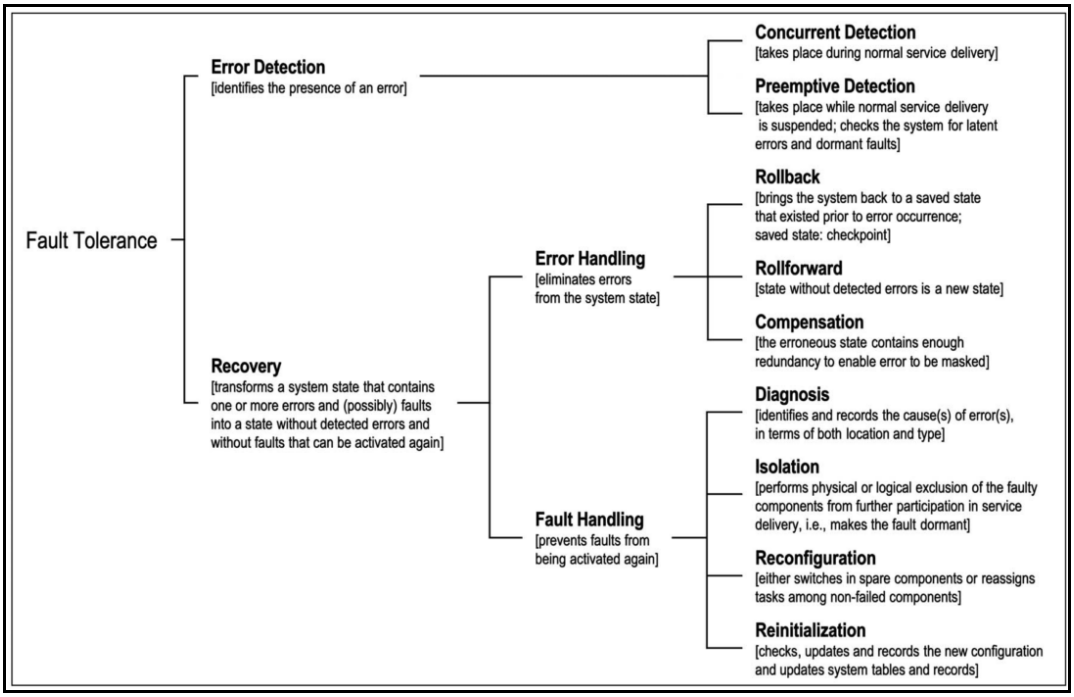

Image 6 – Fault Tolerance Techniques *[AAV2004]*

2.2.4.1 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 during 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.

2.2.4.2 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 activate 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.

2.2.4.3 Fault handling

Summon *[ROG2006]*, Fault handling is a technique that 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)

A Multiple Robots Systems (or, equivalently Collective Robotic Systems) applies the concept of multi-agent system for robotics. Some of the advantages of the use of MRS over Single-Robot Systems (SRS) are the increased speed of task completion through parallelism, improved solutions for tasks that are inherently distributed in space, time, or functionality, cheaper solutions for complex applications that can be addressed with multiple specialized robots, instead of use of one unique all-capable robot, the increased of robustness and reliability through redundancy *[LEP2008]*. But these advantages do not come for free. For instance, determining how to manage the whole system usually is much more complex than a SRS. The lack of centralized control is one of the reason why the increase of complexity of MRS *[VGO2004]*. Also, MRS requires increased communication to coordinate all the robots in the system. Increasing the number of robots can lead to higher levels of interference between themselves (depends 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 robots are a new approach to the coordination of multi-robot systems which consist of large numbers of relatively simple robots which takes its inspiration from social insects.

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].

Modern robots usually use the same electronic components and devices from computers. Computers use unreliable components, for this reason they improve their reliability using techniques like error control codes, duplication with comparison, triplication with voting, diagnostics to locate failed components, etc. 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 systems for mobile robots have to be able to handle and even learn from the new situation several times. Because of this complex scenario, there are several approaches to implement reliability in robotics. This work will introduce some of these techniques and the next section 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.

Follow there 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 robots is a requirement to enable the whole system works as expected. According to *[RCA1993],* all individual robots 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 large possibilities of faults in robotics, 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 problem. Robot middlewares are one of these tools developed to abstract part of the complexity of these problems.

Several robot middlewares *[BRG2009]*, *[BRG2010]*, *[MAK2007]* try to address the fault detection problem but only single parts of the problem are addressed. Each one of these middleware monitoring tools starts from scratch. And also most of them are driven by the capabilities of the robotics middleware and not by the robotics field needs. Also robot middlewares are usually developed to work as a single and it makes difficult to observe the system as a whole.

3.1 Individual Robots Fault Detection

According to *[MHA2003]* the most popular method of fault detection in robot systems is comparing the sensors values with a pre-determined range of acceptable values (use of thresholds). Other well-known fault detection method is creating a vote system based on different redundant components *[RCA2003]*. If a determined individual component is in faulty state the result will be different of the others. So this individual component could be ignored and the others values are used instead.

Logging is another fault detection technique where data is collected in advance to be analyzed later (off-line fault detection). During the normal runtime, all necessary data is collected and stored in some device. The disadvantages of this technique are that a huge amount of data could be generated. Usually Logging needs another monitor to check if the device is not full and needs clean-up actions *[LOT2011]*. Logging could be used for SRS or for MRS.

3.2 Multiple Robots Fault Detection

Fault detection systems in MRS *[MEN2010]* have the distribution as a coefficient that increases the complexity of the process. The MRS must be able to cooperate and communicate with each other to achieve satisfactory performance and stability. A networked control system is a requirement to connect all agents through communication

networks. Because of this complexity these systems are subject to faults, performance deterioration or even interrupt the operation.

According to *[MEN2010],* several different methods and techniques to deal with these problems can be found in the literature. However, usually these methods are centralized designed, without attending the distributed and decentralized nature. A technique that could be used to monitor MRS is the Distributed Artificial Intelligence (DAI). This methodology is based on the creation of a supervision system agent that is able to communicate direct with other agents in order to perform monitor tasks. Summon et al. *[CHR2009]* states that one of the most important advantages of swarm robotic systems is redundancy. In case one robot breaks down, another robot can take steps to repair the failed robot or take over the failed robot's task. The solution proposed in this paper is creating a completely decentralized algorithm to detect non-operational robots in a swarm robotic system. Each robot flashes by lighting up its on-board light-emitting diodes (LEDs), and neighboring robots are driven to flash in synchrony. Robots that contain error do not flash periodically and can be detected by others. This innovative approach does not use conventional networking communication to perform monitoring tasks what is an advantage compared with other approaches because it does not generate network traffic and it does not depend on the network.

The work [KBL2006] proposes a metric for evaluation the effectiveness of fault-tolerance system. Common metrics are defined and used to measure fault-tolerance of the different systems within the context of system. The goal of this work is to measure by identifying the influence of fault-tolerance towards overall system performance. The work also focuses on capture the effect of intelligence, reasoning, or learning on the effective fault-tolerance of the system. According to this work only few methods are designed to attend 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 that offers self-healing capabilities for MRS. This architecture of the self-healing add-on should be resource efficient to prevent indirect interferences. Scalability is another important requirement. The self-healing add-on should be independent from size and distribution of a MRS. 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 and MRS support.

RoSHA is a self-healing add-on that fulfills the dependability requirements. According to image 7 RoSHA architecture, ROS diagnostics provides some of these requirements

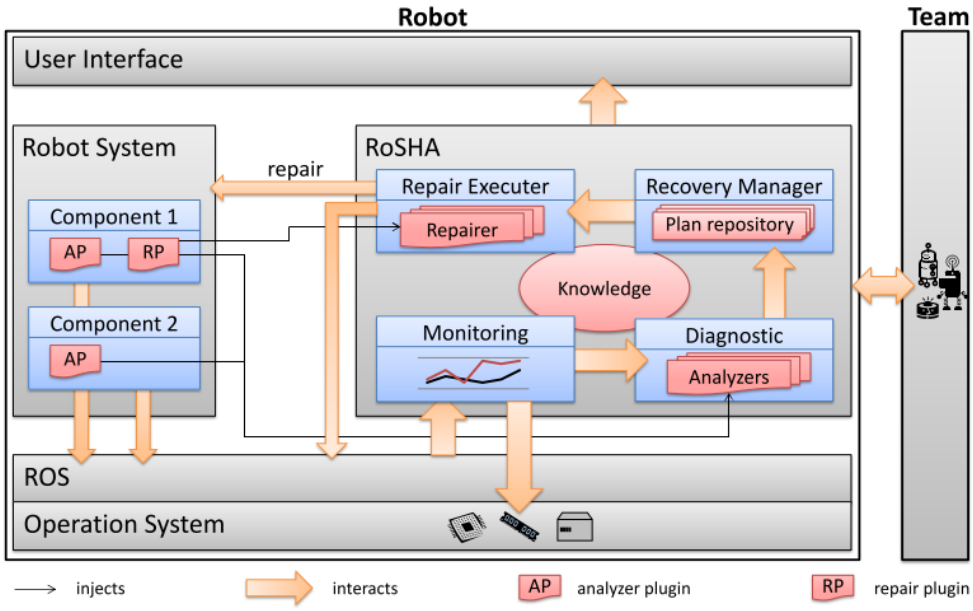

Image 7 – Overview of the RoSHA architecture *[RSH2013]*

Image 7 shows the RoSHA architecture are divided in 4 components. The monitoring component collects information about the current system state. The diagnostic component uses the collected information to identify failure 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 the 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 practicable usage. In order to foster real-world applications and to increase the commercial use. This paper is a very advanced proposal on how to handle the MRS dependability challenges, however this paper 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.

Research Status

This chapter presents the research current status and its goals. It also presents the status of the required activities.

3.3 Research Problem

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.

3.4 Goals

The *goal* of this work is to propose a fault monitoring tool for MRS. Our proposal is to integrate a traditional infrastructure networking monitoring tool with a robotics middleware. Our *hypothesis* is that by combining two consolidated tools we are able to reduce development cost/time by developing an extension for both tools. In return, the proposed MRS fault monitoring tool will have the network scalability, software stability, and software extensibility.

3.5 Research Questions

Considering the main goal and the hypothesis presented previously, this research project intend to address the following research questions:

- Is it possible to adapt an industry standard in IT infrastructure monitoring tool to monitoring and detecting faults in MRS?
- How effective this monitoring system will be?

3.6 Techniques and Tools Analyzed

This section compares the two main types of tools used in this research: IT infrastructure monitoring and robotics middleware. Also describes other technologies used during the development of this work.

3.6.1 IT Infrastructure Monitoring

The goal of a DCIM is to provide to the administrator/users an overview of the entire datacenter status. DCIM tools allow the administrators to store and analyze data related to datacenter servers *[COL2012]*. There are several DCIM tools consolidated in the market with both commercial and free-software licenses. Some of the solutions are open-source and also support the development of extensions or plugins. These plugins are used to enhance the capability of the monitoring tool. The rest of this section introduces well-known IT infrastructure monitoring tools.

Ganglia *[GAN2013]* is a "scalable distributed monitoring system" focused on clusters and grids. It gives the user a quick and easy-to-read overview of your entire clustered system. It is based on a hierarchical design targeted at federations of clusters. It leverages widely used technologies such as XML for data representation, XDR for compact, portable data transport, and data storage and visualization. The algorithms were developed to achieve very low overheads per-node and high concurrency.

Spiceworks *[SPI2013]* is becoming one of the industry standard free network/system monitoring tools. This tool uses SNMP protocol since it has low impact on the network communication with monitoring tasks. Pre-defined alerts can be configured to monitoring 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 a network monitoring tool which offers user-defined views, zooming, and mapping on its Web-based console. This tool uses MySQL to store historical information, its backend is developed in C and the administrator front-end is developed in PHP. The protocols SNMP, TCP and ICMP are supported by the agents that run in the host capturing and sending information to the server.

Nagios is the industry standard in IT infrastructure monitoring according to *[NAG2013]*. This monitoring system was developed focused on scalability and flexibility. Nagios provides information about mission-critical IT infrastructure, allowing detecting and repairing problems and mitigating future issues. Nagios supports the development of extensions or plugin to enhance the original tool capability according with the needs.

The Nagios plugin is a small piece of software that must be developed following the Nagios plugin specification in order to support Nagios API. These plugins can monitor virtually any kind of equipment/devices. Based on these flexible aspects, the proposal is to

create a custom plugin to monitor both software information and also hardware information. Besides the flexibility, Nagios also supports almost all protocols and features supported by the others.

3.6.2 Robotic Middleware

According to *[ELK2012],* robots middleware is a layer between the operating system and software applications, as illustrated in Image 7. It is designed to manage the heterogeneity of the hardware, improve software application quality, simplify software design, reduce development costs, and improve software reusability.

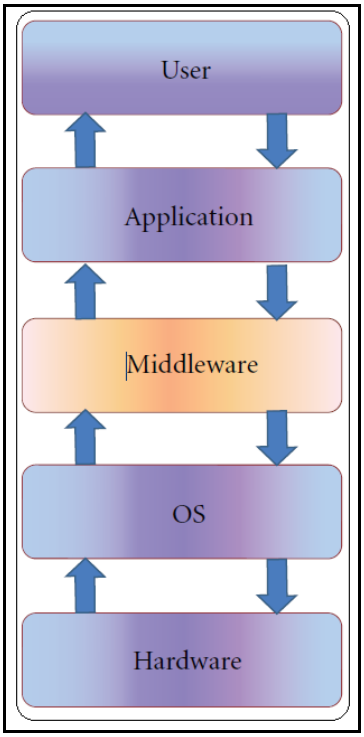

Image 8 - Middleware layer *[ELK2012]*

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. The robot's modules cooperate 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 an object-oriented middleware for robots developed by University of Ulm, Germany. Miro is designed and implemented by applying object oriented design and implementation approaches using the common object request broker architecture (CORBA) standard. According to [MIR2013] the core components have been developed under the aid of ACE (Adaptive Communications Environment), an object

oriented multi-platform framework for OS-independent interprocess, network and real time communication.

Orca *[AMA2006]* is a middleware framework for developing component-based robotics. It is designed to target applications from single vehicles to distributed sensor networks. The main goal of Orca is to enable software reuse in robotics. According to [ORC2013] it provides the means for defining and developing the building-blocks which can be pieced together to form arbitrarily complex robotic systems, from single vehicles to distributed sensor networks.

According to *[SAH2006]* and *[SKJ2006],* UPnP middleware was developed to utilize the Universal Plug and Play (UPnP) architecture for dynamic robot internal and external software integrations and for ubiquitous robot control. UPnP was developed to offer peer-to-peer network connectivity between PCs wireless devices *[UPN2013]. UPnP uses existent standards protocols, such as TCP/IP, HTTP and XML to connect networked devices and manage them.*

Robot Operating System (ROS) is a middleware that provides a communication layer above the host operating system of a heterogeneous computing node. ROS was designed to meet a specific set of challenges encountered when developing large-scale robots systems. According to *[MQU2009]* the ROS main features are:

• Peer to peer (P2P): the purpose of use p2p communication is to avoid unnecessary traffic in the network

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

• Multi-lingual: developed to be language neutral at the message layer;

•Thin: drivers and algorithm development using standalone libraries that have no dependencies on ROS;

• Free and Open-source: The full source code of ROS is publicly available;

• Collaborative development: in order to build large systems the ROS software system is organized into packages.

ROS has a modular design that allows advanced communication functionalities. These advanced communication features could be extended to communicate with any kind of other tools. Moreover ROS middleware provides some tools for fault monitoring *[*LOT2011*]*. These tools are useful for development and monitoring purposes. Also these

tools are developed to monitoring one specific robot each time and not for monitoring the entire MRS. These tools address only a single part of the overall problem of runtime monitoring because they allow to check the status of one component/module at time.

3.6.3 ROS Concepts

ROS *[ROS2014]* has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. The first two, which are relevant for this work, are described next.

About the file system level the most important concept is the ROS packages. Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and release is a package. Metapackages are specialized Packages which only serve to represent a group of related other packages. For this work a simple ROS package is enough and we do not pretend to use metapackages for instance.

Messages and Service types are two important concepts defined in the file system level. Message descriptions are stored into the package folder *MessageType.msg* file and define the data structures for messages sent in ROS. The following snapshot presents an example of a message file that only declares a String attribute:

Message.msg

string input

Service types are service descriptions stored into package folder as a *ServiceType.srv* that define the request and response data structures for services in ROS.

Example of a Service file that only declares a single String attribute for the request and another String attribute for the response:

Monitor.srv

string input

---

string output

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. Nodes are processes that perform computation. For example, one node controls a laser range-finder, one node controls the wheel motors. Nodes

communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs). Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by **publishing** it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will **subscribe** to the appropriate topic. For monitoring and diagnostic purposes ROS suggests the use of /diagnostic topic to publish this kind of data. For request/reply interactions ROS suggests the use of a Service which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.

3.6.4 Fault Reporting using ROS Style - ROS Diagnostics

In ROS the task of analyzing and intuitive reporting the system state is provided by the Diagnostics stack. It consists of development support for collecting, publishing, and visualizing fault information. This tool-chain is built around standardized interfaces, namely the diagnostic topic for monitoring information. Gathered status data are published continuously on the diagnostic topics.

Two instances of the iClebo Kobuki *[KBK2013]* were selected to execute this research. According to *[KBK2013],* iClebo Kobuki is a low-cost mobile research base designed for education and research on state of art robotics. With continuous operation in mind, Kobuki provides power supplies for an external computer as well as additional sensors and actuators. Its highly accurate odometry, amended by our factory calibrated gyroscope, enables precise navigation. The following topics summarize its main features.

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)

Software Specification

- C++ drivers for linux and windows
- ROS node
- Gazebo Simulation

iClebo Kobuki *[KBK2013]* provides C++ drivers for Linux and ROS compatibility that are the requirements of this research. Also this robot already implements the diagnostics information necessary to monitor the robot in the real time and to integrate to the IT Monitoring tool. Kobuki provides status information about the Watchdog, Battery, Cliff Sensor and more. More information about these resources are presented on Section Research progress into the Results of ROS tests chapter.

One example of the Kobuki diagnostic data raw output:

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

The watchdog sensors detects when the Kobuki is connected to the computer via USB, in this example there is no signal of the robot connected to on the computer. Analog input represents the status of the analog buttons present in the robot. Battery shows the robot battery status. The Cliff sensor tries to detected if the robot is in a flat surface or uphill. Digital input is digital buttons that are controlled via software. The gyro sensor gets the current robot orientation. Motor Current monitor if the current of the motor is Ok or

should raise a warning or error. Motor State represents if the motor is enable or disable and wall sensor detects when the robot hits an obstacle. The wheel drop sensors detects if one oh the wheel is not properly in contact with the surface.

3.7 Research Activities Progress

3.7.1 Study theoretical background

This activity was completed. The result of this activity is the Theoretical Background section present on this work.

3.7.2 Study and tests using an IT infrastructure tools:

This activity was completed.

An instance of Nagios was installed and properly configured in order to understand how this tool works and how can we adapt to add new components.

The proposal of this research is to configure each one of the robots of the MRS as a different host in the Nagios database. Installation and configuration steps are described in the appendix at the end of this work.

Results of the Nagios tests

After all Installation and configuration steps completed successfully the Nagios IT infrastructure tool should be up and running on the Linux environment. Nagios provides a Web portal access through Apache Web Server that is accessible at the http://IP_ADDRESS/nagios3/. Any web browser should be able to access this web portal. For this work purposes each robot of a MRS will be added on Nagios as a new host in order to start monitoring the robot as a Host. Image 9 presents the screenshot of Nagios with one host computer (robot).

Image 9 - Nagios Hosts table view

Nagios presents a table containing all added hosts ordered by the Status or any criteria selected by the user. Statuses information will show up for every added hosts table on the Hosts link in the left menu.

Image 10 - Nagios left menu items

On the Hosts option, image 11, the user is able to see all hosts status, and in the top of the screen a *Host Status Totals* and *Service Status Totals* is presented in order to combine all information in one unique place. This example shows only a unique host configured but as more hosts are been added on Nagios the information will be summarized in here.

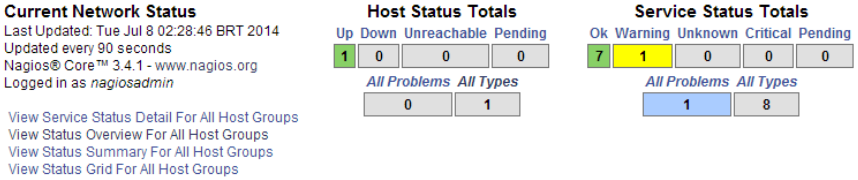

Image 11 - Nagios header - Host and Service Status Totals

On the Image 12 Nagios hosts table view is a central place where the administrator is able to see all configured hosts and their compiled status. The Nagios compile all monitored aspects of the host and summarize it. If all monitored aspects are OK the host status is OK (green line in the image). If one of these statuses is not OK (Critical) a red line will mark the host so the administrator can easily detect and get detailed information about this host. Each line of this table is a different host.

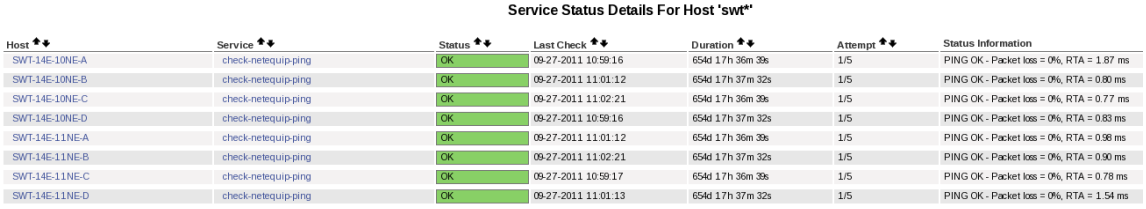

Image 12 - Nagios host table view

In the Services link, Figure 13, the administrator can get all detailed monitored information about a specific host on Nagios. In this research proposal all important information about the host like System load, disk space, Linux services status, total of processes running and also addition robot-related information will be added on Nagios as service.

Service	Status	Last Check	Duration	Attempt	Status Information
Current Load	OK	2014-07-08 02:34:07	40d 5h 31m 11s	1/4	OK - load average: 1.40, 1.51, 1.58
Current Users	OK	2014-07-08 02:34:07	77d 22h 16m 58s	1/4	USERS OK - 5 users currently logged in
Disk Space	OK	2014-07-08 02:34:41	77d 21h 25m 21s	1/4	DISK OK
HTTP	OK	2014-07-08 02:35:04	45d 6h 36m 17s	1/4	HTTP OK: HTTP/1.1 200 OK - 453 bytes in 0,001 second response time
Robot check battery	OK	2014-07-08 02:34:07	1d 8h 59m 35s	1/4	OK - Kobuki Charge Percent 79
Robot sensors	OK	2014-07-08 02:34:07	4d 10h 6m 59s	1/4	SENSORS OK: Status da PUC atualizado
SSH	OK	2014-07-08 02:34:30	40d 11h 31m 52s	1/4	SSH OK - OpenSSH_6.1p1 Debian-4 (protocol 2.0)
Total Processes	WARNING	2014-07-08 02:35:43	1d 8h 49m 52s	4/4	PROCS WARNING: 256 processes

Image 13 - Nagios detailed service table view

3.7.3 Study and tests robotics middleware (ROS)

This activity was completed successfully. An instance of ROS was installed and properly configured. Installation and configuration steps are described in the appendix at the end of this work.

Results of ROS tests

After all installation and configuration steps above completed ROS should be up and running. For start the ROS middleware the ROSCORE service needs to start. Once the *roscore* is running all other topics and services could start as well.

A Kobuki *[KBK2013]* robot was used to validate and test a ROS environment up and running with a real robot. Kobuki provides C++ driver for Linux and ROS compatibility as well as the diagnostic approach implemented that provide to the ROS diagnostic topic all runtime information about the robot sensors. Kobuki provides several different sensors available to monitor and diagnostic the Kobuki robot state. Through the Watchdog sensor the Nagios is able to detect if the robot is connected or not connect to the computer via USB. The Kobuki battery status is also available as well as a Cliff and Gyro sensors. The Motor current sensor is useful to detected robot overload for example and the Wall sensor can detect when the robot knock a barrier. The Whell drop sensors says if all the wheels are in touch with the floor. ROS also provides some tools to monitor and diagnose robot status. For example, RQT Runtime Monitor is a GUI tool distributed with ROS that provides a visual tree of the diagnostic data. This is a real time updated interface, illustrated at Image 14, where the operator can visualize all detailed information about the robot sensors.

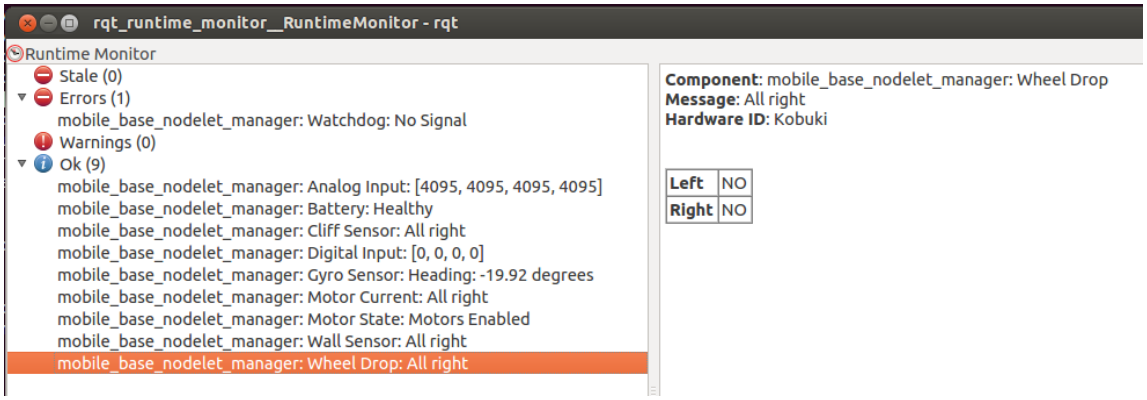

Image 14 - ROS RQT Runtime Monitor connect to Kobuki robot

Image 15 shows all topic information are available from the CLI as well. For instance the CLI *rostopic echo diagnostic* shows exactly the same information as a plain text format.

```
  level: 0
  name: mobile_base_nodelet_manager: Motor Current
  message: All right
  hardware_id: Kobuki
  values:
    -
      key: Left
      value: 0
    -
      key: Right
      value: 0
-
  level: 0
  name: mobile_base_nodelet_manager: Motor State
  message: Motors Enabled
  hardware_id: Kobuki
  values:
    -
      key: State
      value: 1
-
  level: 0
  name: mobile_base_nodelet_manager: Gyro Sensor
  message: Heading: -19.92 degrees
  hardware_id: Kobuki
  values: []
-
  level: 0
  name: mobile_base_nodelet_manager: Digital Input
  message: [0, 0, 0, 0]
  hardware_id: Kobuki
  values: []
-
  level: 0
  name: mobile_base_nodelet_manager: Analog Input
  message: [4095, 4095, 4095, 4095]
  hardware_id: Kobuki
  values: []
--
```

Image 15 - ROS rostopic CLI output

3.7.4 Implement a plug-in to integrate robot middleware with the IT monitoring tool

This activity is completed successfully. Several different approaches were investigated in order to achieve an efficient integration between Nagios and ROS: (I)

publish/subscribe model without diagnostics; (II) Linux service; (III) ROS service; (IV) Subscribe into diagnostic topic.

The ROS topic ‘publish/subscribe model’ did not work as expected because the publish/subscribe paradigm is developed for exchange message in distributed systems without worry about the order. The publisher does not wait until the subscriber receives the message. Nagios request/reply architecture works differently. Once Nagios sends a request the information needs to be available at this moment for the reply.

To overcome this limitation the ‘Linux Service’ approach was studied. This proposal intends to create a C++ Linux Service with network API from scratch. This service would be able to subscribe in a ROS topic and keep all the received data accessible in memory to reply the Nagios request. This approach is feasible, however, the disadvantages are the increase of complexity and difficult the maintainability.

The ‘ROS service’ approach works similar to the previous ‘Linux Service’ approach, but it is much easier and simpler for development and maintainability. This approach was developed and tested during this research. The architecture of this implementation is described in the appendix with details.

The Image 16 illustrate Nagios host table configured with a Robot sensors check (Nagios Reader) getting the information from a ROS Service running. This robot sensors information was generated using ROS publish client. This tool was created to simulate real robot information. In this example the status of the sensors service is OK and an additional string "*Status da PUC atualizado*" was added.

Service	Status	Last Check	Duration	Attempt	Status Information
Current Load	OK	2014-06-29 21:09:49	39d 21h 25m 24s	1/4	OK - load average: 1.33, 1.45, 1.44
Current Users	OK	2014-06-29 21:10:29	77d 14h 11m 11s	1/4	USERS OK - 5 users currently logged in
Disk Space	OK	2014-06-29 21:11:12	77d 13h 19m 34s	1/4	DISK OK
HTTP	OK	2014-06-29 21:11:35	44d 22h 30m 30s	1/4	HTTP OK: HTTP/1.1 200 OK - 453 bytes in 0,002 second response time
Robot check battery	OK	2014-06-29 21:09:31	1d 0h 53m 48s	1/4	OK - Kobuki Charge Percent 79
Robot sensors	OK	2014-06-29 21:10:18	4d 2h 1m 12s	1/4	SENSORS OK: Status da PUC atualizado
SSH	OK	2014-06-29 21:11:01	40d 3h 26m 5s	1/4	SSH OK - OpenSSH_6.1p1 Debian-4 (protocol 2.0)
Total Processes	WARNING	2014-06-29 21:12:14	1d 0h 44m 5s	4/4	PROCS WARNING: 260 processes

Image 16 - Nagios service table getting ROS information

Even though this approach worked, it needs an extra effort to implement for each sensor node a new communication with the ROS Service . This increases the effort to add new robots and sensors. For this reason another approach was evaluated.

The ‘Subscribe into diagnostic topic’ solution implements a client that is able to directly connect to the Diagnostic ROS topic and collect sensor status information as Image

18 shows. This approach requires less implementation effort to add new robots and sensors compared with three previous approaches. For validate this implementation the Battery sensor of a Kobuki robot was added to be monitored by Nagios as a service.

Service	Status	Last Check	Duration	Attempt	Status Information
Current Load	OK	2014-06-29 21:14:49	39d 21h 29m 55s	1/4	OK - load average: 1.60, 1.40, 1.41
Current Users	OK	2014-06-29 21:15:29	77d 14h 15m 42s	1/4	USERS OK - 5 users currently logged in
Disk Space	OK	2014-06-29 21:16:12	77d 13h 24m 5s	1/4	DISK OK
HTTP	OK	2014-06-29 21:16:35	44d 22h 35m 1s	1/4	HTTP OK: HTTP/1.1 200 OK - 453 bytes in 0,002 second response time
Robot check battery	OK	2014-06-29 21:14:31	1d 0h 58m 19s	1/4	OK - Kobuki Charge Percent 79

Image 17 - Nagios service table communicating with Kobuki real robot

This executed test demonstrates the completed solution working with a real Kobuki robot getting the Kobuki's battery charge status. All Kobuki's sensors (and also other robots) could be added using this approach. The Nagios flexibility allows to reuse the same implemented plug-in to monitor a specific sensor in different robots. This approach also supports a MRS running different robots middlewares in the same team at the same time because the adopted solution it is isolated from ROS. For support that, new plug-ins must be developed for others robots middlewares in order to communicate and get sensor information. This approach is more simple and direct than previous ones implemented and tested. The Image 18 illustrates how this architecture works. The Nagios Monitor Host side in the right works exactly the same way it works in the others, the difference here is that Nagios Reader connects to ROS Diagnostic topic instead of connect to a custom ROS Service developed to reply to Nagios.

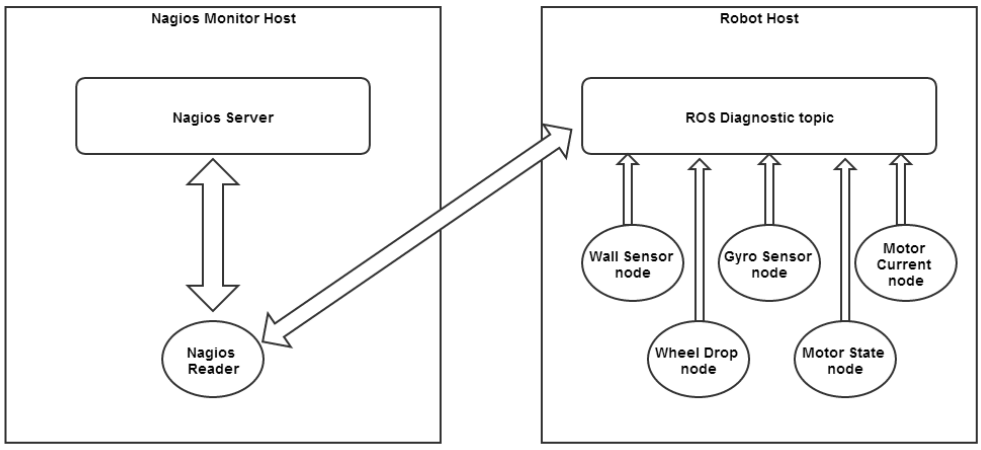

Image 18 - Nagios reader architecture

This architecture connects direct with the ROS diagnostic topic. Some robots manufactures provide drivers that are compatible with ROS diagnostic. There are two

different ways to implement this architecture. The first one is installing the Nagios Reader in the robot computer and executing it remotely from the Nagios server computer. The Nagios Reader connects to ROS diagnostic node through ROS APIs and get the requested information printing the output on a standard output that is read by Nagios engine. This implementation is possible because ROS is developed to work on a network environment. ROS node is a TCP/IP service application listening on a specific port and waiting for subscribers connections. In this case the robots do not need to install any extra software in order to be monitored. The drawback of this approach is because Nagios Reader needs to implement a ROS client, the ROS libraries must be installed on the Nagios machine.

The second option is using Nagios Remote Plug-in Executer (NRPE) instead. This Nagios feature allows the Nagios Server to remotely connect into the target hosts (robot), execute the action of calling Nagios Reader and send the requested information back to the Nagios Server. The advantage of this approach is that Nagios server does not need to install any ROS library. On the other hand, robots need to install Nagios Reader plug-in. Both implementations are feasible and more tests are required to further evaluate these options.

This architecture allows Nagios to be flexible about which sensor should be monitored for each specific robot. It enables heterogeneous MRS containing different kinds of robots or robots with different kinds of sensors. The image 19 illustrates how this configuration works. For each sensor there is a respective Nagios Reader script that to able to connect to ROS diagnostic getting the specific information about this sensor. In this example the Nagios Server trigger the Wall Sensor Reader and this sensor will connect to the ROS Diagnostic topic and return the status of the wall sensor for this specific robot. This operation is repeated for all robots of the MRS.

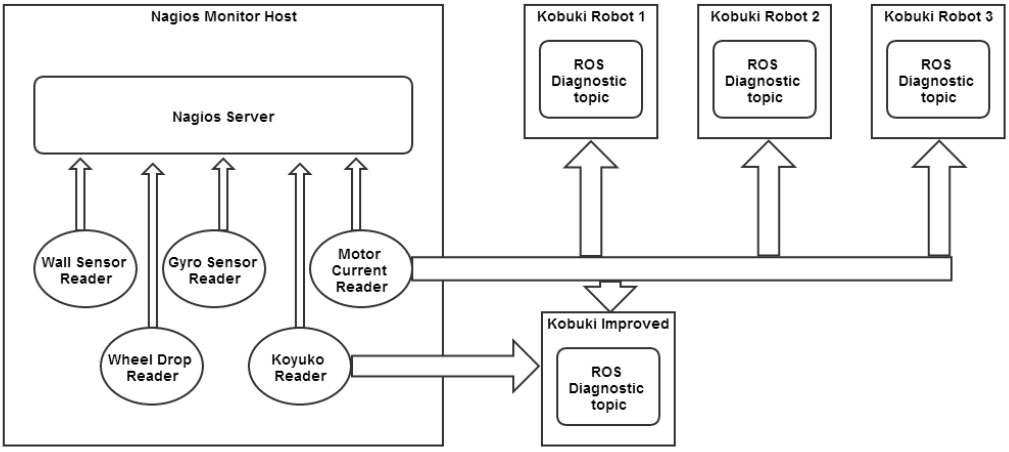

Image 19 - Nagios reader architecture in MRS

In order to configure this environment some parameters should be set in Nagios configuration files. Commands could be defined in Nagios to be used for one or more hosts or in this cases for one or more robots computers hosts. For instance each one this sensors (Wall Sensor, Gyro sensor, motor current sensor...) need to be added on the *commands.cfg* Nagios file using the syntax described in the Source 1 code.

```
1.  define command{
2.          command_name        check-battery
3.          command_line  /home/roman/nagios_ros_kobuki.py -c 15 -w 30
4.  }
5.  define command{
6.          command_name        check-whatchdog
7.          command_line  /home/roman/nagios_ros_kobuki_watchdog.py
8.  }
```

Source 1 - Nagios commands.cfg configuration file

After all commands defined, it is possible to define which commands will be used for each specific host (robot). For instance in the Image 19 the Kobuki Robot 1 needs to add which commands will be checked for this specific robot as source 2 Nagios Kobuki_robot1.cgf configuration file shows.

```
1.  # Define a service to check the robot sensors status
2.  define service{
3.      use                     generic-service     ; Name of service template to use
4.      host_name               localhost
5.      service_description     Robot sensors
6.      check_command           check-robot
7.  }
8.
9.  define service{
10.     use                     generic-service     ; Name of service template to use
11.     host_name               localhost
12.     service_description     Robot check battery
13.     check_command           check-battery
14. }
```

Source 2 - Nagios Kobuki_robot1.cgf configuration file

Another Nagios feature that could be used is Nagios defined groups. These groups combines a set of commands pre-defined for each group. In the example above the robots Kobuki robot 1, Kobuki robot 2 and Kobuki robot 3 have all the same sensors, so these sensors could be defined as a group called Kobuki common sensors for instance. After that all robots of this kind (it means that contains the same sensnors) could just extend the Kobuki common sensors group and do not need to redefine all commands for this robot.

Next steps

- Implement Nagios reader for all available Kobuki's sensors

  Using this approach only check-battery reader was developed, for a complete robot monitoring solution one different reader should be implemented for each sensors.

- Implement Nagios reader for others models of robots on lab

After completed Kobuki's monitoring another robots models could be added to be monitored as well as Kobuki.

- Implement Nagios reader for sensors available on lab

Kobuki and other robots models are flexible and allows another kind of sensors to be added. This different sensors could be monitored as well. See the Kobuki's improved example on Image 19.

- Configure and execute tests on a remote host - Using NRPE

Install Nagios in one robot and configure Nagios to access the Nagios reader plug-ins remotely using NRPE.

- Configure and execute tests on multiple remote hosts (MRS)

Install Nagios in more than one robot and configure Nagios to monitor the entire MRS.

3.7.5 Define the robot’s parameters to be monitored

Study and investigate all available robots and sensors on PUCRS, data, or events that should be collected and monitored. Also identify techniques such as value interval, thresholds or pre-defined acceptable values to monitor and generate warning/alerts on demand. This activity releases a table of values contain the following columns: State or Sensor, recommended interval to check, recommended technique to monitor.

This activity is in progress. At this moment only a proof on concept was created and executed monitoring only one sensor of Kobuki robot. The next steps of this activity is to list and implement the same support for all available sensors.

3.7.6 Planning and executing the experiment

Create a detailed plan on how to mount and configure a controlled environment (containing more than one robot - MRS) and define strategies to simulate fault circumstances. The output of this task is a plan containing the steps to reproduce the tests.

This activity is in progress. At this moment only a proof on concept was executed using only one unique robot. After all robots parameters are defined and the Nagios reader be implemented this experiment should be executed including a MRS.

3.7.7 Tests and analyses of results

Execute the tests as planned before for a continuous period of time, for repeated times, and document all the results. Analyze the results and document the conclusions.

3.7.8 Write the “Seminário de Andamento”

This activity was completed successfully. The result of this activity is this report itself.

3.7.9 Review the state of the art

Keep looking for related papers and approaches. This is a on going activity that will continue until the end of the research. Studied similar works and make some comparisons between the other solutions proposal and this one.

3.7.10 Write papers and Master's dissertation

Write and submit papers for selected conferences. Also write Master’s dissertation according to PUCRS specifications. The deliverables of this task are the submitted papers to conferences and the final dissertation containing the complete work.

3.7.11 Present the work

Make a presentation of all work developed to the board of professors selected to evaluate this work.

Table 1 - Research Activities Progress

Activities	2014 / 2015												
	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan
Study theoretical background													
Study and tests using an IT infrastructure tools													
Study and tests robotics middleware													
Implement a plugin to integrate robot middleware with the IT monitoring tool													
Define the robot's parameters to be monitored													
Planning and executing the experiment													
Tests and analyses the results													
Write the "Seminário de Andamento"													
Review the state of the art													
Write papers and Master's dissertation													
Present the work													

4 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 Transactions on, vol. 1, no. 1, pp. 11–33, 2004.

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

[AMA2006] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for Robotics," In International Conference on Intelligent Robots and Systems (IROS), pp. 163-168, Oct. 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] Brugali, Davide, and Patrizia Scandurra. "Component-based robotic engineering (part i)[tutorial]." Robotics & Automation Magazine, IEEE 16.4 (2009): 84-96. 2009.

[BRG2010] Brugali, Davide, and Azamat Shakhimardanov. "Component-based robotic engineering (part ii)." Robotics & Automation Magazine, IEEE 17.1 (2010): 100-112. 2010.

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

[COL2012] Cole, 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 Transactions on 23.2 (2007): 353-369.

[ELK2012] Elkady, Ayssam, and Tarek 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.

[HUT2002] H. Utz, S. Sablatng, S. Enderle, G. Kraetzschmar, "Miro- Middleware for Mobile Robot Applications," IEEE Transactions on Robotics and Automation, 18(4):493-497, Aug. 2002.

[JCA2003] J. Carlson and R. R. Murphy, “Reliability analysis of mobile robots,” in Robotics and Automation, 2003. Proceedings. ICRA’03. IEEE International Conference on, vol.1. IEEE, pp. 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.

[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.

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

[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, page To appear. Pan Stanford Publishing, 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." Control and Fault-Tolerant Systems (SysTol), 2010 Conference on. IEEE, 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. (IROS2003). 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.

[MLV1994] M. L. Visinsky, J. R. Cavallaro, and I. D. Walker, “Robotic fault detection and fault tolerance: Asurvey”, Reliability Engineering & System Safety, vol.46, no.2, pp.139–158, 1994.

[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.

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

[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.

[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), Oct. 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, no. December, 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.

5 Appendix

5.1 Nagios installation steps

==========

Steps to install Nagios

==========

Install packages

- sudo apt-get install -y nagios3

Set admin password

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

==========

Nagios Remote Plugin Executor

==========

The NRPE (Nagios Remote Plugin Executor) plugin allows you to monitor any remote Linux/Unix services or network devices. This NRPE add-on allows Nagios to monitor any local resources like CPU load, Swap, Memory usage, Online users, etc. on remote Linux machines. After all, these local resources are not mostly exposed to external machines, an NRPE agent must be installed and configured on the remote machines.

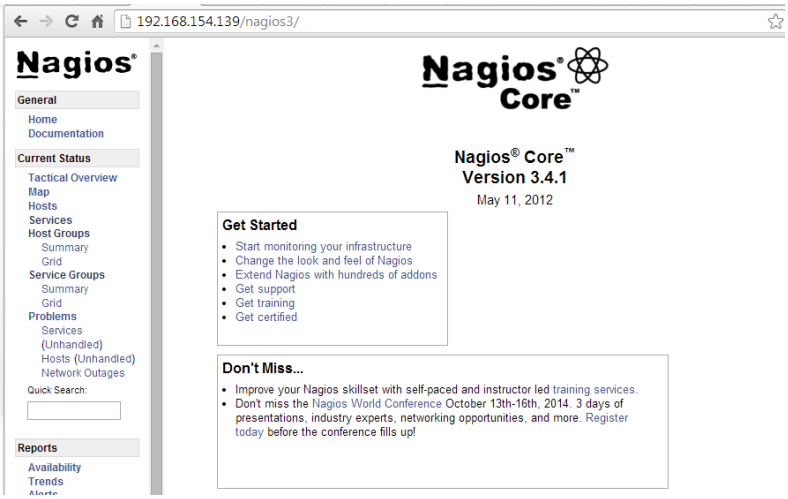

Image 20 - Nagios Web portal

5.2 Nagios configuration steps

fix Disk Critical defect

- This is bug #615848. You can either give the nagios user permission to that file or just ignore the file during check. To ignore the file, edit the disk.cfg file located in /etc/nagios-plugins/config and add the arguments [-A -i '.gvfs'] at the end of the command line arguments for the command check_disk and check_all_disks.

5.3 ROS installation steps

==========

Steps to install and configure ROS

==========

Install ROS

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

Configure environment variables

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

Create a ROS Workspace

mkdir -p ~/catkin_ws/src

cd ~/catkin_ws/src

catkin_init_workspace

cd ~/catkin_ws/

catkin_make

source devel/setup.bash

Filesystem Concepts

Packages: Packages are the software organization unit of ROS code. Each package can contain libraries, executables, scripts, or other artifacts.

Manifest (package.xml): A manifest is a description of a package. Its serves to define dependencies between packages and to capture meta information about the package like version, maintainer, license, etc...

5.4 ROS configuration steps

##

Diagnostic

##

Instalation

##

apt-get install ros-hydro-turtlebot*

apt-get install ros-hydro-kobuki*

run only at first time

rosrun kobuki_ftdi create_udev_rules

##

##

Execute

##

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

5.5 ROS Service approach source codes

ROS Service approach is a ROS package developed to receive and store information from other topics into a Service that is available for Nagios requests.

ROS package structure is

- ROS message file: Define the parameters to be exchanged between the service and clients. Two strings called input and output were created.
- ROS Service: Implement the ROS Service interface and all business rules necessary to send back replies in the Nagios format.
- ROS Publisher: Simple ROS publisher client created to simulate a real robot information.
- ROS Nagios reader: Simple client created to allow Nagios to request information from the ROS Service.

Image 23 illustrates this architecture. At each pre-determined interval of time the Nagios Server execute the ROS Nagios Reader. The TOS Nagios Reader is a Nagios plug-in that try to connect if the ROS Service passing a determined request information that the Service could identify it is a Nagios request and send back the robot sensor statues in Nagios format response.

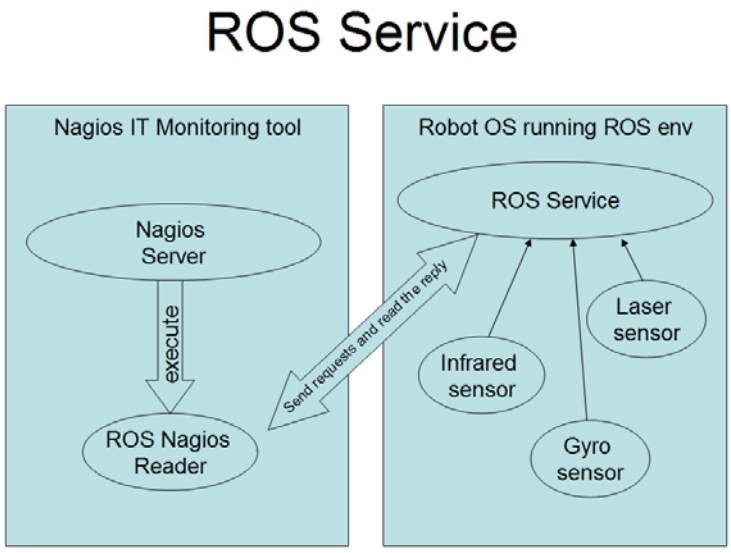

Image 21 - ROS Service diagram

Follow the ROS Service source code and ROS Nagios Reader client example. The ROS publisher is an created example on how to create a client that is able to publish diagnostic information into the ROS Service (in this example could be used to publish Gyro sensor information or any other).

5.5.1 ROS Service.cpp

```
#include "ros/ros.h"
#include "monitor/Monitor.h"
// basic file operations
#include <iostream>
#include <fstream>

// Persist the value on memory
std::string persist;
std::string serviceStatus;
std::string informationText;


// Get the sensor values from memory and format a Nagios output
std::string getNagiosOutput(){

  std::string empty ("");

  /* serviceStatus
    0 - OK - The plugin was able to check the service and it appeared to be functioning properly
    1 - Warning - The plugin was able to check the service, but it appeared to be above some "warning" threshold or did not
appear to be working properly
    2 - Critical - The plugin detected that either the service was not running or it was above some "critical" threshold
    3 - Unknown - Invalid command line arguments were supplied to the plugin or low-level failures internal to the plugin (such
as unable to fork, or open a tcp socket) that prevent it from performing the specified operation. Higher-level errors (such as
name resolution errors, socket timeouts, etc) are outside of the control of plugins and should generally NOT be reported as
UNKNOWN states.
  */
  // If serviceStatus is not set, set to 0
  //if (empty.compare(serviceStatus) == 0) {
  //    serviceStatus.assign("0");
  //    informationText.assign("No information received yet. Wait few minutes.");
  //} else {
    // TODO: Replace by a valid service Status logic

    // key -> value (IR=SENSOR OK)

    serviceStatus.assign("SENSORS OK");
    informationText.assign(persist);
  //}

  // Nagios output should be in the format:
  //  SERVICE STATUS: Information text
  return serviceStatus + ": " + informationText;
```

```
}

bool add(monitor::Monitor::Request  &req,
         monitor::Monitor::Response &res)
{

  // Check if it is a client input or a Nagios request
  std::string str1 ("nagios_request");

  if (str1.compare(req.input) == 0) {
    // Nagios request

    // get memory value and set in the response.output
    res.output = getNagiosOutput();

    // Convert string to char to be used by ROS log
    const char * c = getNagiosOutput().c_str();
    ROS_INFO("Sending response: [%s]", c);


  } else {
    // Sensor input
    // receive the value and handle it properly

    // Store the value
    res.output = req.input;
    persist = req.input;

    // Convert string to char to be used by ROS log
    const char * c = persist.c_str();
    ROS_INFO("Received and persisted value: [%s]", c);

    // Write output in a buffer

  }

  //res.output = req.input;
  return true;
}

int main(int argc, char **argv)
{

  ros::init(argc, argv, "server");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("server", add);
  ROS_INFO("Monitor server started.");

  ros::spin();

  return 0;
}
```

5.5.2 ROS Publisher.cpp

```
#include "ros/ros.h"
#include "monitor/Monitor.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "infrared_fault_detector");
  if (argc != 2)
  {
    ROS_INFO("usage: client String");
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<monitor::Monitor>("server");
  monitor::Monitor srv;
  srv.request.input = argv[1];

  if (client.call(srv))
  {
    std::cout << srv.response.output;
    //ROS_INFO("Input: %s", srv.response.output);
  }
  else
  {
    ROS_ERROR("Failed to call service");
```

```
    return 1;
  }

  return 0;
}
```

5.5.3 Nagios Reader.cpp

```
#include "ros/ros.h"
#include "monitor/Monitor.h"
#include <cstdlib>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "nagios_status_requester");

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<monitor::Monitor>("server");
  monitor::Monitor srv;
  srv.request.input = "nagios_request";

  if (client.call(srv))
  {
    //ROS_INFO("Input: %s", srv.response.output);
    std::cout << srv.response.output << std::endl;
    return 0;
  }
  else
  {
    //ROS_ERROR("Failed to call service");
    std::cout << "SENSOR ERROR: The client is not able to connect to ROS Service." << std::endl;
    return 2;
  }

  std::cout << "Unknown error trying to connect to ROS Service." << std::endl;

  return 3;
}
```

5.6 Nagios reader plug-in source codes

5.6.1 Nagios battery reader.py

```
#!/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

# TEMPLATE FOR READING PARAMETERS FROM COMMANDLINE
parser = OptionParser()
parser.add_option("-H", "--host", dest="host", default='localhost', help="A message to print after OK - ")
parser.add_option("-w", "--warning", dest="warning", default='40', help="A message to print after OK - ")
parser.add_option("-c", "--critical", dest="critical", default='20', help="A message to print after OK - ")
(options, args) = parser.parse_args()
```

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

kobuki_charge = None
kobuki_percentage = None

def callback_kobuki(data):
    global kobuki_charge
    global kobuki_percentage

    ready = False

    while not ready:
        for current in data.status:
            if current.name == "mobile_base_nodelet_manager: Battery":
                    for value in current.values:
                        if value.key == "Charge (Ah)":
                                kobuki_charge = value.value
                        if value.key == "Percent":
                                kobuki_percentage = value.value
                         ready = True

    time = rospy.get_time()
    kobuki_percentage = int(float(kobuki_percentage))
    rospy.signal_shutdown(0)

def listener():
    rospy.init_node('check_battery_kobuki', anonymous=True,  disable_signals=True)
    rospy.Subscriber("diagnostics", DiagnosticArray , callback_kobuki)
    rospy.spin()

def myhook():
    if kobuki_percentage < int(options.critical):
        print "CRITICAL - Kobuki Charge Percent %s | kobuki_battery=%s" % (kobuki_percentage,kobuki_percentage)
        exiting(CRITICAL)
    elif kobuki_percentage < int(options.warning):
        print "WARNING - Kobuki Charge Percent %s | kobuki_battery=%s" % (kobuki_percentage,kobuki_percentage)
        exiting(WARNING)
    else:
        print "OK - Kobuki Charge Percent %s | kobuki_battery=%s" % (kobuki_percentage,kobuki_percentage)
        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) < 5:
            print "usage %s -c <critical> -w <warning>" % (sys.argv[0])
                exiting(UNKNOWN)
        rospy.on_shutdown(myhook)
        listener()
    except rospy.ROSInterruptException:
        exit
```