

Trabajo Fin de Máster

Máster en Ingeniería Electrónica, Robótica y  
Automática

Desarrollo de una arquitectura para la implementación  
y simulación de misiones multi-UAV en la  
competición de robótica aérea MBZIRC

Autor: Francisco Javier Domínguez Jiménez

Tutor: Jesús Capitán Fernández

**Dep. de Ingeniería de Sistemas y Automática**  
**Escuela Técnica Superior de Ingeniería**  
**Universidad de Sevilla**

Sevilla, 2020





Trabajo Fin de Master  
Máster en Ingeniería Electrónica, Robótica y Automática

# **Desarrollo de una arquitectura para la implementación y simulación de misiones multi- UAV en la competición de robótica aérea MBZIRC**

Autor:

Francisco Javier Domínguez Jiménez

Tutor:

Jesús Capitán Fernández

Profesor Contratado Doctor

Dep. de Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2020



Trabajo Fin de Master: Desarrollo de una arquitectura para la implementación y simulación de misiones multi-UAV en la competición de robótica aérea MBZIRC

Autor: Francisco Javier Domínguez Jiménez

Tutor: Jesús Capitán Fernández

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal



*To my family*

*To my teachers*





# Acknowledgements

---

*First, I want to thank my family for their support over the years, especially for the encouragement they have given me in the good times, but especially in the difficult times.*

*Especially to my sister Rocio, whose unconditional support and our constant conversations have helped me to move forward on hundreds of occasions.*

*I would also like to thank Marta, who, thanks to her constant words of encouragement, her advice, and her brilliant attitude, has given me the strength and constancy to carry on every day.*

*To my two closest friends, Alejandro and Fernando, who, together with them, started the path of engineering and have always accompanied me.*

*To all my project partners at MBZIRC, who without them, this project would not have been possible to complete and document, but especially to Alejandro Sánchez, who, from the beginning, was by my side, helping me and supporting me in everything I needed.*

*Finally, I would like to thank my tutor, Jesús, for giving me the extraordinary opportunity to participate in such an important worldwide robotics competition as MBZIRC, and for his willingness to help and support me from the first moment we met.*

*To all of you, thank you.*

*Francisco Javier Domínguez Jiménez*

*Sevilla, 2020*



# Resumen

---

En este documento, se detalla la solución aplicada a la competición de robótica internacional MBZIRC que tuvo lugar en Abu Dhabi en Febrero del año 2020. Aquí, el lector podrá encontrar la arquitectura genérica realizada para resolver el reto propuesto por los organizadores, incluyendo detalles sobre el hardware empleado y realizando una extensa revisión del software y de su estructura interna, así como las herramientas utilizadas para llevarlas a cabo.

También se mostrarán la condiciones de la competición, así como una descripción del Tercer Reto de la competición MBZIRC, junto con imágenes reales tomadas durante la misma.



# Abstract

---

This document details the solution applied to the MBZIRC international robotics competition that took place in Abu Dhabi in February 2020. Here, the reader will find the generic architecture made to solve the Challenge 3 proposed by the organizers, including details about the hardware used and making an extensive revision of the software and its internal structure, as well as the tools used to carry them out.

The conditions of the competition will also be shown, as well as a description of the MBZIRC Challenge 3, together with real images taken during the competition.



# Index

---

<b>Acknowledgements</b>	<b>ix</b>
<b>Resumen</b>	<b>xi</b>
<b>Abstract</b>	<b>xiii</b>
<b>Index</b>	<b>xv</b>
<b>Figure Index</b>	<b>xviii</b>
<b>1 Introduction and Objectives</b>	<b>1</b>
<b>2 Software Tools</b>	<b>4</b>
2.1 <i>ROS Environment</i>	4
2.2 <i>Why ROS</i>	4
2.3 <i>Elements of ROS</i>	5
2.3.1 <i>ROS Filesystem-Level</i>	5
2.3.2 <i>ROS Computation Graph Level</i>	5
2.3.3 <i>ROS Community Level</i>	6
2.4 <i>Key Elements</i>	6
2.4.1 <i>Nodes</i>	6
2.4.2 <i>Topics</i>	8
2.4.3 <i>Messages</i>	9
2.4.4 <i>Services</i>	10
2.5 <i>Gazebo and Rviz</i>	10
2.6 <i>SMACH</i>	12
2.6.1 <i>SMACH Concepts</i>	12
2.6.2 <i>SMACH States</i>	13
2.6.2 <i>SMACH Containers</i>	13
<b>3 General Software Architecture for Multi-Robot Cooperative Missions</b>	<b>15</b>
3.1 <i>Architecture Components</i>	15
3.1.1 <i>What is an Agent?</i>	15
3.1.2 <i>What is a Task?</i>	15
3.1.3 <i>Central Unit Agent</i>	16
3.2 <i>UAV Agent</i>	17
3.3 <i>ROS Components</i>	19
<b>4 Implementation of a Firefighting Mission With a Multi-Robot Team</b>	<b>21</b>
4.1 <i>Challenge 3 Description</i>	21
4.2 <i>Mapping with Sensors and Hector SLAM</i>	23
4.3 <i>Fire Detection in Walls and on the Arena</i>	26
4.4 <i>Shape Detection</i>	27
4.4.1 <i>Window Gap Detection</i>	27
4.4.1 <i>Fire Pipes Detection: Circle Detection</i>	28
4.5 <i>Fire Extinguishing</i>	29
<b>5 Experimental Results</b>	<b>31</b>

<i>5.1 Challenge 3 Simulation</i>	31
<i>5.2 Hector SLAM Implementation</i>	32
<i>5.3 Thermal Camera Implementation</i>	34
5.3.1 Thermal Detection Node Experiments	40
<i>5.4 Circle Detection Implementation</i>	42
5.4.1 Circle Detection Experimental Results	45
<i>5.5 Fire detector and Circle Detector Working Together</i>	46
<b>6 Final Conclusions</b>	<b>49</b>
<b>Annex: Fire Detector Code</b>	<b>51</b>
<b>Annex: Hector SLAM</b>	<b>58</b>





# FIGURE INDEX

---

Figure 1: Main logo of MBZIRC's International Robotics Challenge	2
Figure 2: ROS logo	4
Figure 3: Terminal screen showing rosnode command	7
Figure 4: Terminal screen showing rostopic list command	8
Figure 5: Terminal screen with ROS messages	9
Figure 6: Terminal Screen showing the content of a specific message	9
Figure 7: Screen Terminal showing a list of services	10
Figure 8: Gazebo screen simulator	11
Figure 9: Rviz window	11
Figure 10: State Machine Concept	12
Figure 11: Central Unit Diagram	16
Figure 12: Central Unit services	16
Figure 13: UAV Agent diagram in Challenge 3	18
Figure 14: UGV task diagram for Challenge 2 and Challenge 3	18
Figure 17: UAL diagram	20
Figure 18: Simulated building and real building	21
Figure 19: Fire plate with a red silk, simulated and real	22
Figure 20: Fire spot placed in the wall	22
Figure 21: Diagram of Hector SLAM transformations	24
Figure 22: Transformation Tree for Hector SLAM	25
Figure 23: Gazebo and Rviz simulation for Hector SLAM	25
Figure 24: TeraRanger Evo Thermal 33	26
Figure 25: Thermal Image screen	27
Figure 26: Intel RealSense D435i camera	28
Figure 27: Sensor Camera inside simulation in Gazebo	28
Figure 28: Competition Fire Spot	29
Figure 29: UAV Agent shooting water to the fire spot detected	29
Figure 30: Simulation Building	31
Figure 31: Fragment of code - Arguments to Hector SLAM	32
Figure 32: Fragment of code - Frames and Transformation	32
Figure 33: Hector SLAM simulation in Gazebo	33
Figure 34: Hector SLAM inside the building	33
Figure 33: Thermal Node Flowchart	34

Figure 35: Gazebo Simulation of the laser	35
Figure 36: Fragment of code – Launcher of Fire Detector Node	35
Figure 37: Fragment of code - Subscribers and publisher to the Thermal Node	36
Figure 38: Fragment of code - Function to convert array to matrix	37
Figure 39: UAL position function	37
Figure 40: Fragment of code - Laser measurement to calculate the estimated distance to the wall	38
Figure 41: Fragment of code - Declaration of the black and white thermal image	38
Figure 42: Fragment of code - Calculation of the center of the fire contours	39
Figure 43: Fragment of code - Object Detection message	39
Figure 44: Fragment of code - How to publish the ObjectDetection List	40
Figure 45: Fire spot detected	41
Figure 46: Fire Extinguished	41
Figure 47: Thermal Image Detection	42
Figure 48: Terminal display of the Fire Detector	42
Figure 49: Fragment of code - Reading the parameters from Circle Detector class	43
Figure 50: Fragment of code - Callback camera information	43
Figure 51: Fragment of code - Callback Sync Color Depth	44
Figure 52: Fragment of code – Functions	44
Figure 53: Fragment of code - Masking Grayscale Image	44
Figure 54: Fragment of code - HoughCircle Detection and later comprobations	45
Figure 55: Circle Detection in real	45
Figure 56: Circle Detection on Fire Ring	46
Figure 57: Both algorithms working together	46
Figure 58: Fire detection when the fire ring is turned down	47
Figure 57: MBZIRC's Challenge 3 Score	49



# 1 INTRODUCTION AND OBJECTIVES

---

**W**ith the purpose of obtaining a nearer approach to the aerial robotics world and given the need of adapting to an environment in constant changes, by using a combination of different Unmanned Aerial Vehicle (UAVs) and Unmanned Ground Vehicles (UGVs) integrated into a multi-robot architecture, a group of challenges will be set and resolved to apply, as much as possible, a generic architecture which should be robust when a communication error is encountered.

These challenges are proposed by the international MBZIRC competition which is composed by 3 different tasks, all of them using a multi-robot architecture

## **But, first of all, what is MBZIRC?**

The Mohamed Bin Zayed International Robotics Challenge (MBZIRC) is a biennial international robotics competition that provides a technologically demanding set of challenges. MBZIRC aims to inspire future robotics through innovative solutions and technological excellence that will allow us to perform some different tasks that were impossible before the appearance of robots.

Robotics is poised to have a transformative impact in a variety of new markets and on various human social aspects. These include robot applications in disaster response, healthcare, domestic tasks, transport, space, manufacturing, and construction. However, there is a gap between current reality in robotic capabilities and the requirements of potential applications. Enabling technologies for such applications include robots working more autonomously in dynamic, unstructured environments, while collaborating and interacting with other robots.

MBZIRC aims to focus on some of these applications and enabling technologies by providing a demanding set of robotics challenges to attract the best international teams to showcase and benchmark their solutions. Similar to other major competitions, MBZIRC aims to provide an environment that fosters innovation and technical excellence, while encouraging spectacular performance with robotics technology.

## **What are MBZIRC challenges consist of?**

The first Challenge consists of a team of autonomous UAVs whose purpose is to track and interact with a set of objects following 3D trajectories. In particular, the main objective of this Challenge is to use a UAV to capture another intruder UAV inside the arena.

The second Challenge consists of a team of UAVs and a UGV, that will collaborate to autonomously locate, pick, transport and assemble different types of brick-shaped objects to build a predefined structures, in an outdoor environment. In particular, this Challenge is motivated by construction, automation, and autonomous robot-based 3D building of large structures, such as a wall or a building.

Lastly, the Challenge 3 consists of a team of UAVs and a UGV. They will collaborate to autonomously extinguish a series of simulated fires in an urban high rise building fire fighting scenario. Challenge 3 is motivated by the use of robots in an urban firefighting. It requires the team of robots to collaborate so they can autonomously carry out a series of firefighting tasks in an outdoor-indoor environment.

## **But why MBZIRC?**

Of course, such a challenging competition will not only provide an enormous contribution to the robotics world, but also will force us to develop and create a generic architecture that can be used in future applications, allowing us to reutilize critical parts of the system.

From all the system created and modulated to MBZIRC competition, a generic architecture is one of the most crucial assets obtained. That is because it can be correctly implemented in different robotics applications that can even improve the operative part of the system.

## Which is the objective of this particular project?

In this project it is described how a particular solution for MBZIRC robotics competitions Challenge 3 is provided, using a generic architecture that is also applied to the second Challenge.

More specifically, the main contribution in this project is a general description of the hardware and tools used to create the system and an extensive, detailed explanation of the software developed. In here, a complete review of the Challenge 3 will be made, including the exact sensors, simulations, mock-ups, and demonstrations that were carried out during the project and the real competition.

This document will focus explicitly on showing how to define the multi-robot architecture, explaining the tools used for it, such as ROS, Rviz or Gazebo, and giving a particular solution for the Challenge 3 of the international MBZIRC competition, including simulations and results obtained, in a software perspective.

This document is divided into several different chapters, so the reader can look for the information needed conveniently.

For example, Chapter 2 will explain which software tools were used to develop the system and architecture, including an explanation of what ROS is for all those unfamiliar with this Operating System. Along with it, a how-to-use guide is provided, including several elements such as Node, Topics, or Messages. Also, a review of Gazebo and RVIZ, the simulation environments, are provided as well. And, of course, the Smach library, which stands for State Machine, will be explained.

In Chapter 3, a review of the General Architecture of the Challenges will be explained. Here is a detailed of how every component interacts with each other to create a complex system in which every agent performs a task.

In Chapter 4, an in-depth definition of the Challenge 3 will be provided. This one includes a more specific and technical description of the Challenge, along with the main pieces of software developed to solve the problem. In here, it will be explained how the mapping is done indoors, how to detect fire and windows, and how the fire will be extinguished.

In Chapter 5, the results will be discussed, showing how the simulations work on their own, in the first place and then, how they were integrated together.

To conclude, the last chapter will provide a general summary of everything accomplished and done, providing the results obtained in the actual competition.



Figure 1: Main logo of MBZIRC's International Robotics Challenge



# 2 SOFTWARE TOOLS

---

In order to create a robust architecture based on a collaborative group of UAVs, it is imperative to use the adequate tools for programming, simulation, and testing due to the complex task of coordinating several robots at the same time.

This chapter will explain which software tools are going to be used to develop the system and architecture. Here, a general ROS explanation will be provided, introducing every essential element available in the Operative System, such as Nodes or Topics, so every reader can understand how ROS really works.

Along with ROS, using Gazebo and Rviz, which are the main simulation tools used to develop the system, it will be possible to create a reliable platform that will allow us to create a generic and multipurpose structure for any challenge.

Lastly, an explanation of how SMACH library works will be provided, due to the critical paper it plays in the entire architecture.

## 2.1 ROS Environment

ROS is an open-source system oriented to programming robots. It provides all the services that could be expected from an operating system, plus it includes hardware abstraction, implementation of the commonly-used functionalities, low-level device control, etc. Also, using ROS, it is possible to pass messages between processes and manage packages.



Figure 2: ROS logo

ROS can also provide a few libraries and tools for building, writing, and running any code across multiple computers. Even more, by using ROS, it is possible to exchange messages between different computers connected to the same network, allowing, therefore, communication in UAVs, for example.

To see the connections between different modules in ROS, this system also provides a graph function that allows us to know every type of relationship within nodes. Although it is not a real-time framework, it is possible to integrate it with realtime code

## 2.2 Why ROS

Provided that there are many other types of frameworks, ROS supports code reuse in any robotic research and development. Also, ROS is a distributed framework of processes that allows any executable file to be



individually designed and be coupled at the runtime. Those processes are grouped into packages, which are relatively easy to share and distribute.

ROS also allows language independence. Thus it is possible to use Python, C++, Lisp, Java, or even Lua, which increment the variety of possibilities to program the desired robot. Also, it has integrated a framework called Rostest, which provides a way to debug the code created and spot errors in the system. Additionally, ROS only runs on a Unix-based system.

## 2.3 Elements of ROS

ROS is set with three different levels of concepts that will be discussed in the next pages. By using these three levels, it is possible to create an entire system to run the desired robot.

### 2.3.1 ROS Filesystem-Level

These are the filesystem level concepts that mainly will be covering all ROS resources found on disk. Between them there are:

1. Packages: they are the central unit for organizing all the software in the ROS system. It may contain processes, datasets, any configuration file, and ROS dependent-libraries. These are the main kind of build and item in the entire ROS platform.
2. Metapackages: they are specialized types of packages that are only useful to represent a group of related other packages. These are used as a backward-compatible placeholder for a converted Rosbuild stack.
3. Package manifest: they provide metadata about a package, including the name, the version, the description, among others.
4. Repositories: they are a collection of packages that share a Version Control System or VCS system. This means that this type of packages shares the same version, and they can be released using a catkin release tool.
5. Message types: these are the message descriptions that define the data structure for any message sent in the ROS system.
6. Service types: these are service descriptions that define the request and response data structures for services in ROS services.

### 2.3.2 ROS Computation Graph Level

This level is the peer-to-peer network of ROS system processes.

1. Nodes: they are processes whose task is to perform any kind of computation. Due to the modular nature of ROS, a robot may be executing more than one node.
2. Master: by using it, other nodes are capable of finding other nodes, exchange messages, and invoke services. ROS Master is the one that provides the name registration and lookup to the rest of the elements the computational graph level.
3. Parameter server: it is part of the Master, and it allows data to be stored in a central location.
4. Messages: they are data structures whose data fields are defined with standard types such as integers, floating-point, unsigned, double, character chains, structures, etc. Every node possesses the ability to communicate with other nodes by passing messages.
5. Topics: all the messages are routed through a transport system with a subscribe and publish semantic. One node will be sending a message by publishing it to a specific topic, while another node will be obtaining those messages by subscribing to the same topic that the other node published on. It is possible to have a wide variety of nodes subscribing to the same topic, while a single node may publish or subscribe to multiple topics. In general, those nodes are not aware of the existence of the other ones.
6. Services: it is a way to create a distributed system using a request and reply model. A service is defined by a pair of message structures, one for requesting and another one for replying. A node will offer a service under a particular name, and a client node will be using that service by sending the request message and awaiting a reply.

7. Bags: These are a format that allows us to save and playback ROS message data.

In particular, ROS Master will be acting as a name service in the ROS Computation Graph, and that is because it will be storing any topic and service registration information for the other ROS nodes.

Nodes communicate with the Master to report their registration information, and they can receive any information about other registered nodes in the Master. Also, the Master makes any callback necessary to the nodes whenever the registration information has changed, allowing nodes to dynamically create connections while new nodes are run in the network.

Through the Master, nodes will be able to connect to each other directly. Nodes that subscribe to a topic will request connections from nodes that publish to that specific topic, and they will be connecting using a particular protocol, called TCP-ROS, which comes from TCP/IP sockets communication.

By using this architecture, it is possible to decouple operations. Names are the primary means by which one can create more extensive and complex systems. Every ROS client library can support a command-line remapping of names. This means that a compiled program can be reconfigured at runtime so it can operate in a different Computation Graph topology.

For example, in the case of a laser control, it is possible to start a node driver that communicates to the laser and publishes messages to a topic. So, to receive those data, another node will be subscribing to that particular topic. In case it is needed, and other robots require the use of the laser, the only thing that has to be done is changing the nodes so that the new one can subscribe to the topic.

### **2.3.3 ROS Community Level**

This level is related only to share information, software, resources, and knowledge between the users of ROS.

1 - Distributions: these are collections of versioned stacks one can install in the computer. It provides an easier way to install a collection of different software.

2 - Repositories: by using repositories, one can find almost any code needed to create a software platform for a robot.

There are many other resources included, such as a ROS wiki, in which it is possible to find tutorials, guides, and all the basics required to understand how ROS works.

Due to the high importance of ROS in this project, in the next sections, a more in-depth explanation will be described on how nodes, topics, services and messages perform and relation between one and another, for the sake of clarity.

## **2.4 Key Elements**

To understand how ROS works, an intensive review of how nodes, topics, services, and messages interact with each other will be explained. A few examples will be provided, and a short explanation of how to code will be given along the description.

### **2.4.1 Nodes**

A node is a process that can perform any kind of computation. Nodes can communicate with each other by using topics, services, or the Master's parameter server. As was explained previously, every node is created in a way that it can be used separated from other nodes.

For example, in a UAV with a mounted Rpliddar laser, a node will have the task to obtain the measures from

the laser. Another node will use those measures to create a map and locate the robot. Other nodes could transmit those measures to a UGV platform so that it can track the path the UAV follows and so on.

All nodes in the example can perform their tasks on their own. In this way, an additional fault tolerance is obtained because crashes are potentially isolated to an individual node, making it easy to debug the entire system. Also, by using nodes, it is possible to keep the system much simpler and easier to read compared to other systems.

To write a node, it can be done by using a ROS client library. If it is wished to use C++, Roscpp will be applied. For a python approach, rospy is used.

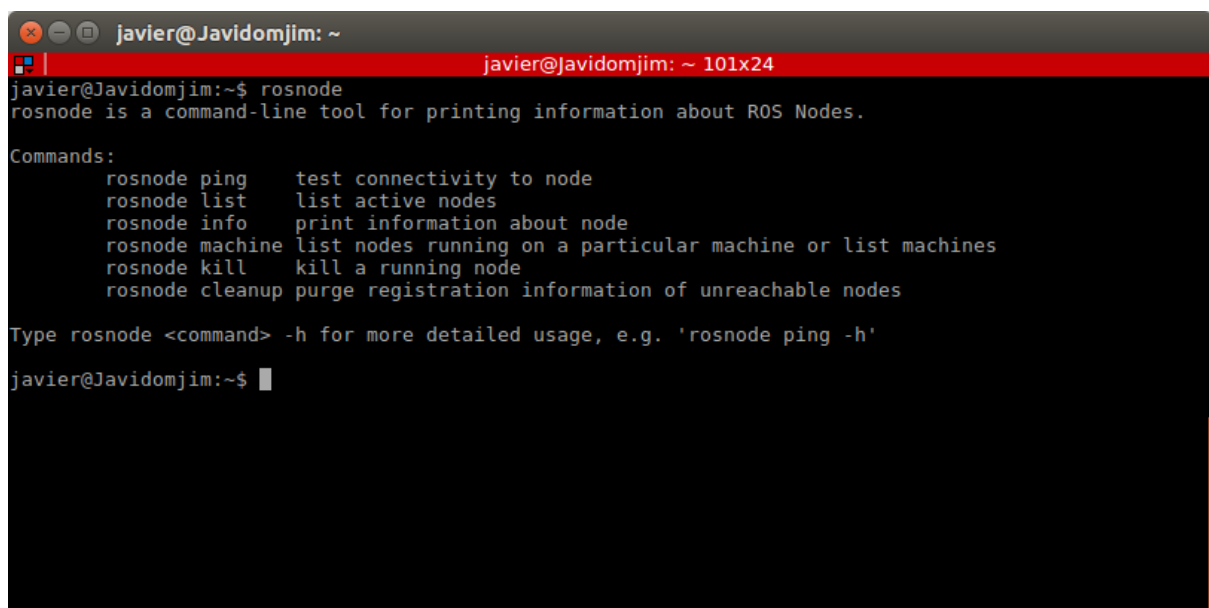
When a ROS node is ready to work after debugging, the next thing that must be done is launching the command in the command line to execute the node. This is accomplished by writing in the terminal the following:

```
roslaunch "name_of_your_workspace" "name_of_your_node" (1)
```

Then, the ROS node will be launching and executing the code that has been assigned. Whenever a node is launched, a master must be called first so it can manage every other node in the system. This is done using the following command in another terminal:

```
roscore (2)
```

There is a way to display information about every node active in the system. By using the command line tool called "rostopic" it is possible to achieve this target.

A terminal window screenshot showing the command 'rostopic' being executed. The terminal title is 'javier@Javidomjim: ~'. The command 'rostopic' is entered, and the output is: 'rostopic is a command-line tool for printing information about ROS Nodes.' Below this, a list of commands is shown: 'rostopic ping test connectivity to node', 'rostopic list list active nodes', 'rostopic info print information about node', 'rostopic machine list nodes running on a particular machine or list machines', 'rostopic kill kill a running node', and 'rostopic cleanup purge registration information of unreachable nodes'. At the bottom, it says 'Type rostopic <command> -h for more detailed usage, e.g. 'rostopic ping -h''. The prompt 'javier@Javidomjim:~\$' is visible at the end of the line.

```
javier@Javidomjim: ~
javier@Javidomjim:~$ rostopic
rostopic is a command-line tool for printing information about ROS Nodes.

Commands:
  rostopic ping      test connectivity to node
  rostopic list      list active nodes
  rostopic info      print information about node
  rostopic machine   list nodes running on a particular machine or list machines
  rostopic kill      kill a running node
  rostopic cleanup   purge registration information of unreachable nodes

Type rostopic <command> -h for more detailed usage, e.g. 'rostopic ping -h'

javier@Javidomjim:~$
```

Figure 3: Terminal screen showing rostopic command

As can be seen in the previous figure, a list of orders will come when rostopic is typed. Through them, it is possible to obtain a lot of useful information about any details needed to understand what is happening in the system.

As for the programming part of a node, as it was explained before, it consists of a standard script written in C++ or Python, that uses a particular function to design it as a node part of the complete system. This function works the same way in both languages. However, it may vary in syntax. Python will be used as the primary language to show every following example.

The invocation to designate a script as a ROS node is as follows:

```
rospy.init_node( 'node_name', anonymous=True) (3)
```

Where in the `node_name` field, it is possible to specify the name that will be receiving the recently created node. This is the name that will be appearing online for the other elements of the system.

The `anonymous` field indicates if the programmer needs the exact name of the node or not. Setting this to `true` means the node will be receiving a random number to the end of the code's name, so it may be unique. For example, if a node acquires the function of a driver, the two similar names could result in a problem because the node executed more recently will be replacing the older one.

Once a node has been declared, a shutdown function should be included as well. There are two different possibilities to do this. The first one requires an infinite loop that will not finish until the process is shut down:

```
while not rospy.is_shutdown() (3)
do something
```

The second way is using the function `rospy.spin()`, which makes the code sleep until a flag called `is_shutdown()` becomes `True`. There are many ways in which a node may receive a shutdown request, such as a `ctrl+C` request. So it is important to use it properly.

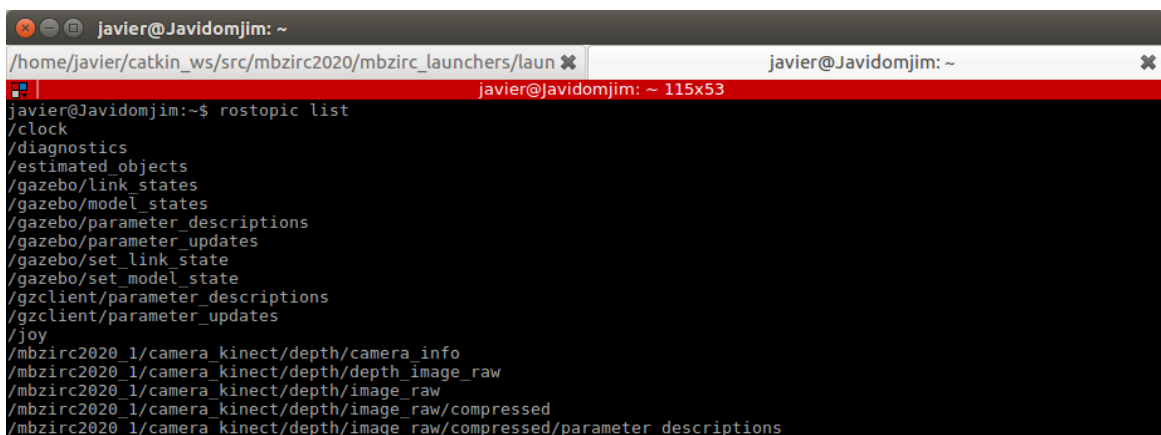
## 2.4.2 Topics

A topic is a bus over which a node can exchange messages. It has an anonymous publish/subscribe semantic, which can decouple the production of information from its consumption. To communicate with each other, a node will use a topic to publish a message or receive it. Using a topic creates a unidirectional and streaming communication.

Every topic uses a determined type that is assigned by the node publisher, and it cannot be changed during a communication process. If a node wishes to receive such a message, it must match the format of the topic message for a correct pick-up.

ROS topics support TCP/IP and UDP based message transportation. They are known as TCPROS and UDPROS, this one only works on Roscpp. By default, TCPROS is set. The node is the one who negotiates the transport type at the runtime.

If one wishes to know the list of topics a system has, by using the command line tool called `rostopic list`, it is possible to see all the topics available.



```
javier@Javidomjim: ~
/home/javier/catkin_ws/src/mbzirc2020/mbzirc_launchers/laun x javier@Javidomjim: ~
javier@Javidomjim: ~ 115x53
javier@Javidomjim:~$ rostopic list
/clock
/diagnostics
/estimated_objects
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gzclient/parameter_descriptions
/gzclient/parameter_updates
/joy
/mbzirc2020_1/camera_kinect/depth/camera_info
/mbzirc2020_1/camera_kinect/depth/depth_image_raw
/mbzirc2020_1/camera_kinect/depth/image_raw
/mbzirc2020_1/camera_kinect/depth/image_raw/compressed
/mbzirc2020_1/camera_kinect/depth/image_raw/compressed/parameter_descriptions
```

Figure 4: Terminal screen showing `rostopic list` command

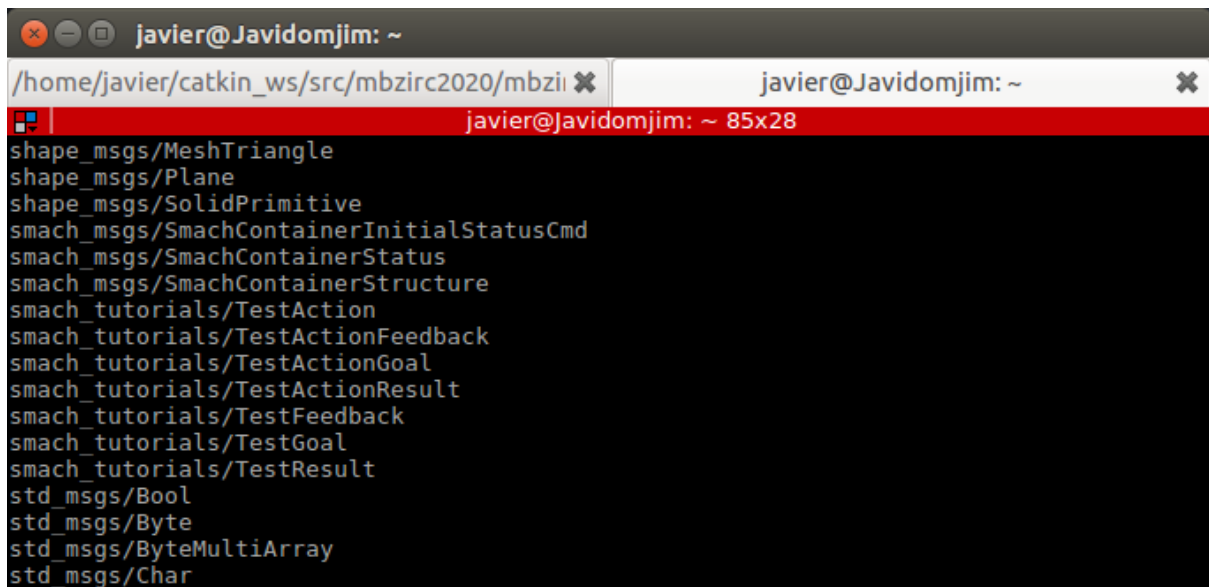
The previous is a list of the topics that conform to MBZIRC's Challenge 3 in the system simulation. As can be seen, there are quite a lot of topics. All of them are being published or subscribed by a node.

### 2.4.3 Messages

Topics are the bus for the nodes to communicate with each other. However, the way they exchange information is through messages published on topics. A message is just a simple data structure that comprises multiple typed fields. This is the simple way to establish a connection between two nodes.

A message file is a simple text file in which one can specify the data structure of a message, and they are stored in the `msg` subdirectory of a package. The message types use a standard convention for ROS, which is created with the structure: package name/name of the `.msg` file. To save a message, it is archived in a folder called `msg` to ROS to be able to implement it and translate it into source code by building the `CMakeLists.txt` file.

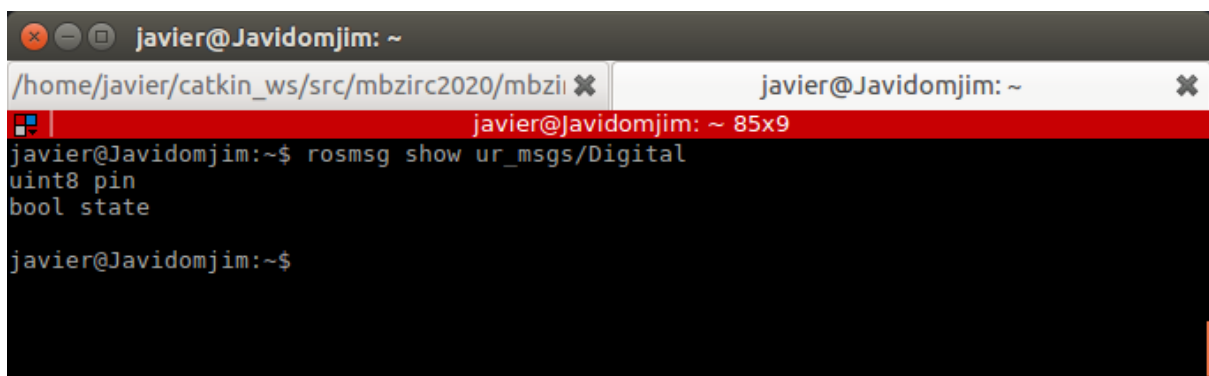
By using `rosmmsg` command-line tool, it is possible to display information about the ROS message type so that it is easier for the programmer to debug the code. Here is an example:



```
javier@Javidomjim: ~  
/home/javier/catkin_ws/src/mbzirc2020/mbzi x javier@Javidomjim: ~ x  
javier@javidomjim: ~ 85x28  
shape_msgs/MeshTriangle  
shape_msgs/Plane  
shape_msgs/SolidPrimitive  
smach_msgs/SmachContainerInitialStatusCmd  
smach_msgs/SmachContainerStatus  
smach_msgs/SmachContainerStructure  
smach_tutorials/TestAction  
smach_tutorials/TestActionFeedback  
smach_tutorials/TestActionGoal  
smach_tutorials/TestActionResult  
smach_tutorials/TestFeedback  
smach_tutorials/TestGoal  
smach_tutorials/TestResult  
std_msgs/Bool  
std_msgs/Byte  
std_msgs/ByteMultiArray  
std_msgs/Char
```

Figure 5: Terminal screen with ROS messages

As before, the messages showed in the previous figure belong to the MBZIRC's Challenge 3. To display the information about the definition of the raw message, by using `rosmmsg show -r 'msg_name'`, the message format will appear.



```
javier@Javidomjim: ~  
/home/javier/catkin_ws/src/mbzirc2020/mbzi x javier@Javidomjim: ~ x  
javier@javidomjim: ~ 85x9  
javier@Javidomjim:~$ rosmmsg show ur_msgs/Digital  
uint8 pin  
bool state  
  
javier@Javidomjim:~$
```

Figure 6: Terminal Screen showing the content of a specific message

As it can be seen in the figure, this message has only two types, one for an image type and another for a boolean state.

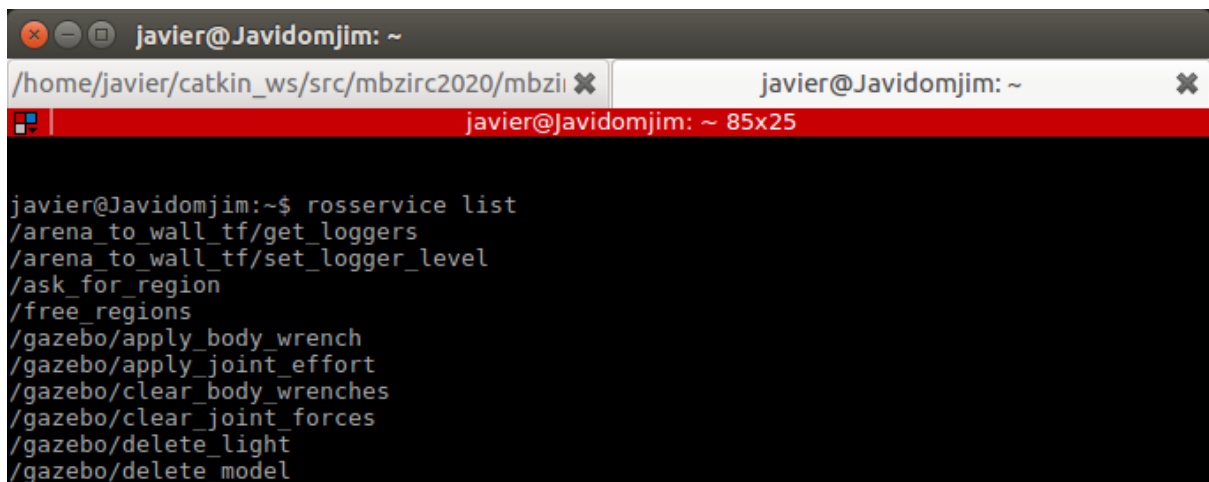
## 2.4.4 Services

Messages are a flexible communication when it comes to a one-to-one publisher/subscriber model. However, sometimes it is better to use a distributed system that includes the request/reply model that a service provides.

Basically, a service works as the following: a pair of messages, one for the request, and the other one for a reply will be forming the service in a providing ROS node. This one offers a service under a string name, and the client who needs the service will just call it by sending a request message and waiting for the reply. A service is defined by using a *srv* file.

Just like topics, services have an associated service type, which is the package resource name of the *.srv* file. To name a package, this format must be used: package name / service\_name. *srv* file.

If it is needed to visualize a list of all the services available in the system, using the command line *rosservice list* will provide such a list.

A screenshot of a terminal window on a Linux system. The window title is 'javier@Javidomjim: ~'. The terminal shows the command 'rosservice list' and its output, which is a list of service names. The output includes: /arena\_to\_wall\_tf/get\_loggers, /arena\_to\_wall\_tf/set\_logger\_level, /ask\_for\_region, /free\_regions, /gazebo/apply\_body\_wrench, /gazebo/apply\_joint\_effort, /gazebo/clear\_body\_wrenches, /gazebo/clear\_joint\_forces, /gazebo/delete\_light, and /gazebo/delete\_model. The terminal window has a red title bar and a dark background.

```
javier@Javidomjim:~$ rosservice list
/arena_to_wall_tf/get_loggers
/arena_to_wall_tf/set_logger_level
/ask_for_region
/free_regions
/gazebo/apply_body_wrench
/gazebo/apply_joint_effort
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
/gazebo/delete_light
/gazebo/delete_model
```

Figure 7: Screen Terminal showing a list of services

Those services that appear in the previous image belong to MBZIRC's Challenge 3

## 2.5 Gazebo and Rviz

To create and simulate all the entire system, it is imperative to use a robot simulator that allows the programmer to try and test different alternatives and run every scenario that approaches reality as much as possible. To achieve this goal, the Gazebo robot simulation will be used to complete all the three challenges proposed in MBZIRC Competition.

By using Gazebo, it is possible to test different algorithms, design any robot that might be needed, perform a regression testing on it, and train an AI system by using a realistic scenario.

Gazebo will also provide the option to simulate populations of robots in complex indoor and outdoor environments, so it will serve well to the multi-robot Challenge that is needed in this thesis.

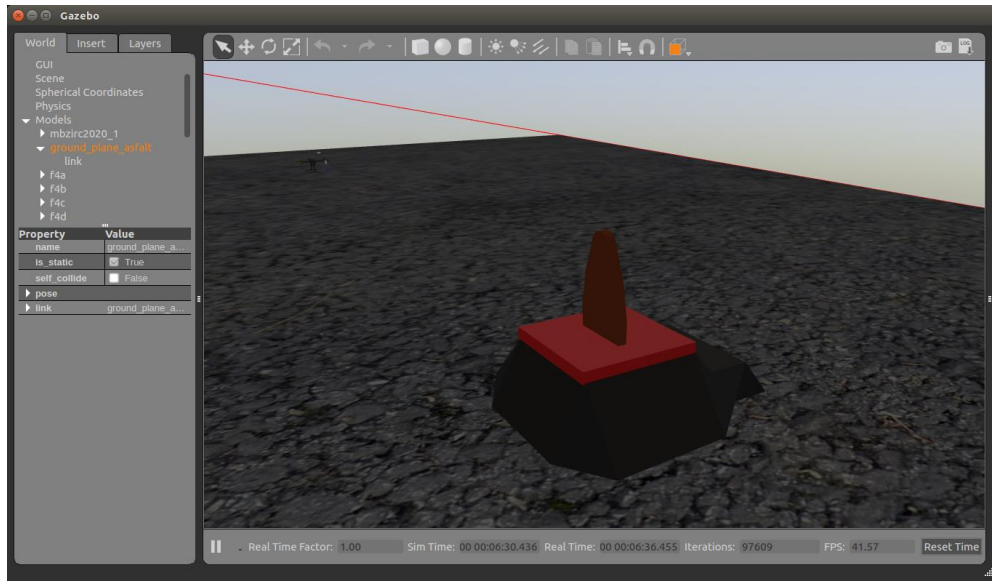


Figure 8: Gazebo screen simulator

However, it is also necessary to know what the robot is "watching" through its sensors. This is quite important for the programmer to understand whether it is being correctly executed or some things need to be changed. To fulfill this purpose, Rviz software will be used.

Rviz is a graphical interface based on ROS, and it allows us to visualize information coming from plugins or sensors that publish to topics. Rviz will read those topics so it can show what the robot sees.

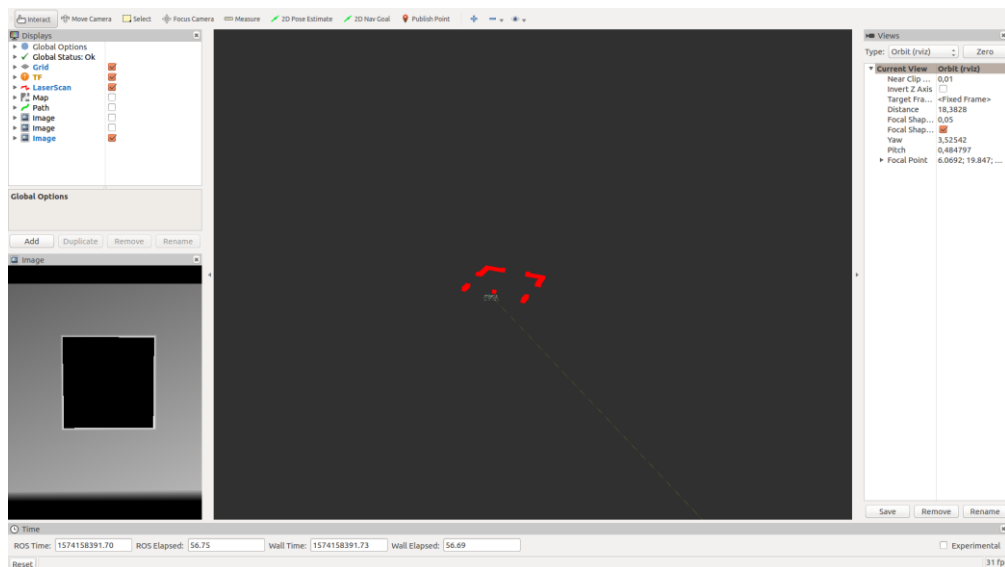


Figure 9: Rviz window

In the previous image, it is possible to see a drone equipped with a Rplidder laser scan and a depth camera trying to fly inside a building. Being aware of this kind of information, it is easier to picture a complete image of what the robot sees.

Combining Rviz and Gazebo, it will be possible to create a complete, functional system in which three UAVs and a UGV can cooperate and work towards achieving different objectives.

## 2.6 SMACH

State Machine, or SMACH, is a task-level architecture for an easy creating complex robot behavior. At its core, SMACH is a ROS-independent Python library to build hierarchical state machines. SMACH is a new library that takes advantage of very old concepts to quickly create robust robot behavior with maintainable and modular code.

SMACH is useful when a robot should execute some complex and ordinated tasks, where all possible states and state transitions can be described explicitly. This basically takes out the need to programming different modules to make systems do other things.

- A fast prototyping: The straightforward Python-based SMACH syntax makes it easy to prototype a state machine and start running it as fast as possible.
- Possibility of creating complex state machines: SMACH allows us to design, maintain, and debug large, complex hierarchical state machines. This functionality will be critical in order to develop a solution to the MBZIRC's Challenge 3.
- Introspection: SMACH provides full introspection in the state machines, state transitions, data flow.

Of course, it is possible to build a finite state machine using SMACH, but it is designed to do much more. SMACH is a library for task-level execution and coordination, and provides several types of "state containers." A container could be a finite state machine, but this container can also be a state in another container.

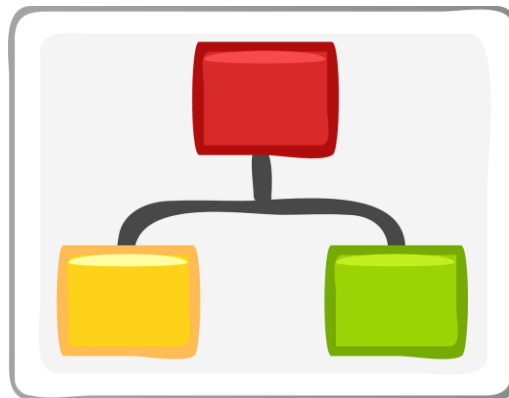


Figure 10: State Machine Concept

### 2.6.1 SMACH Concepts

- Outcome Semantics: For all SMACH containers, the interface to contained states is defined by the state outcomes. A state's potential outcomes are a property of the state instance, and must be declared before it is executed.

State outcomes may cause different things to happen in different types of containers, but what happens after an outcome is emitted is irrelevant from the perspective of a state. That means, outcomes can be considered local to a given state.

- User data: a container has a database that is used to coordinate and pass information between different states. This becomes useful when states compute some result or when they provide data from other sensors. This will allows such data to be held at the execution level and made available to other tasks or procedures.

Similarly to outcomes, the user data keys that are set and retrieved in each state are described by the state's input and output keys. These are also a property of the state instance, and must be declared before execution.



- Preemption: it is built inside SMACH. The State base class includes an interface for coordinating preempt requests between containers and contained states. Each container type has its own well-defined behavior for responding to a preempt request. This allows the system to both respond to termination signals from an end-user and to be able to be canceled programmatically by a higher-level executive.
- Introspection: SMACH containers can provide a debugging interface that allows a developer to set the full initial state of a SMACH tree. This is visualized with the SMACH Viewer. This includes both the initial state labels of each container as well as the contents of the user data structures at each level.

## 2.6.2 SMACH States

State can mean different things in different contexts.

In SMACH, a state is a local state of execution. It can also mean that it corresponds to the system performing some task. This is different from formal state machines, where each state describes not what the system is doing but instead describes a given configuration of the system.

This allows the user to focus on what the system is executing and the results from said execution, as opposed to naming the points between execution. States in SMACH correspond more to states in structured programming

Here is the list of available states in SMACH.

Class	Description	Outcomes
State	State base interface	None
SPA State	It is a state which has three used pre-defined outcomes	Succeeded/Preempted/Aborted
Monitor State	A state that subscribes to a ROS topic and blocks while a condition holds	Valid/Invalid/Preempted
Condition State	A state that executes a condition callback	True/ False
Simple Action State	A state which acts as a proxy to a simple <i>actionlib</i> action	Succeeded/Preempted/Aborted

## 2.6.2 SMACH Containers

Several container types are provided by the SMACH library. Different containers provide different execution semantics, but they can all be treated like states in other containers. They may have their own ways of specifying transitions for contained states, since transition means different things in different contexts.

All SMACH containers have the following properties:

- They contain a dictionary of states where objects were implementing the SMACH. State interface are keyed by string labels.
- They contain a user-data structure that all of their children can access.



# 3 GENERAL SOFTWARE ARCHITECTURE FOR MULTI-ROBOT COOPERATIVE MISSIONS

---

Once all the base tools have been defined and explained, it is time to review the general architecture of the challenges that compose MBZIRC's competition. Basically, it is formed by a central agent, several UAV agents, and different components. A review of every part will be performed.

First of all, an explanation of which is the different parts of the architecture will be detailed, indicating how they are related between them, so the reader can understand this chapter.

Before explaining these concepts, it is imperative to understand what the strategy to be carried out in the competition will be. In this Challenge, one UAV will carry blankets while a thermal camera will be pointing downwards. The other UAV will carry water deposits and the thermal camera pointing forward to detect facade fires. Also, it will be attached with a RealSense camera and a Rpliddar pointing forward. The UGV will carry the water deposit and a thermal camera.

## 3.1 Architecture Components

### 3.1.1 What is an Agent?

An agent is a collection of tasks that jointly create an agent interface that adds subscribers and publishers and which execution can be externally requested.

Each agent node is a ROS node that creates an AgentStateMachine and an AgentInterface. The agent node runs a SMACH state machine with a default task and transitions to a set of tasks that the UAV can be requested to execute.

Therefore, the agent must initialize the AgentStateMachine with the default task and a dictionary of additional tasks. This dictionary is created calling functions `add_task()` and `add_sub_task()`. Then, execute the `smach.StateMachine`.

### 3.1.2 What is a Task?

Tasks are stored as individual python modules. A task module needs to contain some aspects to enable requesting the execution of the task through a ROS service. If the task is just used for composition in more complex tasks, they are not needed.

Tasks should be simple, only to configure and coordinate components, i.e., they are mostly calling ROS services, publishing messages, and listening to topics in order to coordinate the task execution.

Tasks can add publishers, subscribers, servers, etc. to the agent interface. These are task-dependent, i.e., the callback is called just in case the task is active. Thus, multiple tasks can subscribe to the same topic or offer the same server, producing a task-dependent agent behavior. In the case of services, a 'default/inactive' callback is called if no task offering the service is active.

There are two ways to create more complex tasks: The first one is by implementing a task as another stat machine with other tasks; the second one is directly calling sub-tasks from another task, instead of externally.

Each implemented task module that is going to be called externally is added in the agent using `add_task()` method, and it needs to contain

- 1- Task class: implementing the task itself, for instance, as a `smach.State` or `smach.StateMachine`
- 2- `Gen_userdata` method: to convert data in an execution request to the input keys of the task.

- 3- Transitions dictionary: to specify matching from task's outcomes to [success, error], which are the only ones allowed in the TaskWrapper
- 4- DataType: data type to call the ROS service that triggers the task.
- 5- ResponseType: data type of the response in the ROS service triggering the task.

Tasks can add subtasks which can be then called in the body of their execute function:

- 1- A subtask is added using the `add_sub_task()` method with parameters: name (string, the name given to the subtask), parent\_task (usually self if called from a Task), child\_task (python module containing the subtask), task\_args (parameters that the subtask constructor requires)
- 2- A subtask is called from the parent task `execute()` body with the method `self.call_task('go_task',userdata)`

### 3.1.3 Central Unit Agent

The primary purpose of this entity is to allow the programmer to know which individual agents, these are UAVs, are available to operate and perform a new task. Also, the central unit agent is an action-client of every available UAV agent. This means that every agent simple-action-service is a *task* to be performed.

Additionally, the central unit is both a service-client of other agent services, such as "get\_cost\_to\_go\_to" service, and it is the subscriber of UAV topics, such as "data\_feed" and subscriber of other system components like "estimated\_objects". In the next figure, a zoom to the central unit is shown.

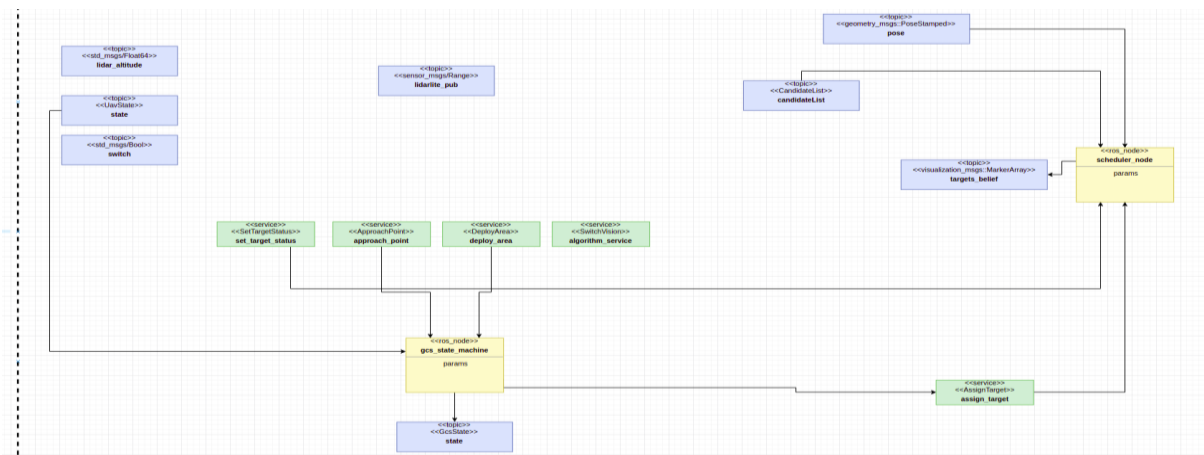


Figure 11: Central Unit Diagram

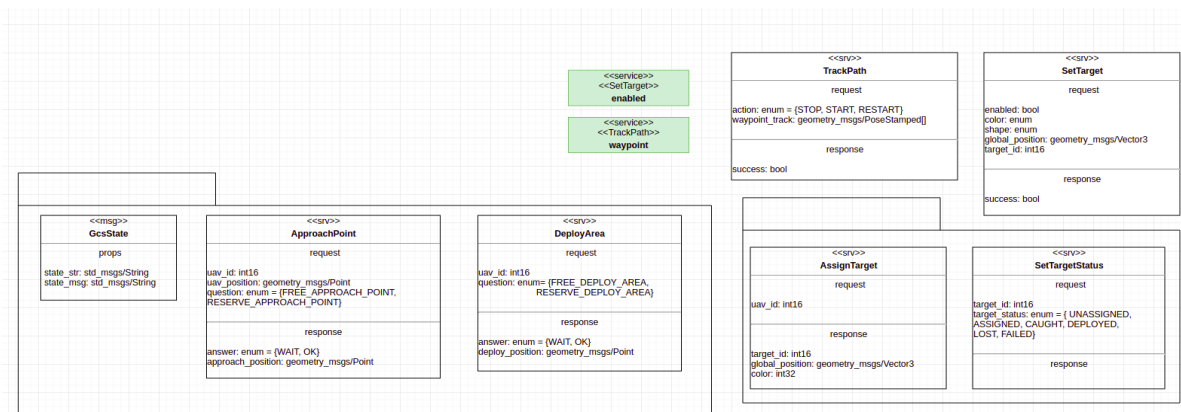


Figure 12: Central Unit services

With this information, what the Central Unit tries to do is to implement the cooperative behavior between all the different agents that are available in the complete architecture. That means, through the Central Unit, it will be possible to coordinate what every agent is doing at any time while commanding orders to them to do specified tasks. For example, in the Challenge 3 of MBZIRC's competition, the main objective is to put out fires with three agents. Two of them are UAVs, while the third is a UGV. In order to command which UAV should move to a particular position and coordinate where are the positions of the fires and which ones are already turned off, the Central Unit will play an immensely important role.

Now, a review of which tasks the Central Unit performs in the Challenge will be exposed. Its primary purpose is to monitor the available UAVs. It is a script implementing the following behavior:

- 1- The UGV is sent to search for fire into the building and extinguish that indoor fire. The position of the door is known and given by the organizers.
- 2- A UAV with a blanket will search outdoors. If a fire is found, it will be extinguished, and then a reset should be done to reinsert the blanket. Then, the same operation will be repeated, and then, the UAV will go home position. The extinguish task with the blanket consists of descending, centering the fire on the image obtained through a camera to a predefined altitude, and release the blanket.
- 3- Other UAVs are assigned a floor each to search for facade fires and extinguish them. Search for a facade is navigating to predefined positions and checking the thermal camera where possible fire spots are detected. The Extinguish task with water consists of a velocity control centering the circle of the facade on the image and activating the pump of the water tank until the deposit is empty.
- 4- When the water deposit is empty, the UAVs will land. After that, a reset should be called to refill the water tank deposit attached to the agent. Later, if there are floors not assigned because UAVs failed or there were no enough, they can get given these. The first floor is the last priority, as there are wind gusts.
- 5- Each UAV travels at a different altitude to avoid conflicts. Each access to its floor through a virtual elevator at opposite corners of the building. If no outdoor fire or facade is found, another fire search will be performed.

## 3.2 UAV Agent

When a UAV agent module is called, a set of SMACH states and state machines defines some *tasks* that the agent offers as SimpleActionServers. There is some standard code that could be grouped somehow, like services (servers/clients) and topics (publisher/subscriber). Actions allow non-blocking calls and are well integrated with SMACH. The agent only acts as a response to service calls, and it may use other system component topics, services, and actions (mainly: shared region manager, UAL, UAL action server, magnetic gripper).

A UAV node will perform different tasks, such as picking or placing objects. It is also possible to command it to go to a waypoint. In particular, in Challenge 3, it is necessary to shoot water to a fire spot in the facade. It may be needed to hover over a specific position, so it is also a critical task for a UAV node to be implemented.

The UAV node will be able to detect different types of objects such as fire in challenge 3, through the sensor mounted on it. This information will be available on a topic for other agents to be used.

UAV node subscribes to four topics to read the pose, velocity, odometry, and the state needed to perform all tasks mentioned above. Finally, by using a variety of services, such as `set_home`, `go_to_waypoint`, `land`, `take_off`, it is possible to command the UAV node to act as the programmer desires. This information is available in the next figure.

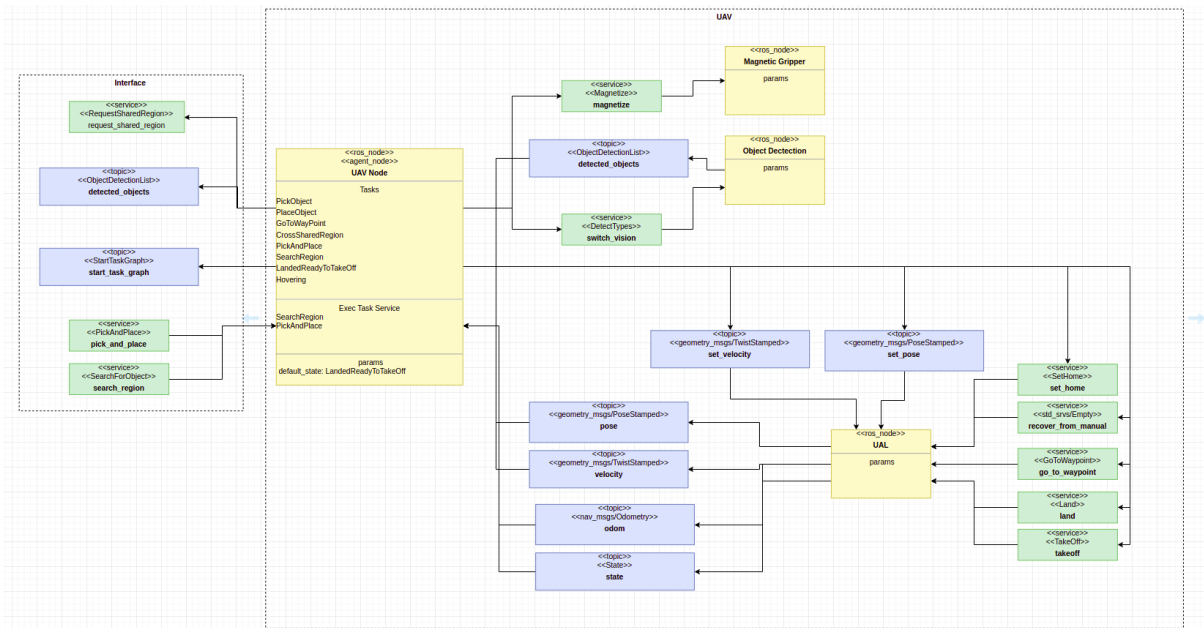


Figure 13: UAV Agent diagram in Challenge 3

The next critical component in these challenges is the UGV Node. While UGV will be supporting UAV's tasks, it can perform several functions such as searching fire in the ground floor area, and extinguish the fire it may encounter.

Between all those tasks that UGV performs are picking objects, placing them in a specific location. Going to a waypoint in a 2D plane, crossing the region that is shared between agents, or searching an area looking for fire objects to detect.

Whenever a task is completed, UGV will notify it through different topics to the Central Unit, so every agent is aware of the possible changes that occurred. For example, it can tell about a fire that has been recently extinguished.

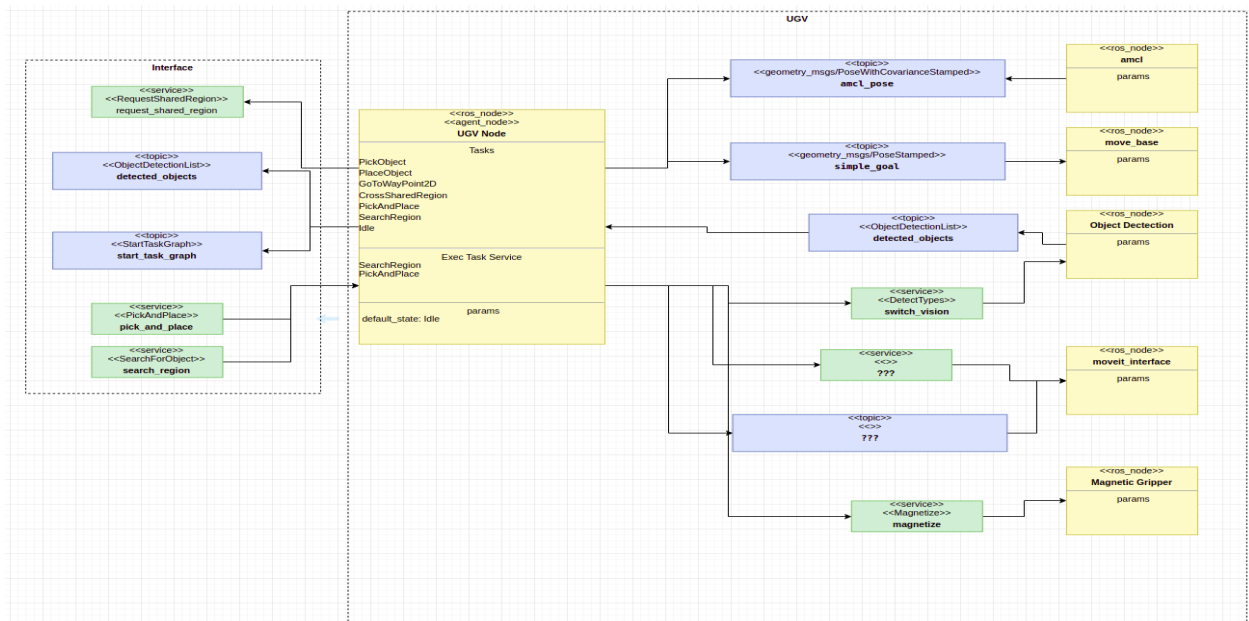


Figure 14: UGV task diagram for Challenge 2 and Challenge 3

### 3.3 ROS Components

Within the ROS components that form the general structure of all the challenges, an object detection, a fire detector, a circle detector, and a UGV navigation stack can be found, plus the critical UAL component. In the following lines, a short explanation of the functionality will be provided.

- Detection of the Fire

This ROS node will provide to the architecture information obtained through the agents about where a fire spot is detected, for both façade and outdoor fires. When a fire is detected, it will receive the UAV position plus a combination of the laser measurements in the fire direction.

As an input, the node will receive thermal images obtained through the thermal camera, and it will provide the fire detection, given by an algorithm that includes OpenCV libraries, to process images in real-time.

- Circle detector

This ROS node will be used to run a simple segmentation on the depth image and detect circle holes in the façade fires. This node will allow agents to position themselves when shooting water and to determine, combined with the fire detection, if the fire spot obtained is the right one.

- UGV navigation stack.

This module is used to localize and navigate the UGV with a path planner and local collision avoidance. The outdoor localization is based on GPS + odometry + IMU. While the indoor localization is based on odometry and a 2D laser.

- UAL component

UAL is an abstraction layer for unmanned aerial vehicles, and its purpose is no other than abstract the user from the particularities of each autopilot, offering a common interface. It will, then, provide tasks that need to be done in a UAV, such as landing, setting a home point, taking off, going to a waypoint, and setting the velocity. By using it, the complete system will drastically improve the efficiency and the computational load.

As UAL is a ROS node, it will receive messages through five services, which are `set_home`, `recover_from_manual`, `go_to_waypoint`, `land` and `takeoff`. Whenever the UAL node receives any message, it will publish a message answer to different topics indicating the velocity, the odometry, the status of the vehicle, the pose, etc.

In the next figure, one can have a better perspective about how UAL node works and which topics does it publishes to.

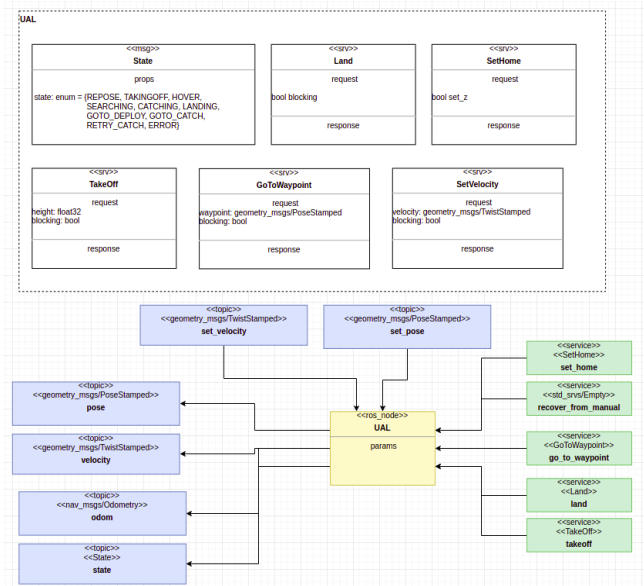


Figure 15: UAL diagram



# 4 IMPLEMENTATION OF A FIREFIGHTING MISSION WITH A MULTI-ROBOT TEAM

In this chapter, an in-depth review of MBZIRC's Challenge 3 will be performed. In this section, it will be described the exact mission that UAVs must accomplish, the world conditions, the problems that must be resolved, a list of more specific ROS components that are needed, and will be implemented, as well as sensors included in every agent node.

Hardware and software will be described in great detail, including UAVs and UGV agent tasks, a description of how the coding has been implemented and, in general, how tasks are expected to perform.

## 4.1 Challenge 3 Description

Challenge 3 of MBZIRC consists of an urban building fire fighting scenario. The main objective of this Challenge is to use two UAVs and a UGV to put fires out. These fires are placed outside the building, inside the building, and at the outer walls or facades.



Figure 16: Simulated building and real building

In the previous image, the main building is displayed. As can be seen, it consists of a three-floor building without windows so UAVs can get in to put out the fires. While the first and second floors can be accessed, the ground floor will also have a door. About the dimensions, every window will have a 2m x 2m size, while the door will have a 2.5m x 2 m size.

Fires that are situated outside the building are simulated fires that are created with a heated plate and a red silk, so they must be put out with a blanket carried by a UAV. However, fires placed inside the building are generated with a flame generator, in order to avoid possible dangers. Lastly, those fires that remain on the walls outside the building are real. To simulate that they are put out, UAVs should shoot water inside a hole, filling a container.

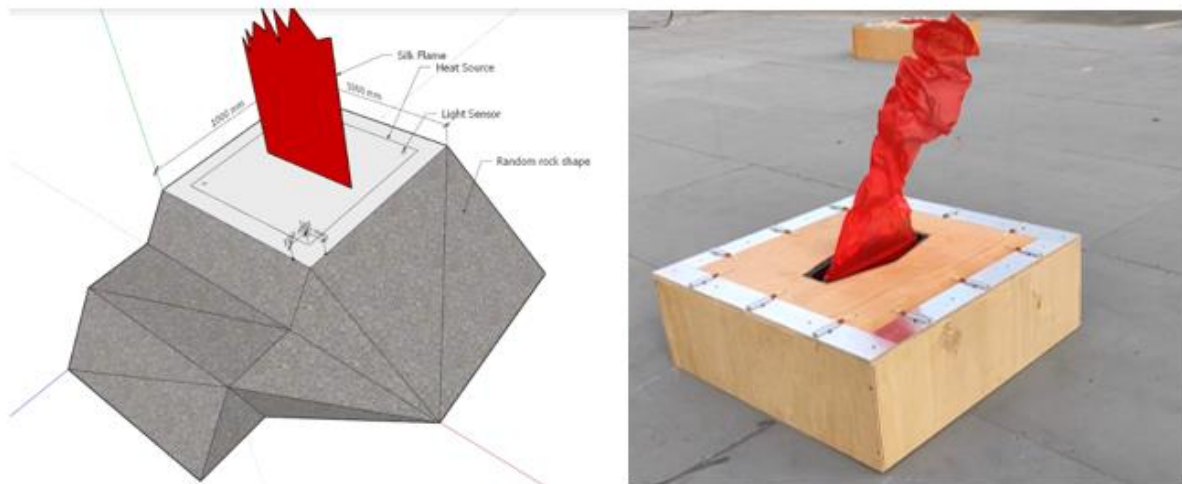


Figure 17: Fire plate with a red silk, simulated and real

These fires will be simulated both thermally and visually, and they will be heating elements with a 60mm x 36mm size that will provide 120 degrees.

The number of fires is eight in total: two of them will be placed on the ground outside the building, three of them are set inside the building, one on each floor, and the three remaining fires are placed externally on the wall, one for each floor.



Figure 18: Fire spot placed in the wall

In the previous image, one can see how the external fires placed in the outside walls will be. These are simulated with propane gas and a heat plate. To be able to score points, UAVs will have to put out the outside simulated rock fire plate with a blanket, while the fires placed in the wall and inside the building, must be put out in a simulated way by filling water. While operating, UAVs can refill their water canisters during a reset period. This fact will not count as a penalty. In the next list, it is possible to review the general parameters of this Challenge.

Parameter	Specification
Number of UAVs per team	Maximum of 3
Number of UGVs per team	Maximum of 1
Arena size	50m x 60m x 20m
Tower height	18m
Location of simulated fires	Up to 16m in height, inside the arena
Environment	Outdoor and indoor
Mode of operation	Autonomous, manual is allowed but penalized
RTK/DGPS positioning	Allowed but penalized
Challenge duration	20 minutes
Extinguisher types considered	Water carrying container up to 1-3 liters
Maximum size of UAV	1.2m x 1.2m x 0.5m
Maximum size of UGV	1.7m x 1.5m x 2m

Table 1: Parameters specification of the Challenge

## 4.2 Mapping with Sensors and Hector SLAM

No matter whether it operates outdoors or indoors, every agent node needs to be able to locate itself in the arena with an exact position, as well as to reckon the place. For this purpose, a [Rpliddar](#) will be attached to every UAV to locate any possible obstacle that might encounter.

Also, every UAV must be able to locate itself whenever it is, whether it enters inside the building or not. Due to the lack of the GPS indoors, it is imperative to create a map using the attached *Rpliddar* to understand and locate where the fires and the possible obstacles might be if the UAV gets inside the building. Thus, SLAM techniques will be implemented within the system so that UAVs can rely on information obtained in their environment.

At the beginning, the main idea was to assign a UAV to different floors. Provided that the maximum number of UAVs allowed is three, and there are the same number of floors, every agent will be able to create a map of their own flight level. However, in the next chapter, it will be discussed how the strategy was really managed. By

now, it will be explained how UAV would manage to fly in the indoor area if that case is needed.

SLAM techniques are usually applied over a 2-Dimension map with a ground-based vehicle to map a zone. However, using a conventional SLAM over a UAV will produce quite a lot of problems due to noise and a 3-Dimension environment. A map produced in these conditions will not be useful for UAVs to adequately reckon their assigned zone. Thus, it will be needed a better iteration of a SLAM that provides a transformation from 3D to 2D. That is why Hector SLAM will be a crucial element in this task.

Hector SLAM consists of a mapping technique that uses 3D laser measurements to build a 2D map integrating them. Hector SLAM uses a node called Hector\_mapping for learning a map of the environment and, try, simultaneously to estimate the platform's 2D pose at the laser scanner frame rate. In the next image, it can be seen all the potential frames of interest in a simplified 2D view of a ground-based vehicle, leading to pitch and roll motion of the platform.

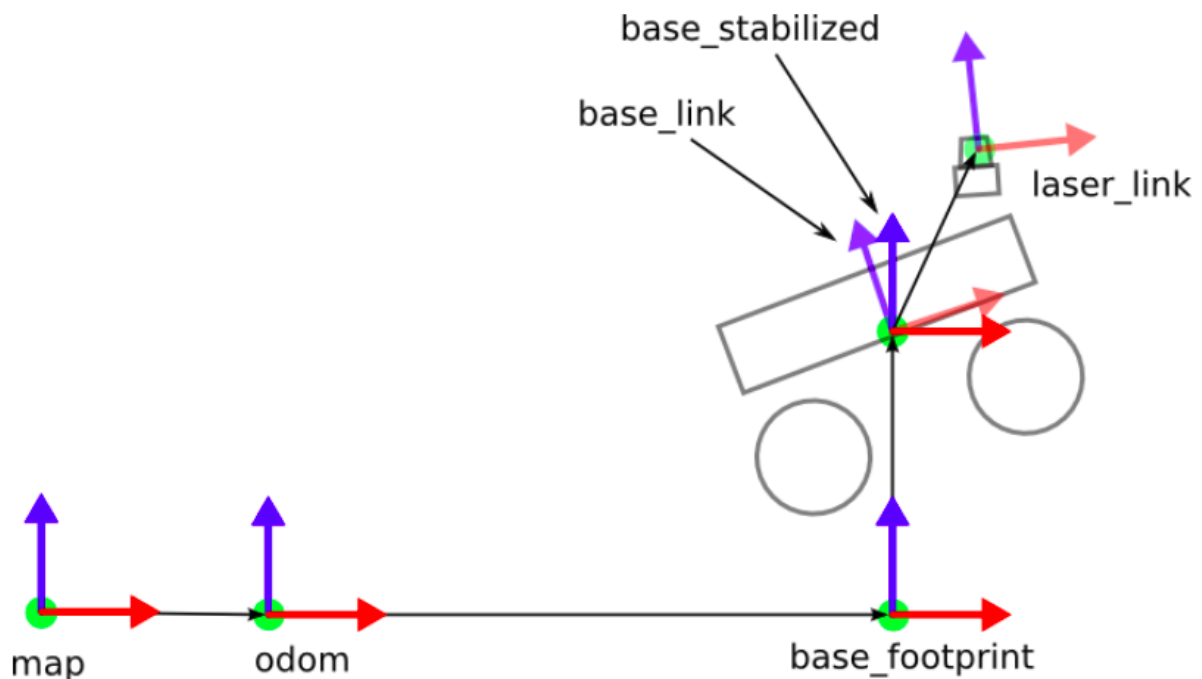


Figure 19: Diagram of Hector SLAM transformations

Between odom and base\_link frames, base\_footprint frame will be providing no height information and represents the 2D pose of the robot. The frame base\_stabilized will add information about the UAV height relative to the map or odom layer. The base\_link frame is attached to the UAV, and it adds the roll and pitch angles compared to the base\_estabilized frame. For this transformation, a system estimation of the vehicle attitude can be used.

Thus, to implement Hector SLAM in the system, this package will be needed along with every node. Due to the high number of processes that must run in the complete Challenge, SLAM mapping will be the last to be executed. This will help to avoid any kind of possible problem that may appear when launching other nodes. To run this node, it is imperative to build a correct transformation frames tree. In the next figure, one can see the adequate configuration.

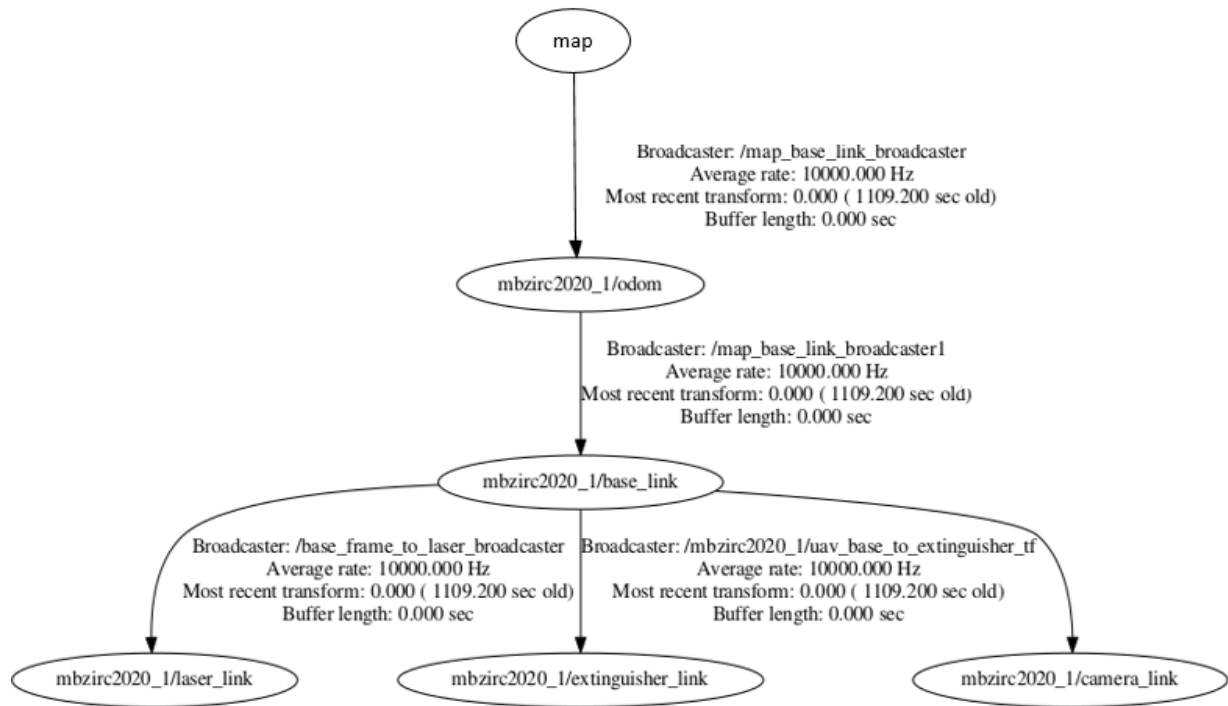


Figure 20: Transformation Tree for Hector SLAM

As it can be seen, the transformation goes from map to MBZIRC2020\_1/odom, which refers to the odometry of UAV agent node one. From this frame, base\_link will be the next frame and from this last one, laser\_link will be pending. This is the correct way to establish the TF tree. Otherwise, Hector SLAM will not be mapped correctly, and many different errors will be appearing.

Lastly, what it is expected to appear is a map of the area where UAV's lasers show where obstacles are located in the surroundings of the UAV.

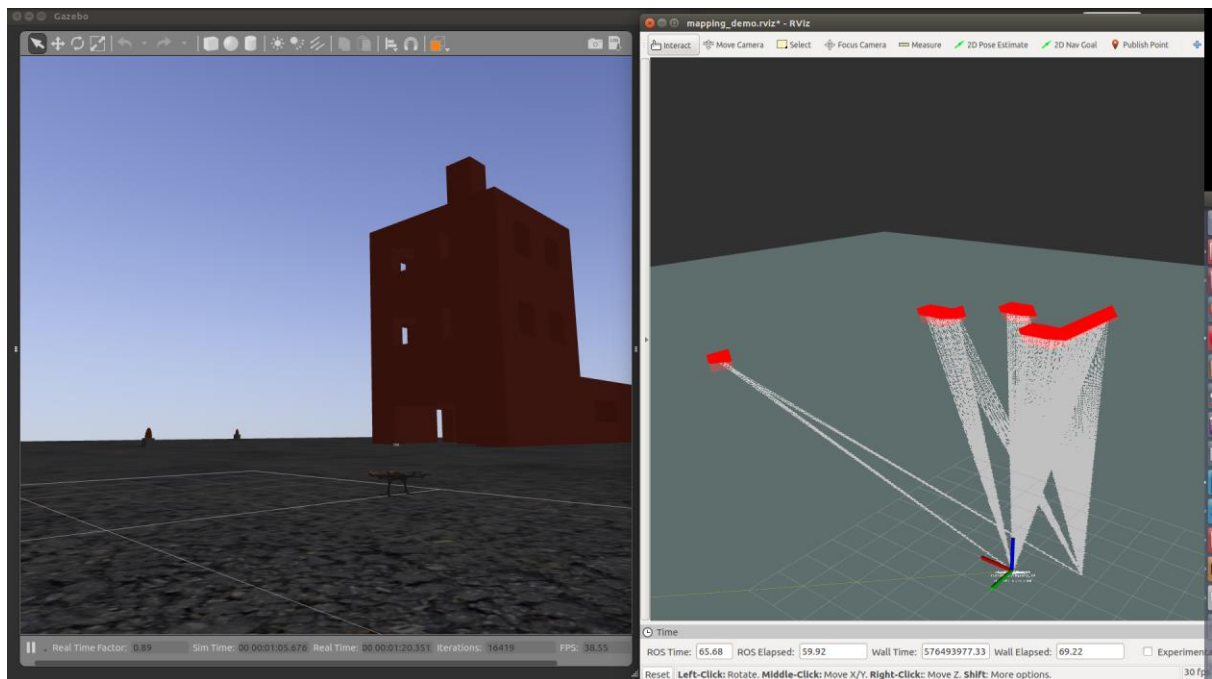


Figure 21: Gazebo and Rviz simulation for Hector SLAM

As it can be seen in the previous image, the left side is the Gazebo simulation in which the agent UAV is waiting to be launched. It can be seen a preview of Challenge's three MBZIRC building a set of simulated fires along the landscape. If Hector SLAM is launched, it will be expected to see a similar map that is shown on the right side of the image. In this one, it is possible to understand what the agent detects within its Rpliddar range, and how it maps the obstacles when those are detected.

The red lines on the right side of the image represents the walls and obstacles that the agent is detecting. The white areas that are projected to these obstacles represent the map that is currently being created.

By using Hector SLAM, in the simulation in chapter 5, it will be shown how it works with different timestamps, so the progress in the construction of the map can be appreciated.

### 4.3 Fire Detection in Walls and on the Arena

In order to detect any kind of fire placed in the arena, a thermal camera will be needed to be attached to all the agents. Through this camera, every agent will be able to spot fires, whether they are situated outside or inside the building. The main idea is to attach two thermal cameras, one will be directed toward the ground, while the second one will be looking to the front. By doing this, agents will be able to spot any kind of fire, whichever the place they are situated.

The thermal camera that will be used is a TeraRanger Evo Thermal 33, which has a 32x32 PX and can detect fire, in theory, to 10 meters of distance and provides a sampling rate of 14 Hz. In practice, the better range to detect without any false positive should be to 2 meters of distance maximum. The camera also provides a field of view of 33° with a weight of 9 grams. This will be a great advantage at the time to mount it on the UAV.



Figure 22: TeraRanger Evo Thermal 33

Due to the challenging conditions to simulate fire inside the Gazebo platform, to test and prove how the thermal camera works and make it suitable for the purposes of the Challenge, different recordings of a real mission were used to determine if the UAV was able to detect fire.

Using the rosbag feature, it was possible to obtain a thermal record that allowed to work and process the images obtained during different missions performed. Thus, the main idea is to locate a fire in the image, mark it with a contour, calculate its centroid and send that information through a message so every agent can be aware of the presence of fire along with the exact location and other parameters. In the next section, a more precise explanation will be provided along with a code debriefing.

What it is expected to be captured in the image will be something like the next figure, in which one can see different fire locations along with their centroids.





Figure 24: Intel RealSense D435i camera

A depth camera will be an immensely important asset to proceed to the inner part of the building. Thus, it will be implemented in simulation through Gazebo. The main idea is to detect windows in a first approach to the building. After they have been detected, via depth camera or Rpliddar sensor, an algorithm will be applied in order to position the agent over the correct spot so it can enter through the space window to navigate inside the building. What is expected to be seen is an image like this one.

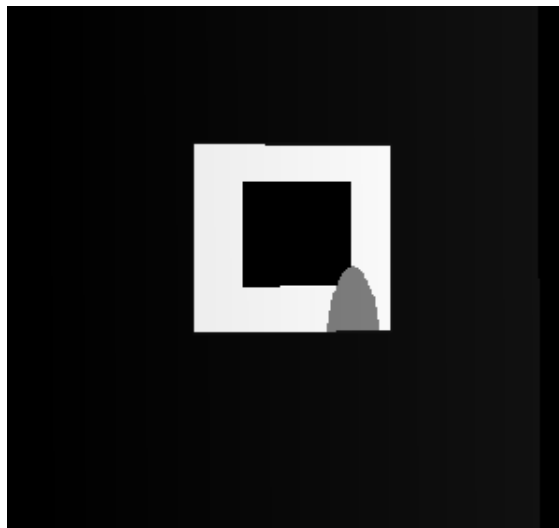


Figure 25: Sensor Camera inside simulation in Gazebo

As it can be seen in the previous image, the agent is getting closer to a window, and it can see what it is inside the building. However, although it is possible that entering the structure might not be necessary to achieve a better score in this Challenge, the depth camera will also be crucial when detecting the outer outline of the fire.

#### 4.4.1 Fire Pipes Detection: Circle Detection

External fires, the ones that are set in the building walls, are constructed as a ring of pipes that emits fire flames to the exterior, along with a fire resistor in the inner circle. However, detecting that a fire is active and approaching it is not as simple as it may seem.

In order to mark a fire as active and shoot water to it, it is imperative to reposition the UAV in the right spot, aligned with the fire spot. To do this, using the depth camera along with a circle detector algorithm, it will be possible to position the UAV correctly. Nevertheless, using only this algorithm may cause problems, since the fire spot in the wall is surrounded by two more circle-like shapes.





Figure 26: Competition Fire Spot

As can be seen in the previous image, in the middle, it is placed the fire ring, and on the right and left side of it, it is possible to see two circle-like objects. In order to overcome this problem, using both the circle detector and the fire detector, it will be possible to align the UAV with the water canister.

#### 4.5 Fire Extinguishing

As it has been explained before, there are two types of fires. Some of them are placed in the ground while the rest are set in walls, whether indoor or outdoor.

However, in order to put them out, it is not possible to use the same mechanism. It is imperative to apply a different method adapted to the fire spot location. With that purpose, fires placed on the ground will be put out with a fire-proof blanket.

By using this method, the blanket will be attached to the UAV and will be hanging from it. When the order to shoot the blanket is active, the mechanism will be opening just like an inverse umbrella, so it can easily eliminate the fire.

In order to put out the vertical fires, that is, the ones that are placed in the wall, a canister that is situated inside the wall must be filled with water. So, UAVs have attached a container and a barrel so it can shoot this water inside the wall. The score will be higher as more water fills the canister.



Figure 27: UAV Agent shooting water to the fire spot detected

The previous methods must be included within the complete system, and one UAV will have the blanket, while the other one will be carrying the water containers, along with the shooting water mechanism.



# 5 EXPERIMENTAL RESULTS

In this section, a closer approach of the code, the complete simulation, and the final conditions found in the MBZIRC's Challenge 3 will be displayed and explained. Also, records of the simulation created to solve the Challenge will be shown and commented, along with the reason of the solution applied to the particular problem.

Every result and shown in this chapter comes from the simulations of a mock-up created to emulate the same conditions that would be found in the real competition. And, of course, some of these results come from the real arena and the Challenge itself.

Once the simulation of the Challenge has been explained in detail, the thermal camera code will be described as well. In this particular segment, it will be shown how the camera works, what is showing and detecting, and what is the message and format that is sending to other nodes.

Finally, some figures of the real competition will be presented, along with detections and the actual conditions that were found in the real performance.

## 5.1 Challenge 3 Simulation

Using Gazebo and RViz, it will be possible to emulate the conditions that will be found in the real competition. By gathering all the information provided by the organization, a quite accurate simulation was created in which many different test flights, missions, and sensor testing were performed.

The next figure shows an image of the simulated world that tries to emulate the Challenge conditions. As it can be seen, horizontal fires are placed on the ground in the form of the shape that was provided in the information. Also, the vertical fires that are already set in the building.

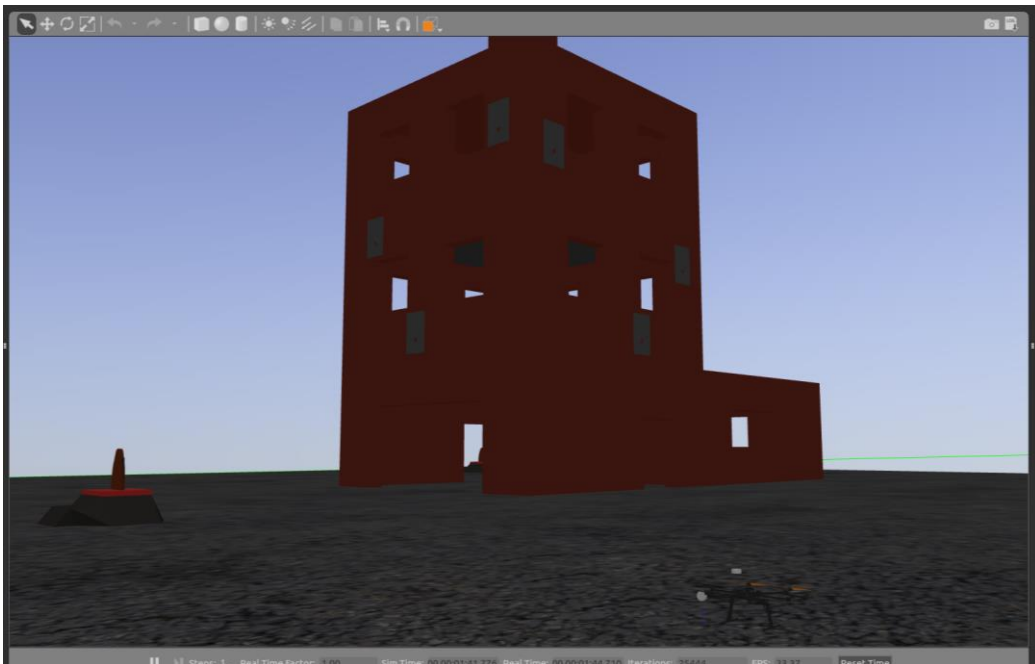


Figure 28: Simulation Building

These vertical fires do have a fixed position, so it will be much easier to determine where to send the UAV to check if fire is active or not. Of all the vertical fire positions, not every one of them is active. Thus, to check if they are, UAVs should fly to that position and determine the state using fire detection and circle detection

algorithms.

Also, in the simulation, it can be seen that the shape of the building tries to emulate the same building that was provided in the information. It consists of a ground floor that possesses three windows and three doors, and two more floors, with only windows in which a UAV can enter in.

Two UAVs are placed in the simulation, and they are equipped with a Rpliddar, and a depth camera. These will be used as was described before. The first one to calculate the distances on the horizontal plane and the other one to position the UAV with the circle detector over the fire.

Unfortunately, Gazebo does not have a way to simulate a fire in their inner system, so these nodes were tested outside the simulation. In the next section, it will be described in more detail.

## 5.2 Hector SLAM Implementation

Although, putting out fires in the inner side of the building was not a priority task due to significant difficulties compared to the ones that laid outside, implementing this technique was an alternative that was set if more score was needed in the real Challenge. Thus, in this section, it will be described with more details on how this SLAM was implemented, along with some code that was necessary to add in the inner parts of the system.

In order to implement Hector SLAM in the system, a specific launcher with the name of c3\_slam.launch was created to execute this mapping technique. Along with the installation of the package, a SLAM launch file is provided. In it, the adequate transformation, and the proper links must be set. In the following, it is possible to see which lines of the code must be changed to set Hector SLAM in motion.

```
<launch>
  <arg name="tf_map_scanmatch_transform_frame_name" default="scanmatcher_frame"/>
  <arg name="base_frame" default="base_footprint"/>
  <arg name="odom_frame" default="nav"/>
  <arg name="pub_map_odom_transform" default="false"/>
  <arg name="scan_subscriber_queue_size" default="5"/>
  <arg name="scan_topic" default="scan"/>
  <arg name="map_size" default="2048"/>
```

Figure 29: Fragment of code - Arguments to Hector SLAM

In the previous fragment of code, it is possible to see that the changes that have to be done are quite simple. In the list, it is necessary to change the name of the arguments to the adequate parameters. For example, one change is to set the scan\_topic to the name of the scan that is created with the Rpliddar laser when the UAV agents are ready.

```
<!-- Frame names -->
<param name="map_frame" value="map" />
<param name="base_frame" value="$(arg base_frame)" />
<param name="odom_frame" value="$(arg odom_frame)" />

<!-- Tf use -->
<param name="use_tf_scan_transformation" value="true"/>
<param name="use_tf_pose_start_estimate" value="true"/>
<param name="pub_map_odom_transform" value="$(arg pub_map_odom_transform)"/>
```

Figure 30: Fragment of code - Frames and Transformation

Among other needed changes, it is imperative to change the map\_frame parameter to the name of the map, to avoid overwriting information given by different frames. Also, it is needed to add the use of the transformation

trees by setting them to true.

The complete code can be found in the appendix at the end of this document, if the reader wishes to check out all the changes needed. Once every parameter is set and ready, it is time to launch the simulation to visualize what that UAV agent is perceiving through its sensors.

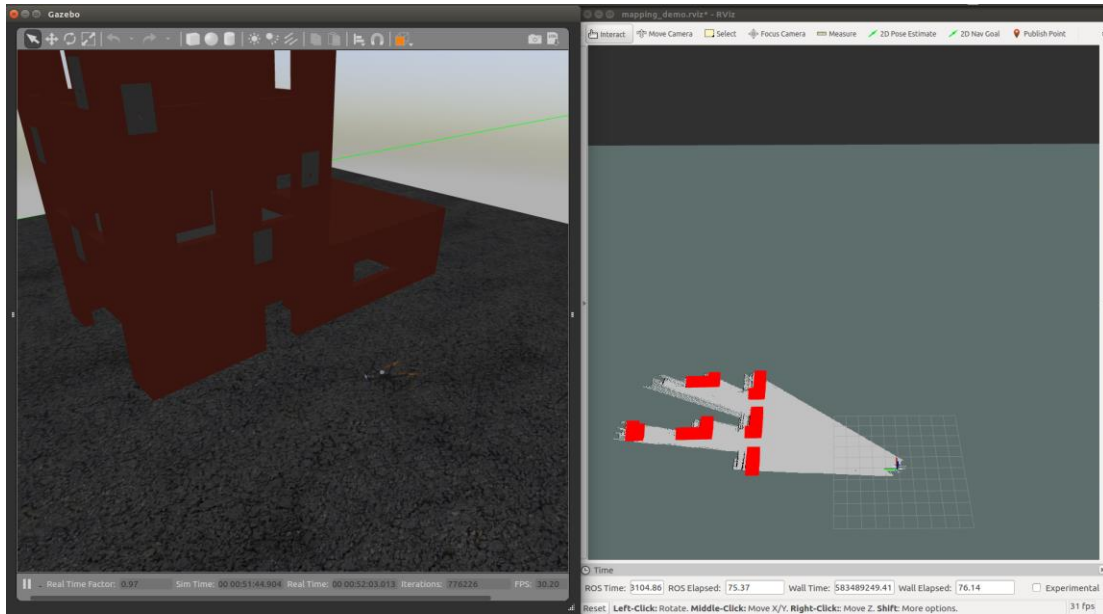


Figure 31: HECTOR SLAM simulation in Gazebo

As it can be seen in the previous figure, the left side of the image corresponds to the simulation run on Gazebo, whereas the right side is the sight of the sensors attached to the agent. In the simulation, the UAV agent is placed at the height of the first floor, and it is detecting two of the frontal windows and part of the inner building.

While the agent is advancing through the simulation, the map will be created with the information provided by the Rpliddar.

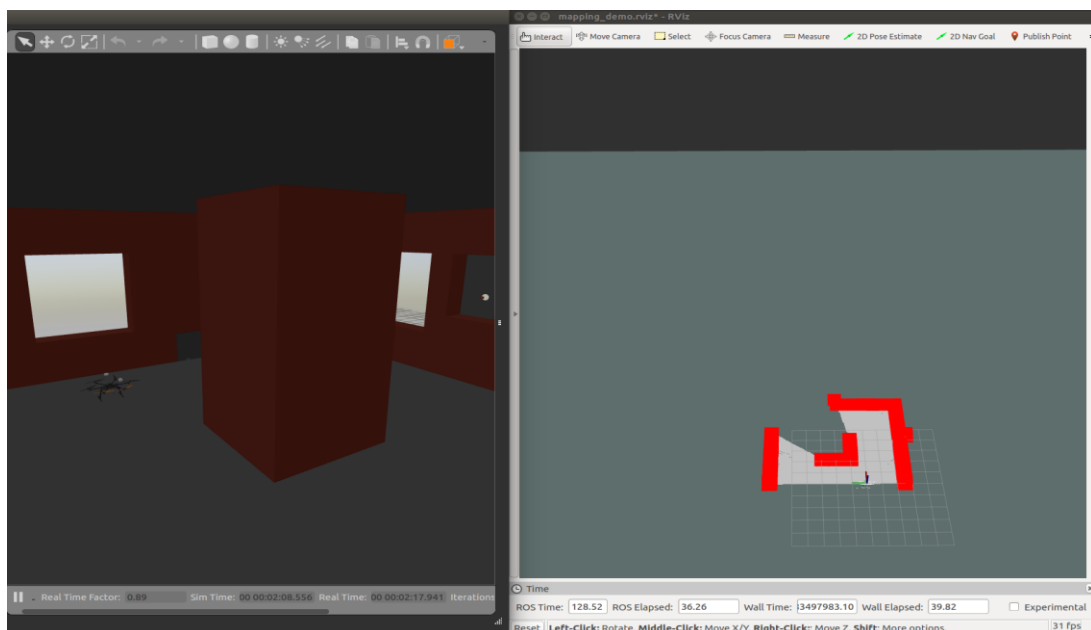


Figure 32: HECTOR SLAM inside the building

By doing this, it is possible to locate the agent and spot fires with the combination of the fire detector and the circle detector. However, in reality, this method was not used due to the challenging complications that produces. Detecting fires in the inner part of the building is a very complicated task that does not really deserve the effort compared to the detection of fires outdoors. The better approach is to look for the fires that are outside and trying to put them out instead.

This technique, although it was implemented in the final solution, was not used in the real competition.

### 5.3 Thermal Camera Implementation

In this section, it is provided a detailed explanation about how the thermal camera is working along with the code and what is supposed to perceive when the fire is being observed. As it already has been explained, every agent will have two thermal cameras, one will be pointing downwards, while the other one will be pointing to the front. This is done to detect fires in the arena.

First of all, it is important to show a flowchart of how the operation to detect fire is made. In the next image, it is possible to see how everything is structured.

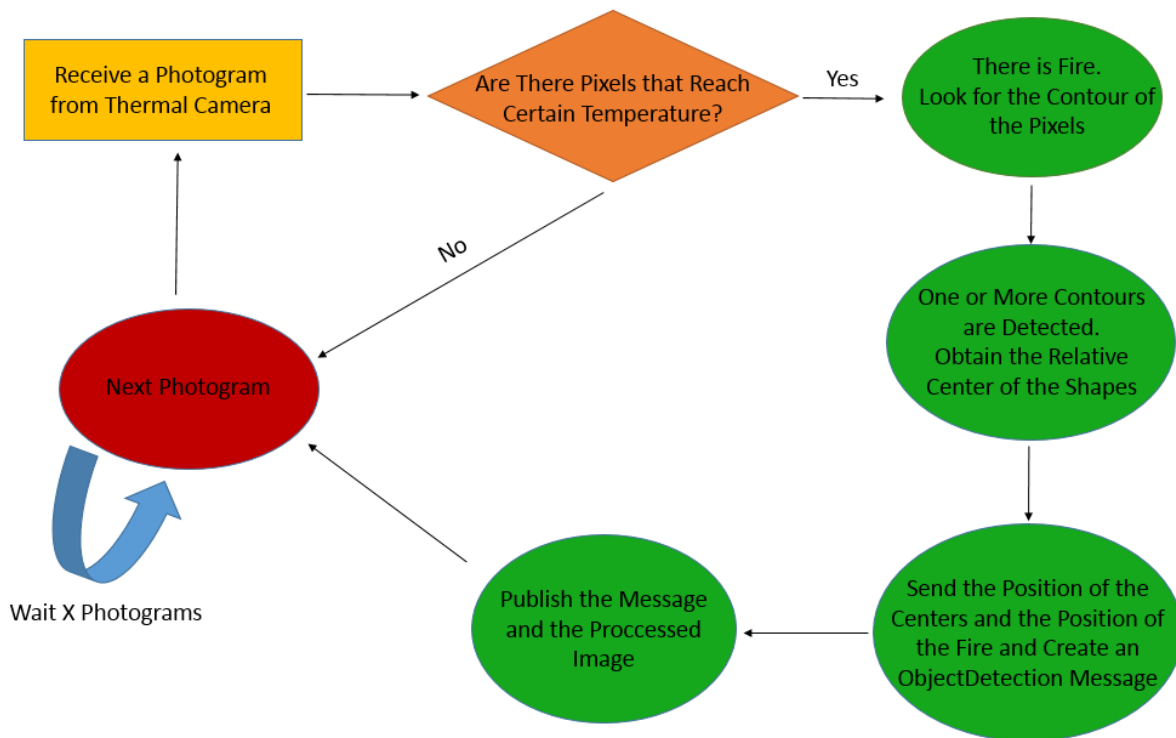


Figure 33: Thermal Node Flowchart

Apart from direction, before launching the code, different parameters can be set to specify essential data such as the thermal threshold that it is desired to detect. By changing this parameter, the camera will perceive pixels whose temperature is higher than the established number. This will allow a correct detection of the fire, whether it is placed in a wall or on the ground arena.

Another critical parameter is the one named "angle\_amplitude." The purpose of this one is no other than set an angle of detection projected to a wall using Rpliddar measurements. While Rpliddar works by launching a 2D scan in a circular area like the one in the next figure, the parameter angle amplitude allows us to read only a particular angle arc, which is determined by this value. This is done to calculate the exact distance between the UAV position and the wall, which contains the fire. Thus it is possible to obtain a better measure of the distance between UAV and the fire spot.

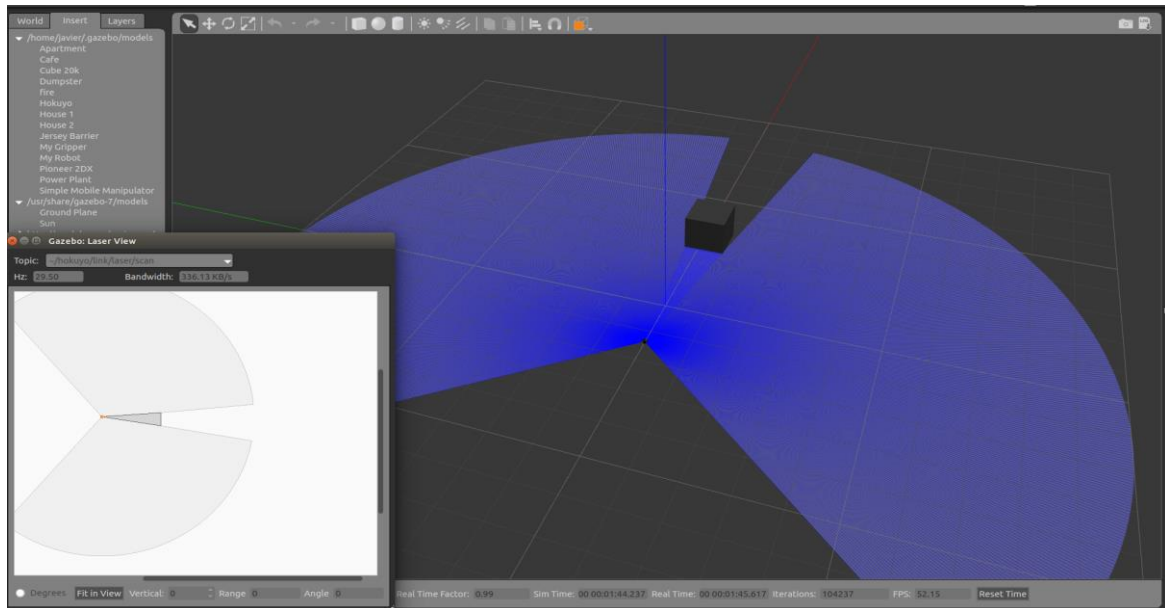


Figure 34: Gazebo Simulation of the laser

In the figure, it is possible to see a Rplidar placed on the ground that is detecting a box within its range. In the window found in the left corner, it can also be seen the sensor data. By applying an angle increment of 5 degrees, every laser measure from  $-5^{\circ}$  to  $5^{\circ}$  will be collected to determine the exact fire location.

Also, a UAV id (identifier) must be assigned to the code. This is done to identify which agent is sending fire messages detections and making easy determine which one is detecting fire.

All of these parameters must be set before launching the code. For this purpose, a specific launcher was created to apply all the information by the name of "therm\_detection.launch." In the next code fragment, one can see in detail every parameter needed.

```
<?xml version="1.0"?>
<launch>
  <arg name="agent_id"          default="1"/>
  <!-- Debug viewer for local use and debug topic publisher enable -->
  <arg name="debug_view"       default="false"/>
  <arg name="debug_publisher"  default="false"/>
  <!-- Angle (degrees) / Note: Incrementing 1 grade increase 1 grade per side -->
  <arg name="angle_amplitude"  default="5"/>
  <!-- Camera pointing direction: FORWARD/DOWNWARD -->
  <arg name="camera_config"    default="FORWARD"/>
  <arg name="covariance_x"     default="1.5"/>
  <arg name="covariance_y"     default="1.5"/>
  <arg name="covariance_z"     default="0.5"/>
  <arg name="thermal_threshold" default="37 "/>
  <arg name="num_frame_filter" default="15"/>

  <node name="thermal" pkg="fire_detector" type="fire_detector" respawn="true" output="screen">
    <param name="angle_amplitude" value="$(arg angle_amplitude)"/>
    <param name="camera_config"   value="$(arg camera_config)"/>
    <param name="covariance_x"    value="$(arg covariance_x)"/>
    <param name="covariance_y"    value="$(arg covariance_y)"/>
    <param name="covariance_z"    value="$(arg covariance_z)"/>
    <param name="debug_view"      value="$(arg debug_view)"/>
    <param name="debug_publisher" value="$(arg debug_publisher)"/>
    <param name="thermal_threshold" value="$(arg thermal_threshold)"/>
    <param name="num_frame_filter" type="int" value="$(arg num_frame_filter)"/>
    <param name="uav_id"         type="string" value="$(arg agent_id)"/>
  </node>
</launch>
```

Figure 35: Fragment of code – Launcher of Fire Detector Node

Once the parameters have been explained, it is time to review the actual code and how it works. The overall code consists of six functions, and they will be described in general lines, providing only crucial parts of the coding. The complete code can be found in the appendix at the end of this document.

In the first place, through the main function, the `fire_detector` node will be declared, and after that, the thermal function will initialize and receive every parameter of `therm_detection` launcher. Once it is done, the node will subscribe to Terabee's thermal camera topics. When the camera capturing data, it will provide three different topics with the information obtained, these are:

1 - `/teraranger_evo_thermal/rgb_image`: a color mapped RGB image based on thermal data. The color map is relativized, whenever it detects a pixel whose temperature is higher than the other ones, this pixel will have a yellow color while the rest will have a blue color. This is completely independent of the temperature perceived.

2 - `/teraranger_evo_thermal/raw_temp_array`: an array of 1024 raw thermal data, this means the resolution will be of 32x32 pixels.

3 - `/teraranger_evo_thermal/ptat`: internal temperature of the sensor.

However, only `rgb_image` and `raw_temp_array` will be picked up to process the image and calculate where the fire is and if it surpasses the thermal threshold established.

The thermal node will also subscribe to `ual/pose` topic, so it can obtain UAV agent current position so it can then place where the fire spot is, whenever fires are detected. This will provide a more precise position of the fire object combined with Rpliddar laser measurements. Thus, the node will also subscribe to the laser scan topic, so it can read laser measurements, as it has already been explained.

When the image has been processed, the node will publish information about where the fire object is through a topic called "sensed\_objects," which has a specific type called `ObjectDetectionList`, which will be reviewed ahead. Also, a service called `thermal_detection/fire_detected` will be included to check if fire was detected or not. Additionally, a debug feature along a debug topic has been included. This will allow us to visualize through terminal the images received and the images processed.

```
sub_raw_temp_ = nh.subscribe("teraranger_evo_thermal/raw_temp_array", 1, &Thermal::thermalDataCallback, this);
sub_rgb_img_ = nh.subscribe("teraranger_evo_thermal/rgb_image", 1, &Thermal::thermalImageCallback, this);
sub_ual_pose_ = nh.subscribe("ual/pose", 1, &Thermal::ualPoseCallback, this);
sub_scan_ = nh.subscribe("scan", 1, &Thermal::laserCallback, this);

pub_sensed_ = nh.advertise<mbzirc_comm_objs::ObjectDetectionList>("sensed_objects", 1);
if(debug_publisher_) {
    pub_debug_ = nh.advertise<sensor_msgs::Temperature>("thermal_detection/debug", 1);
}

srv_checkfire_ = nh.advertiseService("thermal_detection/fire_detected", &Thermal::checkFireCallback, this);
```

Figure 36: Fragment of code - Subscribers and publisher to the Thermal Node

The next function will redimension the thermal array of 1024 points obtained from `/teraranger_evo_thermal/raw_temp_array` to a 32x32 matrix, so it is possible to create an image of the thermal pixels only. By creating this image, it will be possible to apply the thermal threshold to every point in the image, thus to determine if there is a fire or not. Also, it will obtain the pixel whose temperature is the highest. This last data is only done to debug.



```

void Thermal::thermalDataCallback(const std_msgs::Float64MultiArray::ConstPtr& msg) {
    int aux; // to loop through thermal array data
    float aux_max = 0;
    for (int j = 0, aux = 0; j < M_TEMP; j++) {
        for (int k = 0; k < M_TEMP; k++, aux++) {
            // Save current temp matrix
            temp_matrix_[k][j] = msg->data[aux];
            // Search highest temp of the array
            if (msg->data[aux] > aux_max) {
                aux_max = msg->data[aux];
            }
        }
    }
    max_temp_ = aux_max;
}

```

Figure 37: Fragment of code - Function to convert array to matrix

The next function described will be the one that obtains position data from UAL. Along with the position and orientation, the header will also be recovered to relocate it to the ObjectDetection message, in which different data belonging to the fire position will be set.

```

void Thermal::ualPoseCallback(const geometry_msgs::PoseStamped& msg) {
    // Position in x,y,z
    uav_position_.header = msg.header;
    uav_position_.point = msg.pose.position;

    tf2::Quaternion q;
    tf2::fromMsg(msg.pose.orientation, q);
    tf2::Matrix3x3 Rot_matrix(q);
    double roll, pitch, yaw;
    Rot_matrix.getRPY(roll, pitch, yaw);
    uav_yaw_ = yaw;
}

```

Figure 38: UAL position function

The next function that will be reviewed is the one that reads laser measurements and establishes the angle amplitude of the arc of the measure. In order to calculate the arc, the first thing to understand is that the complete range of the Rpliddar sweeps an arc that covers from  $-\pi/2$  to  $\pi/2$ . This scan store the measures into an array of 720 points. That means a point is equivalent to 0.004369 radians. Thus, to obtain the measures that are in front of the UAV, it is needed to place the initial end of the loop in 360.

Therefore, provided that the calculations of the laser measurements are set in an arc whose value is the one provided by the parameter "angle\_amplitude," it is imperative to calculate the equivalent grades in points in the loop, in order to calculate the actual distance between the UAV and the fire spot. To avoid infinite measures, there is a constraint to take only measures that are included between a maximum and minimum range within the laser scan.

Once the correct measures are taken, an average of the values of the measures will be applied, and the value obtained will be estimated distance between UAV agent and the fire spot.

```

//Routine to connect and obtain data from laser scanner and obtaining an average of the values
void Thermal::laserCallback(const sensor_msgs::LaserScan& msg) {
    float sum_measurement = 0.0;
    int n_measurement = 0;
    // Reading every measure established and calculating the average
    for (int i = 0; i < (angle_amplitude *2); i++, n_measurement++) {
        if ((msg.ranges[initial_index +i] > msg.range_min) && (msg.ranges[initial_index +i] < msg.range_max)) {
            sum_measurement = msg.ranges[initial_index +i] + sum_measurement;
        }
    }
    laser_measurement_ = sum_measurement / n_measurement;
}

```

Figure 39: Fragment of code - Laser measurement to calculate the estimated distance to the wall

Finally, the last function that remains is the one that will process the image and will send the message indicating whether fires are detected or not.

The first thing that has to be done is to convert the information obtained from the topic /teraranger\_evo\_thermal/rgb\_image received through the thermal camera. Truth is, this information is not really crucial to the fire detector message. However, it is quite vital to obtain it so that the programmer understands what is happening and what the camera is perceiving. The information on the topic is converted from an 8UC3 format, which is a unique format for ROS topics, to an RGB format that the OpenCV library can process adequately.

In parallel, a new template for the temperature image of 32x32, which is the one that will be sending information, is created. This auxiliary image will allow us to process the fire detection, and it works as it follows. First, all the pixels of the image are read, and then they are compared to the thermal threshold applied. Whenever the pixel temperature is higher than the threshold, that pixel will be painted in white, acquiring the value 255. Nevertheless, for those pixels that do not surpass the threshold, they will be painted in black, reaching the value 0. That disposition will create an artificial black and white image. In reality, the image has an RGB format, and that is because different painting of lines and centroids will be required in the future to observe where the fire is located in the image with precision.

```

// Convert msg from ros topic to image
cv_bridge::CvImageConstPtr cv_ptr = cv_bridge::toCvCopy(msg, "8UC3");
cv::Mat therm = Mat::zeros(Size(M_TEMP*SCALE_FACTOR, M_TEMP*SCALE_FACTOR), CV_8UC1);

// Routine to obtain a black & white filter,
// Black = there is no fire
// White = pixel temperature is bigger than thermal threeshold
for (int i = 0; i < M_TEMP*SCALE_FACTOR; i++) {
    for (int j = 0; j < M_TEMP*SCALE_FACTOR; j++) {
        if (temp_matrix [int(floor(i/SCALE_FACTOR))][int(floor(j/SCALE_FACTOR))] >= thermal_threshold) {
            *((uint8_t *) (therm.data + M_TEMP*SCALE_FACTOR * i + j)) = 255;
        }
    }
}
}

```

Figure 40: Fragment of code - Declaration of the black and white thermal image

After this is set, it is crucial to create a filter to avoid false positives and false negatives in the fire detection. Thus, if the maximum temperature detected in the image is lower than the thermal threshold established, a false negative is seen. This is done to avoid any possible bug that may appear.

Once this is done, a loop will be established to look for different groups of white pixels that are alone or together, distributed in the image. By doing this operation, contours are being detected. Thus, once they are located, the loop will calculate the center of the shape and it will paint a green rectangle around those contours. This will paint and find every outline of the fire in the image.

The approach taken to do this fire detector is to assume that there will only be one fire at the same time in the image. Thus, the next step after locating the center of the fires in every fire contour is to obtain the center that conforms to every fire at the same time. That is why it is assumed only one fire per image. Doing this, it is possible to locate the center of the real fire in the image.

Once the center of all of the fire pixels has been located, a circle of a determined radius will be painted in the image, and the real fire will have that precise location. This point center is the information that will be sent through the ObjectDetection message.

```

Point sum;
float x_comp, y_comp;
for (int d = 0; d < outline.size(); d++) {
    sum.x = cx[d] + sum.x;
    sum.y = cy[d] + sum.y;
    if (d == outline.size() - 1) {
        sum.y = sum.y / outline.size();
        sum.x = sum.x / outline.size();
        circle(image_color, sum, R_CIRCLE, CV_RGB(0,255,0), 1, 16, 0);
        circle(image_color, sum, 3, CV_RGB(0,255,0), 1, 16, 0);
        line(image_color, center, sum, CV_RGB(0,255,0), 1, 16, 0);

        x_comp = center.y - sum.y;
        y_comp = sum.x - center.x; // TODO: Check
    }
}

```

Figure 41: Fragment of code - Calculation of the center of the fire contours

Once everything is calculated, the next step is to recover the parameters that must be sent to the ObjectDetection message. Although this message is a generic one that can include different objects used in other challenges, it is important to review which fields are needed to fill in the case of the fire detector. In the next figure, it is possible to observe the complete message and its parameters.

```

geometry_msgs/Point point_of_interest
geometry_msgs/Point relative_position
float32 relative_yaw
geometry_msgs/Vector3 scale
uint8 type
uint8 color
ThermalImage image_detection
bool is_cropped

uint8 TYPE_UNKNOWN = 0
uint8 TYPE_BALLOON = 1
uint8 TYPE_BALL = 2
uint8 TYPE_BRICK = 4
uint8 TYPE_BRICK_TRACK = 5
uint8 TYPE_UCHANNEL = 6
uint8 TYPE_LWALL = 7
uint8 TYPE_FIRE = 8
uint8 TYPE_HOLE = 9
uint8 TYPE_PASSAGE = 10

int8 BRICK_COLORS = 4
uint8 COLOR_UNKNOWN = 0
uint8 COLOR_RED = 1
uint8 COLOR_GREEN = 2
uint8 COLOR_BLUE = 3
uint8 COLOR_ORANGE = 4
uint8 COLOR_WHITE = 5
uint8 COLOR_FIRE = 6

```

Figure 42: Fragment of code - Object Detection message

From the fields that appeared in the previous image, the ones that need to be filled are the header, which comes from the UAL position obtained in the function that was explained before. Object type and object color will let other nodes know that the sensor has spotted a fire with a new color. Color field comes from Challenge 2, so in reality, this one must be filled with another color type. That is the reason why this field is color unknown.

Also, covariances have been added to the position. The value provided is a sigma that is included in therm\_detection.launch file. These values will add the standard deviation in which a UAV is expected to move.

Field image\_detection is referred to different parameters of the temperature image. It is included img\_height and img\_width, which is referred to as the resolution of the camera. U and v are designated to x,y coordinates of the

center of the fire object in the image. And lastly, height and width is the radius of the circle painted in the image. This is also recovered to determine where the fire is and not only the center point of the image.

After all these parameters are recovered and set into ObjectDetection message, the next one is key to situate the fire in the arena space. First of all, it is imperative to identify where the camera is pointing to. Depending on the mode, if it is downward or forward, positions and depth of the fire spot may vary.

If the camera is pointing downward, the camera depth, which is referred to where the fire is projected in front of the camera will receive the current height of the UAV and the position of the fire will be then, the same position of the UAV but with its height at zero.

However, if the camera is pointing forward, camera depth will be Rpliddar laser measurement that was obtained in a previous function, explained before. The real position of the fire will be, then, for the x-axis, UAV's position plus the laser measurement multiplied by cosine of UAV yaw. For y-axis will be UAV's position plus laser measurement multiplied by sine of UAV yaw. And finally, the z-axis will be the same as UAV's position.

Once this is set, the next step is to push the message so other nodes can read it. Nevertheless, the message will not be sent just as an independent message. Instead, an ObjectDetectionList will be the message sent. So, to complete its fields, stamp, agent\_id, and objects must be filled with the necessary information.

```
// Publishing the Object
mbzirc_comm_objs::ObjectDetectionList rec_list;
rec_list.objects.push_back(rec_object);
rec_list.stamp = ros::Time::now();
rec_list.agent_id = uav_id_;
pub_sensed_.publish(rec_list);
rec_list.objects.clear();
```

Figure 43: Fragment of code - How to publish the ObjectDetection List

Finally, to finish up the code, the last thing needed to do is to convert the temperature image to a format compatible with rostopic image format. After that is done, the fire detector is finished.

### 5.3.1 Thermal Detection Node Experiments

The best way to comprehend how and what the code is trying to do is to show the results obtained through different experiments, along with an explanation of what is happening.

These experimental results are obtained from a mock-up scenario made before the real competition took place. In here, all the experimental tests were arranged and done.

The first experiment that will be performed is a UAV agent getting close to a fire spot, determining if there is the fire in the wall that is approaching and then shooting water to the gap to fill the canister that provides a score in the competition.

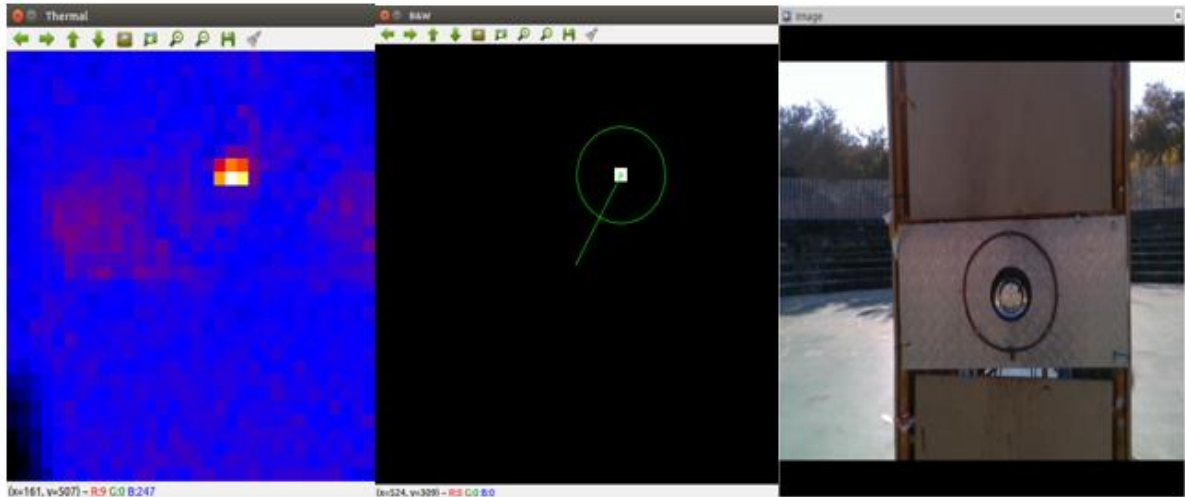


Figure 44: Fire spot detected

As it can be seen in the previous figure, when UAV is getting close to the fire spot, the thermal camera will spot a heat zone in the image. However, if the temperature does not surpass the value of the threshold, it will not be a candidate to be a fire. In this case, it does reach the fire threshold. Thus fire is detected in the black and white image. The right image is what the UAV agent is watching.

Once the fire has been spotted, UAV will position itself so it can shoot water to the gap. When that happens, the fire detected will be extinguished.

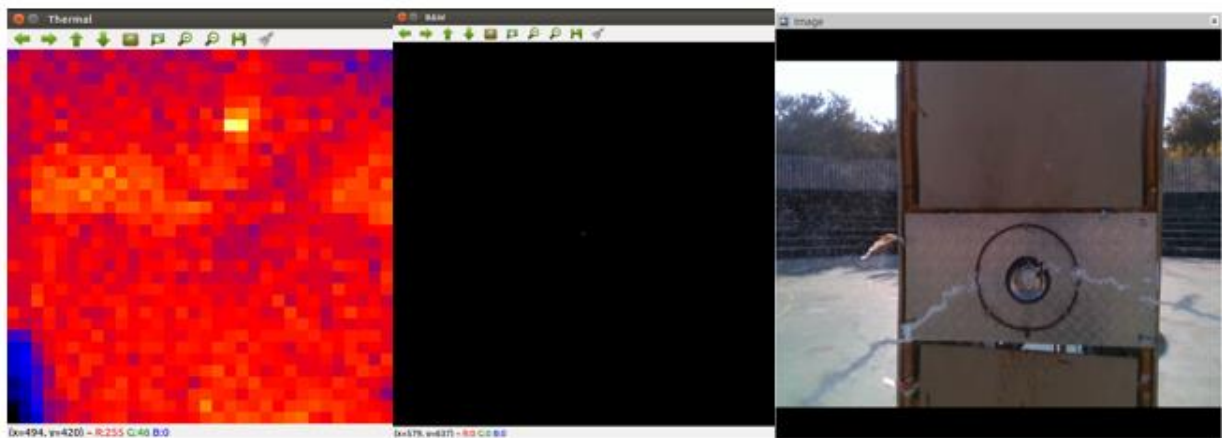


Figure 45: Fire Extinguished

As it can be seen in the previous image, the thermal image will show that the fire spot is now refrigerated. In the black and white image, fire is no longer visible while in the right window it is possible to see how UAV is shooting water inside the gap, to the canister

The next experiment consists of the UAV approaching a wall but with the fire ring active. In this case, the thermal image will provide a more prominent spot which includes all the circle of fire. This is when painting a radius inside the center of the spotted fire makes sense.

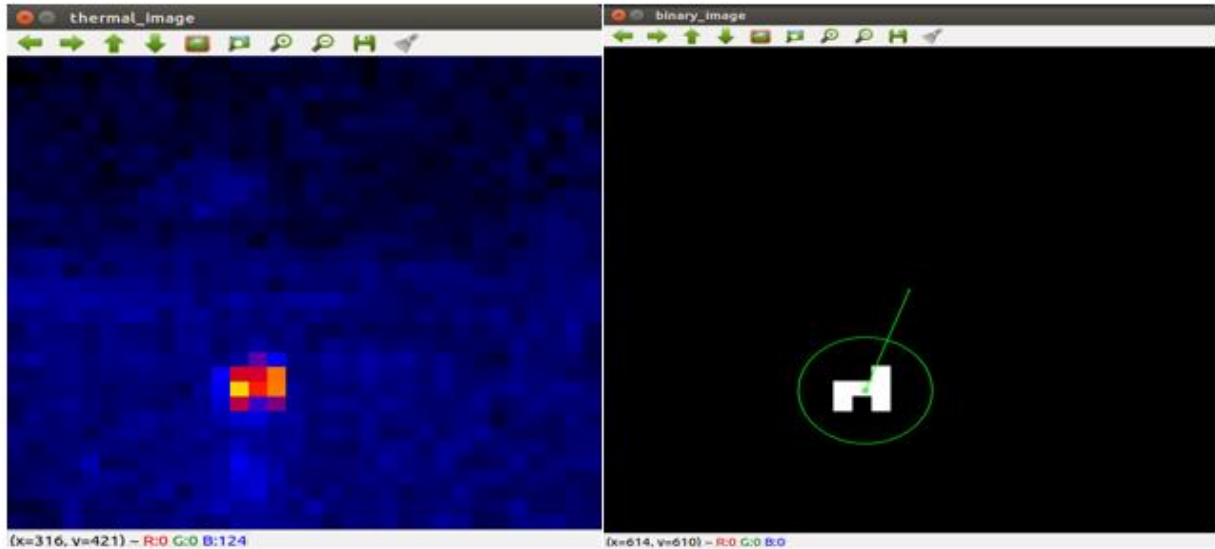


Figure 46: Thermal Image Detection

In this case, only the thermal and temperature images are active. The left window shows where the fire spot is probably to be, whereas the right window will mark the exact center of the fire within a circle. When this happens, ObjectDetection will display the information provided by the fire detector, which can be consulted in the next terminal display.

```

javier@Javidomjim: ~
---
agent_id: "1"
stamp:
  secs: 1583835819
  nsecs: 597444610
objects:
-
  header:
    seq: 0
    stamp:
      secs: 1583835819
      nsecs: 597397443
    frame_id: "odom"
  pose:
    pose:
      position:
        x: -3.72621011734
        y: 8.08833503723
        z: 3.42185616493
      type: 8
      color: 0
      image_detection:
        camera_direction: 1
        img_height: 32
        img_width: 32
        u: -2.15000009537
        v: 6.15000009537
        height: 3.5
        width: 3.5
        depth: 0.0
        is_cropped: False
  ---

```

Figure 47: Terminal display of the Fire Detector

As can be seen, the UAV agent operating has agent id 1. In the message, it is included the position of the fire object, along with the camera direction, which in this case, 1 means forward. The center of the fire is spotted in the pixels -2.15, 6.15.

The last thing is implementing this code in the complete system, and with that, this module will be ready for the competition.

## 5.4 Circle Detection Implementation

In this section, a review of the circle detection node will be performed. To explain how the implementation has been done, every function will be described in detail, just like it was done with the fire detector node.

First, to declare every variable, function, and information needed, a class by the name of CircleDetector was created. This one also includes every publisher and subscriber needed. For example, to obtain data from Realsense camera, it is necessary to subscribe to two topics that the sensor provides, which are:

1 - `"/camera/aligned_depth_to_color/image_raw,"` which provides the raw image that the camera is providing, just like a normal camera would,

2- `"/camera/color/image_rect_color"` which provides the information about depth obtained through the camera.

When all of the variables needed are declared, the next step is to read the parameters from the class created before.

```
// Read parameters
pnh_.param<std::string>("tf_prefix", tf_prefix_, "mbzirc2020_1");
pnh_.param<bool>("publish_debug_marker", publish_debug_marker_, false);
pnh_.param<bool>("publish_debug_images", publish_debug_images_, false);
pnh_.param<double>("dp", dp_, 1);
pnh_.param<int>("min_dist_between_circle_center", min_dist_between_circle_center_, 480/8);
pnh_.param<double>("canny_edge_upper_threshold", canny_edge_upper_threshold_, 150);
pnh_.param<int>("accumulator_threshold", accu_th_, 50);
pnh_.param<int>("min_radius", min_radius_pixels_, 0);
pnh_.param<int>("max_radius", max_radius_pixels_, 0);
pnh_.param<float>("min_radius_meters", min_radius_meters_, 0.05);
pnh_.param<float>("max_radius_meters", max_radius_meters_, 0.6);
pnh_.param<float>("max_depth_th", max_depth_th_, 10.0);
pnh_.param<bool>("use_gaussian_blur", use_gaussian_blur_, false);
pnh_.param<float>("gaussian_blur_sigma", gaussian_blur_sigma_, 2);
pnh_.param<int>("gaussian_blur_kernel_size", gaussian_blur_kernel_size_, 3);
pnh_.param<bool>("simulation", simulation_, false);

std::string camera_info_topic = "camera/aligned_depth_to_color/camera_info";
camera_info_sub_1_ = nh_.subscribe(camera_info_topic, 1, &CircleDetector::callbackCameraInfo, this);
camera_info_sub_2_ = nh_.subscribe("camera/color/camera_info", 1, &CircleDetector::callbackCameraInfo, this);
```

Figure 48: Fragment of code - Reading the parameters from Circle Detector class

After the declaration has been established, just like the Fire Detector node, the possibility of activating a debug feature has been included. In this case, the activating debug will allow the programmer to visualize the images and markers in those images. With debugging features deactivated, the program will run much faster, and its performance will be relatively better.

The first function in this node will unsubscribe from the topic that receives the information from the camera, because this one is always the same, and it is not necessary to maintain those resources active in the system. Once this is done, this function will not be called until the next restart of the node.

```
void CircleDetector::callbackCameraInfo(const sensor_msgs::CameraInfoConstPtr &camera_info_ptr)
{
    // Unsubscribe upon reception of first camera info message, because is supposed to be always the same
    camera_info_sub_1_.shutdown();
    camera_info_sub_2_.shutdown();
    ROS_INFO("Received camera_info");
    model_.fromCameraInfo(camera_info_ptr);
    has_camera_info = true;
}
```

Figure 49: Fragment of code - Callback camera information

The next function is `callbackSyncColorDepth`, which is crucial in the code. This one will check out whether the frame received in the camera matches or not with the one expected. If that is the case, the program will continue, and it will launch the next function, which is the one that will process the circle detection in the image.

Also, in this function, it has been added an alternative in which it is possible to launch a specific function if the node is started in a simulation environment or not. For the sake of clarity, only the function that works in the real scenario will be explained. Nevertheless, the entire code can be found in the appendix at the end of this document.

```

void CircleDetector::callbackSyncColorDepth(const sensor_msgs::ImageConstPtr &color_img_ptr, const sensor_msgs::ImageConstPtr &depth_img_ptr)
{
    if (this->hasCameraInfo())
    {
        if ( color_img_ptr->header.frame_id != (tf_prefix_ + "/camera_color_optical_frame") )
        {
            ROS_WARN("Received camera frame doesn't match expected one");
            return;
        }
        if (!simulation_)
            findCirclesColor(color_img_ptr, depth_img_ptr);
        else
            findCirclesColorSim(color_img_ptr, depth_img_ptr);
    }
    else
    {
        ROS_WARN("No camera_info available");
    }
}

```

Figure 50: Fragment of code - Callback Sync Color Depth

The next two functions are called `get3DPointCameraModel`, whose aim is to obtain a better estimation of the object position, just like in the Fire Detection node. The second one is called `getRadiusINMetersFromPixels`, which, just like its own name implies, will allow us to determine the radius of the circle in meters using pixels.

```

void CircleDetector::get3DPointCameraModel(geometry_msgs::Point &point, float &depth, int &pixel_row, int &pixel_col)
{
    point.x = (pixel_col - model_.cx())/model_.fx() * depth;
    point.y = (pixel_row - model_.cy())/model_.fy() * depth;
    point.z = depth;
}

float CircleDetector::getRadiusInMetersFromPixels(float &depth, int &radius_pixel)
{
    return radius_pixel/( (model_.fx() + model_.fy())/2 ) * depth;
}

```

Figure 51: Fragment of code – Functions

Last, the function called `findCirclesColor` is the one that will process the image and determine if the UAV agent has detected a circle or not through the Realsense camera. A more detailed review of this part of the code will be done in the following.

First, after camera topics have been read, just like it was done in the fire detector node, the images received must be converted from the message to a format that OpenCV libraries can process. After that is done, it is possible to apply a gaussian blur into the image to reduce potential noise detected.

The next step is to look for fire circles in the image. For this purpose, Hough detection algorithm for circles will be used over the color image received from the camera. Still, first, a mask is created to filter the grayscale image based on pixels values from depth images. If these pixels have depth values that are higher than a determined threshold, or higher than the maximum range of the camera or 0, the grayscale image will receive a zero value in that concrete pixel. Otherwise, pixels will receive value 255. By doing this, a mask with maximum and minimum values will be obtained.

```

int i1 = 0;
int j1 = 0;
for(i1 = 0; i1 < depth_image_cv->image.rows; i1++)
{
    for(j1 = 0; j1 < depth_image_cv->image.cols; j1++)
    {
        const cv::Mat & original_image_ref_mm = depth_image_cv->image;
        const cv::Mat & mask_image_ref = mask_image_cv.image;
        uint16_t * value_original_mm_ptr = (uint16_t *) (original_image_ref_mm.data + original_image_ref_mm.step[0]*i1 + original_image_ref_mm.step[1]*j1);
        uint8_t * mask_image_pixel_ptr = (uint8_t *) (mask_image_ref.data + mask_image_ref.step[0] * i1 + mask_image_ref.step[1] * j1);
        if ( (*value_original_mm_ptr == 0) || (*value_original_mm_ptr/1000.0 > 10) || (*value_original_mm_ptr/1000.0 > max_depth_th_ ) )
        {
            *mask_image_pixel_ptr = 0;
        }
        else
        {
            *mask_image_pixel_ptr = 255;
        }
    }
}

```

Figure 52: Fragment of code - Masking Grayscale Image



After that is done, erosion and dilation will be applied to the mask image obtained in the previous step, to apply a mask to the grayscale image later. The next step is applying the HoughCircles algorithm, which includes the Canny Edge detector as one parameter of the function.

Once the circles are detected through the function, it will return a vector, which includes x,y of the center point and the radius of the circle. Later, it will be checked whether the radius of that concrete circle matches with the measures taken from the real circle, which had been measured previously.

```

cv::HoughCircles(gaussian_blur_image_cv.image, circles_detected, CV_HOUGH_GRADIENT, dp, min_dist_between_circle_center,
                canny_edge_upper_threshold, accu_th, min_radius_pixels, max_radius_pixels);
//int valid_circles = 0;
for (size_t i = 0; i < circles_detected.size(); i++)
{
    cv::Point center(cvRound(circles_detected[i][0]), cvRound(circles_detected[i][1]));
    int radius = cvRound(circles_detected[i][2]);
    //std::cout << "Circle " << i << " radius : " << radius << std::endl;
    // Extract circle center depth from depth image
    float circle_center_depth = (float)((depth_image_cv->image.at<uint16_t>(center))/1000.0);
    //std::cout << "Circle center depth: " << circle_center_depth << std::endl;

    if (circle_center_depth != 0 && circle_center_depth < 10.0 && circle_center_depth < max_depth_th)
    {
        float circle_radius_meters = getRadiusInMetersFromPixels(circle_center_depth, radius);
        if ( (circle_radius_meters >= min_radius_meters) && (circle_radius_meters <= max_radius_meters) )
        {
            //valid_circles++;
            // circle center
            cv::circle( color_image_cv->image, center, 3, cv::Scalar(0,255,0), -1, 8, 0);
            // circle outline
            cv::circle( color_image_cv->image, center, radius, cv::Scalar(0,0,255), 3, 8, 0);
            //std::cout << "Circle radius meters: " << circle_radius_meters << "\n";
            //std::cout << "Circle center depth: " << circle_center_depth << std::endl;
            // Calc 3D position of circle center
            geometry_msgs::Point circle_center_3D;
            get3DPointCameraModel(circle_center_3D, circle_center_depth, center.y, center.x);
        }
    }
}

```

Figure 53: Fragment of code - HoughCircle Detection and later comprobations

Last, ObjectDetection message will be assembled. For this purpose, the header from RealSense will be collected, and the position of the circle obtained will be the center of the circle plus depth. Once that is set, the message will be added to ObjectDetectionList, just like the case of the Fire Detection node. With this, the code is explained.

### 5.4.1 Circle Detection Experimental Results

Again, to understand better what is really happening, a few experiments will be show. This time, the experiments are carried out in the real competiton that took place in Abu Dhabi. As it is possible to see in the next figures, the UAV agent is approaching to a wall with a fire that is not active. As it was explained in the previous chapter, the fire spot will have two circles around it.



Figure 54: Circle Detection in real

However, the algorithm will be able to spot the three different circles. Knowing that the one that is placed in the middle is the fire ring, with the help of the Fire Detector node, it will be an easy task to accomplish.



Figure 55: Circle Detection on Fire Ring

In the previous image, it is possible to see the fire activity in the wall. In this case, the circle detector is also able to determine if there is a circle shape in the image.

### 5.5 Fire detector and Circle Detector Working Together

Finally, here it will be shown how both algorithms worked together in the experiments run. These experiments, again, were done in the trials in the actual competition in Abu Dhabi, so in every image, it is possible to see how the conditions were.

In the next image, it is possible to see that the ring of fire is active. Fire Detector is determining where the fire is placed while Circle Detector is finding the three circles that appear in the image.

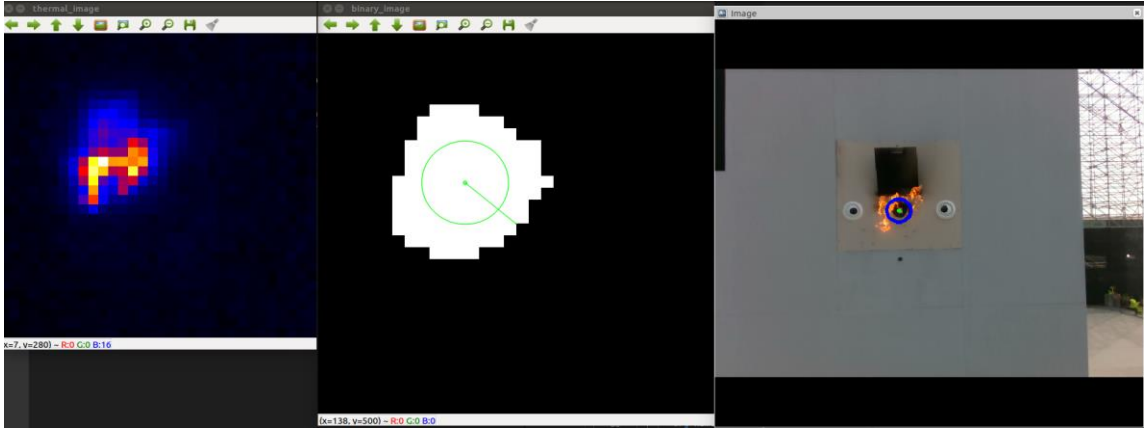


Figure 56: Both algorithms working together

It is possible to see the precision that these algorithms provide together. In the next figure, tests have been done over a circle whose ring of fire is not active, but its inner resistance still is.

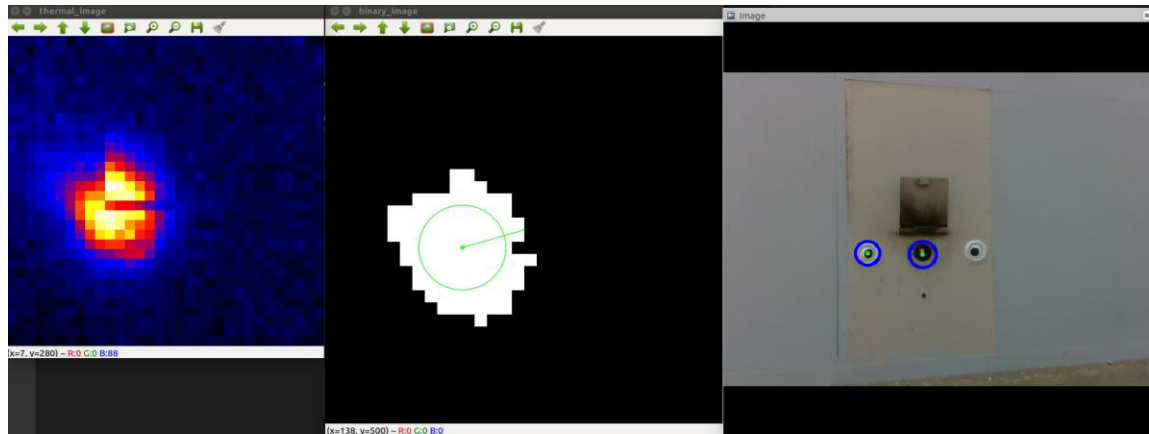


Figure 57: Fire detection when the fire ring is turned down

As it can be seen, results are satisfactory. The fire detector determines that fire has been detected in this section of the wall, while the circle detector provides information about the possible sources of heat.

With the combination of both algorithms, detection is quite accurate when the time to position the UAV agent comes. Once the position has been established, a signal that launches the water launcher will be activated, and the canisters will be filled.

In the real Challenge, once the fire had been put out, a reset was made in order to refill the container carried by the agent. After that had been completed, the next step was to launch another mission that looked for another fire spot in the building and repeated the process until time completion.



# 6 FINAL CONCLUSIONS

Although achieving the aim proposed in MBZIRC's Challenge 3 is a difficult task, it is not impossible. Combining the power of the ROS platform, Gazebo simulation, Rviz and SMACH along with sensors that provide information from the arena, it has been made possible to put out fires with UAV agents in an effective way. With a consistent high hierarchy in the system, it was possible to command and manage every node in the system, including the ones that were not explained in detail in this document. Using the central unit was possible to accomplish this.

In this Challenge, only the outdoor fires were visited and put out. Unfortunately, it has not been possible to enter the building and try to put the remaining fires out. However, with the provided information using Hector SLAM explained before, it will set a good start point to investigate and apply detection algorithms together. It can lead to a future line of work.

As it was discussed through the thesis, MBZIRC's Challenge 3 architecture was designed to be generic and multipurpose. It is quite important to mention that second Challenge of the competition is designed with the same basic architecture, but with its own particularities. In essence, this multi-robot system is applied in practice in the same competition, and as it can be seen, it does work.

Finally, this document would not be finished correctly without showing the score and the leaderboards our team formed by IST Lisboa, CATEC and University of Seville obtained.

Teams	Autonomous	Manual	Time left	Rank
University of Seville, Tecnico Lisboa (IST Lisbon), CATEC	12.2625	0	0	1
Technical University of Denmark	10	0	0	2
University of New South Wales Sydney	10	0	0	2
Czech Technical University in Prague, University of Pennsylvania, NYU	7	0	0	4
Korea Advanced Institute of Science and Technology (KAIST)	7	0	0	4
University of Tokyo	5	0	120	6
University of Zagreb Faculty of Electrical Engineering and Computing	5	0	0	7
Polytechnic University of Madrid, University Pablo Olvide, Poznan University of Technology	5	0	0	7
Virginia Tech	4.5	13.64	0	9
Lodz University of Technology Institute of Automatic Control	0	14.498	0	10
University of Leeds	0	13.952	0	11
University of Applied Sciences Aachen/MASKOR	0	10.84	0	12

Figure 58: MBZIRC's Challenge 3 Score

As it can be seen in the previous image, our team won Challenge 3 of MBZIRC's competition, and we were awarded with the first prize in this category.



# ANNEX: FIRE DETECTOR CODE

---

```
#include "ros/ros.h"
#include <fire_detector.h>
#include <vector>
#include <math.h>
#include <opencv2/opencv.hpp>
#include <std_msgs/String.h>
#include <std_msgs/Float64MultiArray.h>
#include <sensor_msgs/Image.h>
#include <sensor_msgs/LaserScan.h>
#include <sensor_msgs/image_encodings.h>
#include <sensor_msgs/Temperature.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/PointStamped.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>

#include <MBZIRC_comm_objs/ObjectDetection.h>
#include <MBZIRC_comm_objs/ObjectDetectionList.h>
#include <MBZIRC_comm_objs/CheckFire.h>

using namespace std;
using namespace cv;

Thermal::Thermal () {

    ros::NodeHandle n("~");
    n.getParam("uav_id", uav_id_);
    n.getParam("camera_config", mode_);
    n.getParam("thermal_threshold", thermal_threshold_);
    n.getParam("covariance_x", sigma_[0]);
    n.getParam("covariance_y", sigma_[1]);
    n.getParam("covariance_z", sigma_[2]);
    n.getParam("debug_publisher", debug_publisher_);
    n.getParam("debug_view", debug_view_);
    n.getParam("angle_amplitude", angle_amplitude_);
    n.param("num_frame_filter", num_frame_filter_, 15);-

    angle_amplitude_ = angle_amplitude_*CONV2PNT;
    initial_index_ = LASER_RANGE/2-angle_amplitude_; // Point to laser front
    false_negative_ = 0;
    false_positive_ = 0;
    detected_ = false;
```

```

ros::NodeHandle nh;

sub_raw_temp_ = nh.subscribe("teraranger_evo_thermal/raw_temp_array", 1,
&Thermal::thermalDataCallback, this);

sub_rgb_img_ = nh.subscribe("teraranger_evo_thermal/rgb_image", 1, &Thermal::thermalImageCallback,
this);

sub_ual_pose_ = nh.subscribe("ual/pose", 1, &Thermal::ualPoseCallback, this);
sub_scan_ = nh.subscribe("scan", 1, &Thermal::laserCallback, this);

pub_sensed_ = nh.advertise<MBZIRC_comm_objs::ObjectDetectionList>("sensed_objects", 1);
if(debug_publisher_)
{
pub_debug_ = nh.advertise<sensor_msgs::Temperature>("thermal_detection/debug", 1);
}

srv_checkfire_ = nh.advertiseService("thermal_detection/fire_detected",
&Thermal::checkFireCallback, this);
}

// Routine to find fire in the image - Just one fire in the image
void Thermal::thermalDataCallback(const std_msgs::Float64MultiArray::ConstPtr& msg) {
int aux; // to loop through thermal array data
float aux_max = 0;
for (int j = 0, aux = 0; j < M_TEMP; j++) {
for (int k = 0; k < M_TEMP; k++, aux++) {
// Save current temp matrix
temp_matrix_[k][j] = msg->data[aux];
// Search highest temp of the array
if (msg->data[aux] > aux_max) {
aux_max = msg->data[aux];
}
}
}
}
max_temp_ = aux_max;
}

// Routine to obtain data pose and header to create fire messages
void Thermal::ualPoseCallback(const geometry_msgs::PoseStamped& msg) {
// Position in x,y,z
uav_position_.header = msg.header;
uav_position_.point = msg.pose.position;

tf2::Quaternion q;
tf2::fromMsg(msg.pose.orientation, q);
tf2::Matrix3x3 Rot_matrix(q);
double roll, pitch, yaw;

```



```

    Rot_matrix.getRPY(roll, pitch, yaw);
    uav_yaw_ = yaw;
}

//Routine to connect and obtain data from laser scanner and obtaining an average of the values
void Thermal::laserCallback(const sensor_msgs::LaserScan& msg) {
    float sum_measurement = 0.0;
    int n_measurement = 0;
    // Reading every measure established and calculating the average
    for (int i = 0; i < (angle_amplitude_*2); i++, n_measurement++) {
        if ((msg.ranges[initial_index+i] > msg.range_min) && (msg.ranges[initial_index+i] <
msg.range_max)) {
            sum_measurement = msg.ranges[initial_index+i] + sum_measurement;
        }
    }
    laser_measurement_ = sum_measurement / n_measurement;
}

// Routine to process the image and determine if there is fire and where
void Thermal::thermalImageCallback(const sensor_msgs::ImageConstPtr& msg) {

// Main routine
try {
    // Convert msg from ros topic to image
    cv_bridge::CvImageConstPtr cv_ptr = cv_bridge::toCvCopy(msg, "8UC3");
    cv::Mat therm = Mat::zeros(Size(M_TEMP*SCALE_FACTOR, M_TEMP*SCALE_FACTOR), CV_8UC1);

    // Routine to obtain a black & white filter,
    // Black = there is no fire
    // White = pixel temperature is bigger than thermal threshold
    for (int i = 0; i < M_TEMP*SCALE_FACTOR; i++) {
        for (int j = 0; j < M_TEMP*SCALE_FACTOR; j++) {
            +if      (temp_matrix_[int(floor(i/SCALE_FACTOR))][int(floor(j/SCALE_FACTOR))])
thermal_threshold_) {
                *((uint8_t *) (therm.data + M_TEMP*SCALE_FACTOR * i + j)) = 255;
            }
        }
    }

    Point center;
    center.x = M_TEMP*SCALE_FACTOR/2.0;
    center.y = M_TEMP*SCALE_FACTOR/2.0;

    vector<vector<Point>> outline;
    findContours(therm, outline, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE);
    vector<Moments> mu(outline.size());
}

```

```

vector<Point2f> mc(outline.size());

cv::Mat image_color;
cvtColor(therm, image_color, CV_GRAY2BGR);
circle(image_color, center, 1, CV_RGB(0,255,0), 1, 25, 0);

if ((false_positive_ > num_frame_filter_) && ((max_temp_ >= thermal_threshold_) ||
(false_negative_ <= MAX_FILTER_NEGATIVES)) && (laser_measurement_ < 7.5 || mode_=="DOWNWARD")) {
    detected_ = true;

    if (max_temp_ <= thermal_threshold_) {
        false_negative_++;
    } else {
        false_negative_ = 0;
    }

    float cx[SCALE_FACTOR], cy[SCALE_FACTOR];
    // Calculating moments and centers in the fire
    for (int d = 0; d < outline.size(); d++) {
        // Obtaining centers in the fire
        mu[d] = moments(outline[d], false);
        mc[d] = Point2f(mu[d].m10/mu[d].m00, mu[d].m01/mu[d].m00);
        cx[d] = mu[d].m10/mu[d].m00;
        cy[d] = mu[d].m01/mu[d].m00;
        // Painting a rectangle in the fire
        Rect rect = boundingRect(outline[d]);
        circle(image_color, mc[d], 2, CV_RGB(0,255,0), 1, 16, 0);
    }

    Point sum;
    float x_comp, y_comp;
    for (int d = 0; d < outline.size(); d++) {
        sum.x = cx[d] + sum.x;
        sum.y = cy[d] + sum.y;
        if (d == outline.size() - 1) {
            sum.y = sum.y / outline.size();
            sum.x = sum.x / outline.size();
            circle(image_color, sum, R_CIRCLE, CV_RGB(0,255,0), 1, 16, 0);
            circle(image_color, sum, 3, CV_RGB(0,255,0), 1, 16, 0);
            line(image_color, center, sum, CV_RGB(0,255,0), 1, 16, 0);

            x_comp = center.y - sum.y;
            y_comp = sum.x - center.x; // TODO: Check
        }
    }
}

```

```

MBZIRC_comm_objs::ObjectDetection rec_object;
rec_object.header.stamp = ros::Time::now();
rec_object.header.frame_id = uav_position_.header.frame_id;
rec_object.type = MBZIRC_comm_objs::ObjectDetection::TYPE_FIRE;
rec_object.color = MBZIRC_comm_objs::ObjectDetection::COLOR_UNKNOWN;
rec_object.pose.covariance = { sigma_[0]*sigma_[0], 0, 0, 0, 0, 0,
                                0, sigma_[1]*sigma_[1], 0, 0, 0, 0,
                                0, 0, sigma_[2]*sigma_[2], 0, 0, 0,
                                0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0};

rec_object.image_detection.img_height = M_TEMP;
rec_object.image_detection.img_width = M_TEMP;
rec_object.image_detection.v = y_comp/SCALE_FACTOR;
rec_object.image_detection.u = x_comp/SCALE_FACTOR;
rec_object.image_detection.height = R_CIRCLE/SCALE_FACTOR;
rec_object.image_detection.width = R_CIRCLE/SCALE_FACTOR;

if (mode_ == "DOWNWARD") {
    //Thermal image fields
    rec_object.image_detection.depth = uav_position_.point.z;
    rec_object.image_detection.camera_direction =
MBZIRC_comm_objs::ThermalImage::CAMERA_DIRECTION_DOWNWARD;
    // Object Detection fields
    rec_object.pose.pose.position.x = uav_position_.point.x;
    rec_object.pose.pose.position.y = uav_position_.point.y;
    rec_object.pose.pose.position.z = 0.0;
} else if (mode_=="FORWARD") {
    //Thermal image fields
    rec_object.image_detection.depth = laser_measurement_;
    rec_object.image_detection.camera_direction =
MBZIRC_comm_objs::ThermalImage::CAMERA_DIRECTION_FORWARD;
    //Object detection fields
    rec_object.pose.pose.position.x = uav_position_.point.x +
laser_measurement_*cos(uav_yaw_);
    rec_object.pose.pose.position.y = uav_position_.point.y +
laser_measurement_*sin(uav_yaw_);
    rec_object.pose.pose.position.z = uav_position_.point.z;
}

if (debug_view_) {
    if (false_negative_) {
        cout << "Possible false negative: Max temp detected-> " << max_temp_ << " | Current
threshold-> " << thermal_threshold_ << endl;
    } else {
        // Fire detected
        cout << "Fire_detected" << endl;
    }
}

```

```

        cout << "Distance to center ~ x: " << to_string(x_comp) << " y:" << to_string(y_comp)
<< " total:" << to_string(sqrt(x_comp*x_comp+y_comp*y_comp)) << endl;
    }
}

// Publishing the Object
MBZIRC_comm_objs::ObjectDetectionList rec_list;
rec_list.objects.push_back(rec_object); // TODO - Check if publish full list or current
point

rec_list.stamp = ros::Time::now();
rec_list.agent_id = uav_id_;
pub_sensed_.publish(rec_list);
rec_list.objects.clear();
} else {

    detected_ = false;
    if ((max_temp_ >= thermal_threshold_) && (laser_measurement_ < 5.0 || mode_=="DOWNWARD")) {
        false_positive_++;
        if (debug_view_) {
            cout << "POSSIBLE FIRE DETECTED: Max temp detected-> " << max_temp_ << " | Current
threshold-> " << thermal_threshold_ << endl;
        }
    } else {
        false_positive_ = 0;
        if (debug_view_) {
            cout << "FIRE NOT DETECTED: Max temp detected-> " << max_temp_ << " | Current
threshold-> " << thermal_threshold_ << endl;
        }
    }
}

flip(image_color, image_color, -1);
rotate(image_color, image_color, ROTATE_90_COUNTERCLOCKWISE);

if (debug_view_) {
    // Displaying images on screen
    string thermal_image_window = "thermal_image";
    namedWindow(thermal_image_window);
    flip(cv_ptr->image, cv_ptr->image, 1);
    imshow(thermal_image_window, cv_ptr->image);

    string binary_image_window = "binary_image";
    namedWindow(binary_image_window);
    moveWindow(binary_image_window, 565, 0);
    imshow(binary_image_window, image_color);

    waitKey(3);
}

```

```

// Converting image to msg
std_msgs::Header header = msg->header;
cv_bridge::CvImage img_bridge = cv_bridge::CvImage(header, sensor_msgs::image_encodings::RGB8,
image_color);

if (debug_publisher_) {
    sensor_msgs::Temperature measure_debug;
    measure_debug.header = header;
    measure_debug.temperature = max_temp_;
    measure_debug.variance = thermal_threshold_;
    pub_debug_.publish(measure_debug);
}

} catch (cv_bridge::Exception& e) {
    ROS_ERROR("cv_bridge exception: %s", e.what());
    return;
}
}

bool Thermal::checkFireCallback(MBZIRC_comm_objs::CheckFire::Request &req,
MBZIRC_comm_objs::CheckFire::Response &res)
{
    res.fire_detected = detected_;
    return true;
}

int main(int argc, char** argv) {
    ros::init(argc, argv, "fire_detector");
    Thermal thermal;
    ros::spin();

    return 0;
}

```

The complete code can be found in this repository in GitHub: <https://github.com/grvcTeam/mbzirc2020>

# ANNEX: HECTOR SLAM

---

```
<?xml version="1.0"?>
<launch>
  <arg name="tf_map_scanmatch_transform_frame_name" default="scanmatcher_frame"/>
  <arg name="base_frame" default="base_footprint"/>
  <arg name="odom_frame" default="nav"/>
  <arg name="pub_map_odom_transform" default="false"/>
  <arg name="scan_subscriber_queue_size" default="5"/>
  <arg name="scan_topic" default="scan"/>
  <arg name="map_size" default="2048"/>
  <node pkg="hector_mapping" type="hector_mapping" name="hector_mapping" output="screen">

    <!-- Frame names -->
    <param name="map_frame" value="map" />
    <param name="base_frame" value="$(arg base_frame)" />
    <param name="odom_frame" value="$(arg odom_frame)" />

    <!-- Tf use -->
    <param name="use_tf_scan_transformation" value="true"/>
    <param name="use_tf_pose_start_estimate" value="true"/>
    <param name="pub_map_odom_transform" value="$(arg pub_map_odom_transform)"/>

    <!-- Map size / start point -->
    <param name="map_resolution" value="0.050"/>
    <param name="map_size" value="$(arg map_size)"/>
    <param name="map_start_x" value="0.5"/>
    <param name="map_start_y" value="0.5" />
    <param name="map_multi_res_levels" value="2" />

    <!-- Map update parameters -->
    <param name="update_factor_free" value="0.4"/>
    <param name="update_factor_occupied" value="0.9" />
    <param name="map_update_distance_thresh" value="0.4"/>
    <param name="map_update_angle_thresh" value="0.06" />
    <param name="laser_z_min_value" value = "-1.0" />
    <param name="laser_z_max_value" value = "1.0" />

    <!-- Advertising config -->
    <param name="advertise_map_service" value="true"/>
    <param name="scan_subscriber_queue_size" value="$(arg scan_subscriber_queue_size)"/>
    <param name="scan_topic" value="$(arg scan_topic)"/>
    <param
      name="tf_map_scanmatch_transform_frame_name"
      value="$(arg
tf_map_scanmatch_transform_frame_name)" />

  </node>
</launch>
```

