Professional Bachelor Applied Information Science

Bavo Knaeps

Promoters:

Tim Dupont   Center of expertise PXL Smart ICT

Professional Bachelor Applied Information Science

Bavo Knaeps

Promoters:

Tim Dupont   Center of expertise PXL Smart ICT

Acknowledgements

Abstract

A single drone is not able to accomplish much on a large scale. Compare this to a bee trying to harvest a field of flowers, an impossible task for just a single bee. Nature solves this problem using swarm intelligence, using a colony of bees working together to solve a bigger problem. This phenomenon is not only perceived in bees. Birds flocking, hawks hunting, animal herding, bacterial growth, fish schooling and microbial intelligence are all examples of swarm intelligence in nature. Multi-agent UAV systems are based on swarm intelligence, using multiple UAVs (aka drones) to solve more complex and bigger problems that are impossible for just a single UAV to solve.

This research is conducted by PXL Smart ICT, the IT & Electronics center of expertise and part of the research department at PXL University of Applied Sciences and Arts. The goal of the project is to enable IT companies to implement UAV projects via rapid robot prototyping.

The internship provides a multi-agent system (MAS) architecture that supports homogeneous as well as heterogeneous structures. Each UAV is an agent that processes data on its own and shares its knowledge with the other agents. The agent organization can handle hierarchical, hologenic and coalition teams. Different types of organizations will be compared regarding task completion time, computational time and accuracy of the completed task. A blackboard system takes care of knowledge sharing between agents, with the option of working over multiple computers.

ROS (Robotic Operating System) is used to control the UAVs, each equipped with a PX4 autopilot. ROS is a set of software libraries and tools that help to build smart robot applications. The PX4 autopilot uses MAVLink, a very lightweight messaging protocol for communicating with UAVs. MAVROS, an implementation of MAVLink in ROS, is a component that can convert between ROS messages and MAVLink messages. This provides easy control of the UAVs. By gathering information from the UAV components the agents can analyze data. For example, the agent receives images from the camera and can detect if an object is a car using OpenCV, a person, or something else.

The goal of this internship is to create a flexible multi-agent system (MAS) architecture including a few exemplary demonstrators. In these examples, it is made clear how each UAV makes decisions and how data is shared.

Contents

List of Figures

List of Tables

List of Abbreviations

MAS   multi-agent system

UAV   unmanned aerial vehicle

ROS   robot operating system

FIPA   foundation for intelligent physical agents

XML   extensible markup language

GCS   ground control station

OS   operating system

CI   continuous integration

CD   continuous deployment

3D   three-dimensional

AI   artificial intelligence

API   application program interface

GPS   global positioning system

UDP   user datagram protocol

TCP   transmission control protocol

SDF   standard database format

CUDA   compute unified device architecture

GPU   graphics processing unit

IP   internet protocol

LAN   local area network

SLAM   simultaneous localization and mapping

Introduction

A new technical era has arrived with current development in UAV technologies with it becoming available to the public. The UAVs of today are capable of accomplishing tremendous complex tasks on their own. But what if the potential of multiple UAVs are combined in a multi-agent system? How will these UAVs cooperate and how can this be implemented? To define the capabilities of a multi agent-system multiple scenarios are tested with multiple UAVs.

In this thesis an example of a multi-agent system will be explored to familiarize the reader with the concepts of agents, multi-agent systems, and the cooperation of agents in these systems. First the technologies will be explained to help the reader comprehend the concepts. In the example a multi-agent system is set up where between 5 and 10 UAVs can cooperate as intelligent agents, communicating through a blackboard communication system. Each agent is able to decide its own actions autonomously. The entire system is run inside docker containers with the ability to visualize the environment in Gazebo, a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments.

1 Research topic

1.1 Research

1.1.1 Problem

Currently there is no framework available to easily prototype multi-agent UAV systems in a simple environment. Creating a flexible multi-agent architecture would provide an effective solution for developers that want to start prototyping UAV systems. In this project the architectures of a multi-agent system must be demonstrated with examples. A generic solution must be found for creating easy prototyping. Examples of how to use the different architectures must be present with documentation on how to create them.

In case a company/person wants to develop such technologies, they have to start off from scratch which would mean they to invest time/money into developing such environments. Using this project, it would erase the issue of a developer having to create their own environment. With examples of the architectures allready built-in it has the ability to boost a project by months.

The current state of the project is a proof of concept with a single UAV, developed for firefighters to give live feedback from the air, scanning the area for possible dangerous chemicals.

1.1.2 Question

How can a flexible MAS architecture for robotics be used in rapid UAV prototyping

1.1.3 Sub-questions

1. What is an Agent?
2. What is a MAS?
3. What is a MAS architecture?
4. What is a blackboard communication system?
5. What are the advantages of a MAS
6. What are the necessary/core components of a MAS architecture?
7. Are there usable off-the-shelf rapid robot/UAV prototyping MAS architectures?
8. What are the pros and cons of off-the-shelve rapid robot/UAV prototyping MAS architectures?

1.1.3.1 What is an Agent?

There has yet to be a true definition of what an agent is. A lot of different literatures define an agent in their own way. All these definitions are strongly based on the background of the research they are conducting. By S. Russel [10], an agent is defined as :

> *"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors."*
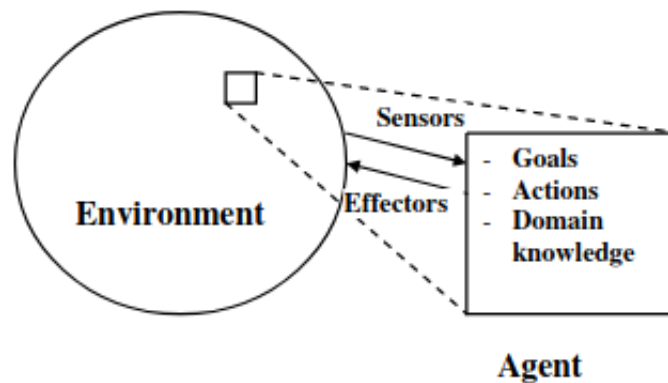
Figure 1: How the environment can be perceived by an agent[1].

This means that an agent is an entity that can perceive its environment through sensors and perform tasks using data received from these sensors.

An agent has two basic properties, being autonomous and being situated in an environment [3]. To achieve autonomy the entity must have a goal. An autonomous agent will try and achieve this goal using a set of actions. The agent will decide independently what the best set of actions is. During this process there will be no direct intervention from other entities.

To enable an agent to be situated it must have the ability to adapt to dynamic (changing rapidly, unpredictable (not static) and unreliable environments (it is not able to predict future states of the environment).

An intelligent agent has additional properties :

- Reactive
- Proactive
- Flexible
- Robust
- Social

A dynamic environment is a repidly changing environment. An agent is often situated in a dynamic environment. Because of this it will need to adapt to each change to keep pursuing the goal, the agent is reactive.

The second additional property, proactive, is that the agent will keep pursuing the goal over time. If an attempt fails, the agent will need to recover and be robust. To achieve robustness the agent will need a range of actions to achieve the goal, so it can be flexible in case some actions are not available, have failed before.

Lastly, an intelligent agent is able to interact with other agents and share its knowledge, the agent is social.

1.1.3.2   What is a MAS?

Like the definition of an agent, there has yet to be a true definition for a multi-agent system. Yet again there are a lot of definitions, each depending on the background of the research they are conducting. For this research the definition is chosen from P. Stone. By P. Stone [7], a multi-agent is defined as:

> ***"A multi-agent system is a loosely coupled network of problem-solving entities (agents) that work together to find answers to problems that are beyond the individual capabilities or knowledge of each entity (agent)."***

Per definition a multi-agent system is a group of agents organized to solve a problem a single agent would not be able to solve on its own. The agents are organized, they have a structure and are able to communicate with each other to share knowledge.

An agent in a multi-agent system has some important characteristics. As mentioned before, the agent must be autonomous in some form. It can make decisions on its own without any intervention. Secondly, no agent has a full view of the world. And thirdly the system must be decentralized. There is no controller controlling each agent independently.

There can be different types of internal hierarchies in a multi-agent system, depending on how the developer wants the agents to work together.

1.1.3.3 What is a MAS architecture?

In multi-agent systems there are different ways the agents can be organized. Depending on the architecture of this organization the system will adapt to the given problem in different ways depending on the architecture.

When agents work in teams, each team has its function. When a team reaches a goal, it will share this information with the other teams.
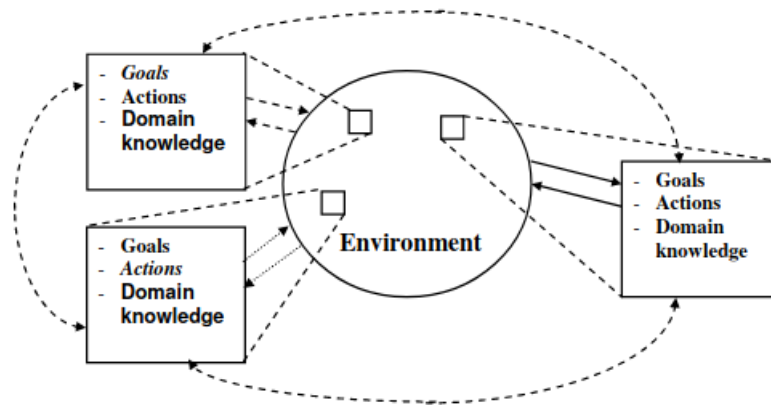
Figure 2: Communication as teams[2].

For example a patrol, search, and track problem. Where each team has its own goal. Team A will patrol an area and report if it finds some abnormalities. Team B will search for these abnormalities in the area and report any findings to team C. Team C will then on its turn start and track these objects.

Coalitions, like teams, consist of multiple UAVS working together. In coalitions not each UAV has the same goal. The agents will adapt to the information and the environment to change their goal.
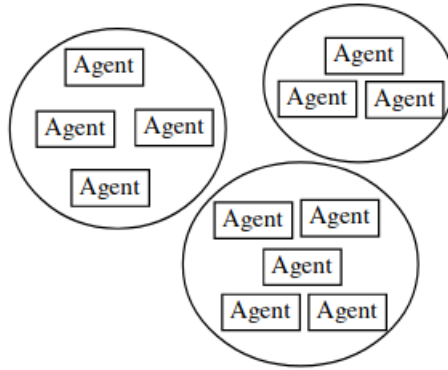
Figure 3: Communication as coalitions[3].

Let's take the same example from teams. Each coalition will take on a subarea of the original area. Within the coalition, each agent will take action depending on the current state of the area (if an object is found or not). Depending on the information an agent can decide to switch actions, for example from searching to tracking.

In a hologenic hierarchy all the agents will communicate with each other, prioritizing the one closest to them. Each UAV will share its current state and other agents will adapt to this information. When receiving new information from the blackboard system, agent states can change, updating the entire hierarchy.
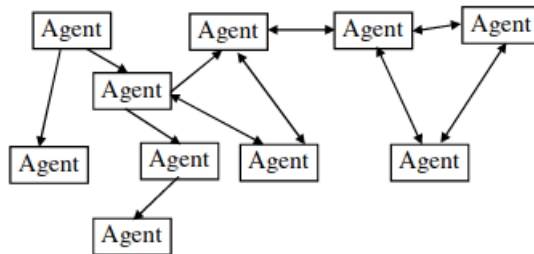
Figure 4: Communication as hologenic hierarchy[4].

1.1.3.4 What is a blackboard communication system?

To enable the agents to communicate with each either in a multi-agent system a blackboard communication system is introduced. The blackboard system is essentially a node containing all the information for agents to share with each other. The information can be separated using different topics depending on the internal hierarchy of the multi-agent system. As doing so the agents can be split into groups and only specific information will be shared while still maintaining information sharing inside the groups.

Figure 5: Blackboard communication[5].

1.1.3.5 What are the advantages of a MAS

When handling a large problem multi-agent systems provide a robust and scalable solution. By spreading the computational workload the system is not only fast but also very reliable.

When an agent fails in a multi-agent system the system will have the availability to recover fairly easily from this due to the robustness. Control and responsibilities are shared over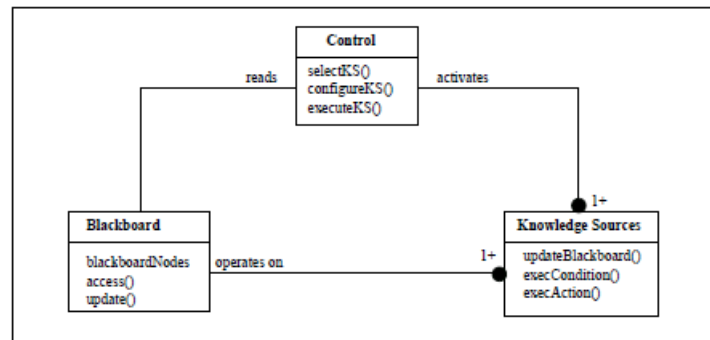 all the agents. The system can tolerate failures of one or more agents by adding the failed actions that were being performed by these agents back into the blackboard system. Other agents can then try and complete these actions again.

Another advantage of a multi-agent system is scalability. Adding another agent to a multi-agent system is easier than expanding an already existing application. In a multi-agent system, all the agents are identical therefore introducing a new agent will only update the internal hierarchy.

1.1.3.6 What are the necessary/core components of a MAS architecture?

A multi-system does not have a lot of necessary core components to be able to work. First of all the system does need to have multiple intelligent agents. These agent need to be able to "see" their invironment with sensors. These sensors can very depending on the system.

The data from the sensors from each agent, the knowledge each agent has, needs to be sharable with the other intelligent agents. Using this knowledge the system must distribute the computational load over the agents, No agent is designated as controlling.

1.1.3.7 Are there usable off-the-shelf rapid robot/UAV prototyping MAS architectures?

1. micros_mars_task_alloc

   ROS has a package, micros_mars_task_alloc, that is used for multi-task allocation. It is based on a multi-agent theory using an ALLIANCE model. The ALLIANCE model uses a cooperative robot team where each robot is an intelligent agent.

   The ALLIANCE model is based on Brooks' subsumption model. A subsumption architecture is a reactive robot architecture that tries to solve the problem of intelligence from a different perspective than traditional AI [9].
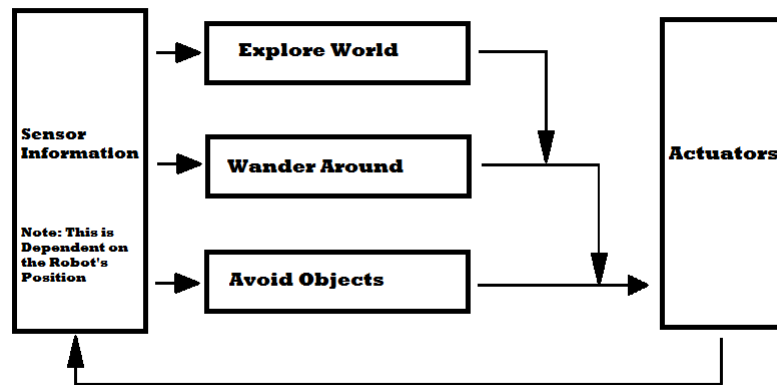
Figure 6: Subsumption architecture abstract diagram[6].

   The ALLIANCE model adds several different behavior sets and behavior layers. These sets and layers are both implemented by following the subsumption model.

2. FIPA
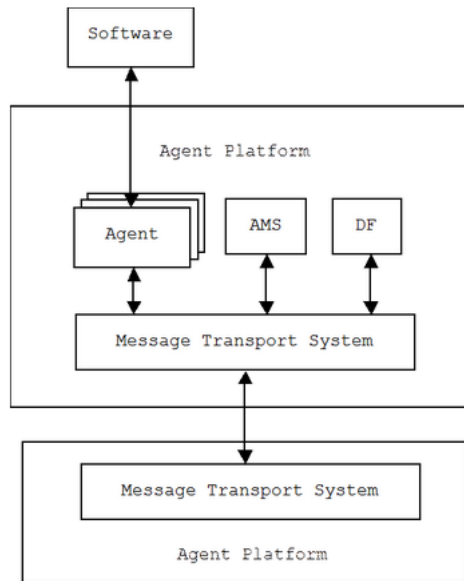
Figure 7: FIPA diagram[7].

FIPA (Foundation for Intelligent Physical Agents) is an open source agent platform from Nortel Networks. The platform uses an agent communication language that conforms to the FIPA agent standards. [11]

1.1.3.8 What are the pros and cons of off-the-shelve rapid robot/UAV prototyping MAS architectures?

One of the biggest cons with off-the-shelve multi-agent architectures is that they are allmost always designed for a specific research topic. The interactions with the environments are not generic and prototyping is very difficult because of the limitations in setting the goals of the intelligent agents. The data sharing between agents is often immutable and therefor adding sharable knowledge is impossible.

An advantage of using an existing architecture is that the research is allready conducted and the limitations found by the developer. Knowing the limits of an architecture can help optimize problem solving by knowing its optimal performance setup.

1.1.4 Research method

1.1.4.1 Experiments

In this project a lot of different small components work together to form a bigger system. To test each different component small experiments were tested on a smaller scale. Each experiment was run in a smaller system to isolate problems. By isolating a component it is easier to write generic code for further use.

This provides more insight on each component and how they work. If experiments turn out positive they are implemented in the bigger system.

1.1.4.2 Observing other projects

A lot of components allready have been coded in other projects in different ways. Observing other projects gave a lot of insight on how to tackle a problem or on how to evaluate an experiment.

2 Traineeship report

2.1 About the company

Within the PXL-Digital department, Electronics-ICT and Computer Science clusters are uniquely integrated. PXL-Digital is creating a new platform to take the lead in many areas; think of Internet of Things, embedded software, hardware, security, cloud, Full-stack Development, AI & Robotics, Augmented & Virtual Reality, Software Quality ... This creates a rich palette of specializations within digital technology and digital business.

The community-driven educational model is central. This community consists of junior colleagues (our students), colleagues, companies and researchers. PXL-Digital creates innovative education by and for this community.They leave the metaphor of building bridges between education and the business world and continuously bring these worlds together.

2.2 Technologies

2.2.1 ROS

Figure 8: ROS logo[8].

The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. ROS enables the user to simplify the creation of robust robot applications. Using nodes, ROS provides an easy method of communication between a robot and the user or, in case of multiple robots, between robots themselves.A ROS node is a process that performs computation, it is an executable program [18]. All the nodes are connected to ROScore, a collection of ROS nodes, differentiated by name. Each node can contain multiple topics,with each topic assigned a topic name. A topic is a bus over which nodes exchange messages [19]. Using this topic name a user can create subscribers (to access information from this topic) and publishers (to add new data to the topic). Information can be shared between nodes using these subscribers and publishers from different nodes [1].
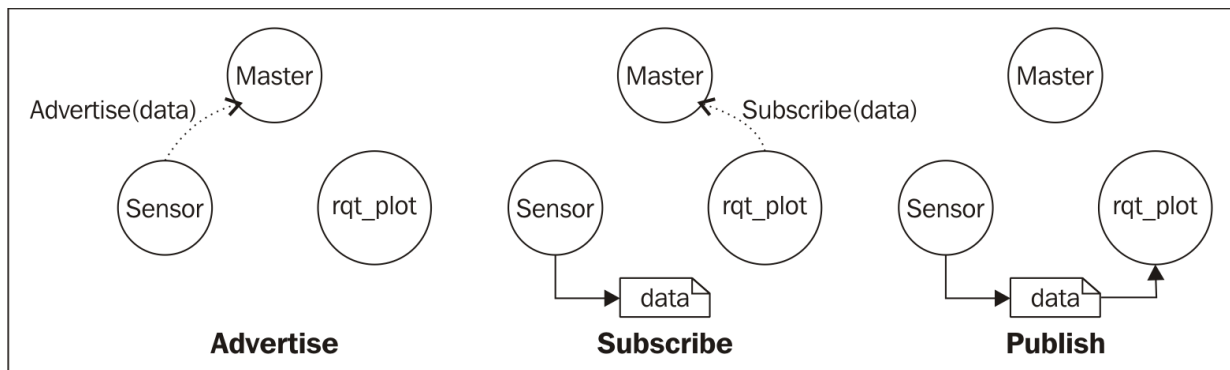

Figure 9: ROS core[9].

2.2.2 MAVLink

MAVLink is a very lightweight messaging protocol for communicating with air vechicles. Data streams are sent/published as topics. The messages are defined within XML files. MAVLink is very efficient, it does not require additional framing and is very well suited for projects with limited bandwidth. Next to being very efficient, MAVLink is also very reliable. It provides methods for detecting corrupt packages, if any packets have been dropped, and for authenticating packets. A developer is able to control 255 concurrent systems on the network when using MAVLink, with each system having the ability for onboard and offboard communication [4].

Figure 10: MAVLINK logo[10].

An example of a mavlink message is a HEARTBEAT message. A vechicle must regularly broadcast their HEARTBEAT to make sure it is still connected. A developer can set the rate at which the message is broadcasted, and how many of these messages may be missed before a vechicle is labbeled missing or disconnected [5].

Many languages support MAVLink v1, MAVLink 2 and Message Signing. There are also prebuilt MAVLink C libraries available on the MAVLink website.

2.2.3 MAVROS

Mavros is a MAVLink extendable communication node for ROS with a proxy for GCS (ground control station). A ground control station is a land- or sea-based control center that provides the ability to control unmanned vehicles. This is the crucial link between MAVLink and ROS. It is a ROS "node" that can convert between ROS topics and MAVLink messages allowing ArduPilot vehicles to communicate with ROS [6]. Next to providing communication between ROS and the ArduPilot vehicles MAVROS also contains a multitude of builtin methods to simplify coding when handling basic actions of a vehicle. MAVROS provides, for example, methods to arm and take off with a UAV.

2.2.4 PX4

PX4 is an open-source autopilot system oriented toward inexpensive autonomous aircraft. It is supported by an active worldwide community, it powers all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles [8].

Figure 11: PX4 logo[11].

PX4 can be used with ROS for offboard control. Offboard mode is primarly used for controlling a vehicles position, velocity and thrust. It requires an active connection to a remote MAVLINK system (HEARTBEAT messages).

2.2.5 Docker

Docker is a set of platforms as a service product that uses OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries, and configuration files.

Figure 12: Docker logo[12].

Containers use shared operating systems. This means that they are much more efficient than virtual machines. Instead of virtualizing hardware, containers rest on top of a single Linux instance so they leave most of the virtual machine behind. Another great feature of docker is that it is great for Continuous Integration/Continuous Deployment (CI/CD). This methodology comes from DevOps. Developers can share and integrate code into a shared repository early and often. Offering quick and efficient deployment [16].

Using docker in combination with ROS gives a team the ability to easily share code and update dependencies. Using bash scripts new developers can easily clone the project and build their own docker containers. Using these containers a developer can then start their own project without worrying about the dependencies because these are all installed in the container images.

2.2.6 Gazebo

Gazebo, an open-source 3D robotics simulator, makes it possible for developers to rapidly test algorithms, design robots, perform regression testing, and train AI systems using realistic scenarios. In gazebo the developer can create custom environments. These environments have realistic rendering including high-quality lighting, shadows, and textures. Sensors that can see the simulated environment on robots can be modeled [2].

Figure 13: Gazebo logo[13].
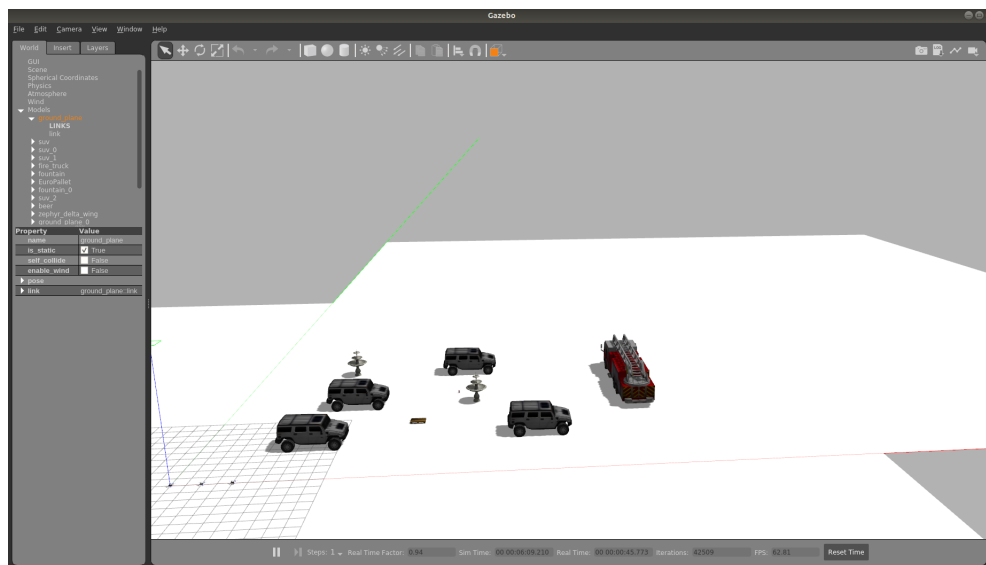
Figure 14: Gazebo Simiulation[14].

2.2.7 Python

Python is an indentation language emphasizing code readability with notable use of whitespaces. The standard library is very comprehensive. Object-oriented and structured programming are both fully supported. Python has not all functionality in its core, it is rather built to be highly extensible [12].

Figure 15: Python logo[15].

As a dynamically typed language, Python is very flexible. There are no hard limitations or rules on how to build an application or solution to a problem. Encountering errors python still compiles and runs your program until it hits the problematic part.

ROS provides a pure python client library, rospy. The rospy API enables python developers to easily interace with ROS, Services, and parameters.

2.3 Implementation

2.3.1 Iris drone

PX4 provides a default UAV, the iris. The iris is a quadcopter that has multiple control options, providing redundancy and flexibility. It has a built-in radio for real-time mission monitoring, data logging and control. For controlling the UAV it has powerful cross-platform ground station/mission planning and analysis software that runs on Windows, operating systems (OSs) X and Linux, providing simple point-and-click programming and configuration. And finally Maybe the most important feature for a developer: failsafe programming options in the event of lost control signal, global positioning systems (GPSs), or low battery conditions [14].

Figure 16: Iris drone[16].

Using xarco files a developer is able to add components to this UAV. In this project it is able to spawn up to 10 unmanned aerial vehicles (UAVs). This limitation does not come from the xarco files but from this project which will be later clarified As a generic solution the project contains a generate_launch script that can accept a parameter for n amount of UAVs. This script is written in bash so the argument can be passed in the command line. It generates a launch file as a string, and then writes this launch string to a file named "smartuavs.launch" inside the launch folder. This will later be further detailed in the simulation container.

In the project a 3D-camera is added using a components snippets from the PX4 github repository. The 3D camera has multiple topics where it publishes to where it publishes data.

2.3.2 Agent implementation

Each UAV is an agent. To create an agent in ROS, firstly a node is created. A node is an executable that uses ROS to communicate with other nodes [21]. For this small example the name UAV_0 is given to the node.

```
rospy.init_node('UAV_0', anonymous=True)
```

The UAV is able to situate itself using an internal GPS. Using MAVROS the developer is able to control a UAV. MAVROS will translate the messages sent from ROS via MAVLink so the UAV can "understand" these messages. To send a message the developer must first create a publisher to a specific topic, in this example the UAV is sent to a specific location.

```
point_push = rospy.Publisher('/iris_0/mavros/setpoint_raw/local',
                                     PositionTarget, queue_size=1)
```

A publisher is made for the topic '/iris_0/mavros/setpoint_raw/local'. When publishing to this point the data must be a PositionTarget. Each topic has a defined type of message so that MAVROS can translate this message to MAVLink. Following the same reasoning multiple publishers can be made for more complex movements or actions. Next to creating more publishers a developer also has the ability to create custom topics with custom messages.

A UAV can also receive information with a subscriber. Like in a publisher the type of message has to be defined. Finally a callback has to be given as an argument where the data is processed. These callbacks are default run single-threaded. This can cause some performance issues in bigger problems. To solve this problem ROS provides Multi-threaded spinning and AsyncSpinner [20]. For this short example, processing a local GPS message, a subscriber is needed with the callback.

```
gps_local_sub = rospy.Subscriber('iris_0/mavros/global_position/local',
                                       Odometry, self.gps_local_callback)
```

The subscriber receives messages from the topic 'iris_0/mavros/global_position/local' where the message type is Odometry. The callback, self.gps_local_callback , will process the data each time it is updated.

```
def gps_local_callback(self, info):
    pose = info.pose
    self.x = pose.pose.position.x
    self.y = pose.pose.position.y
    self.z  = pose.pose.position.z
```

In this case the local position of the UAV is updated each time the callback is called. Multiple callbacks can be made in a single node to further improve the complexity and possibilities of the agent.

2.3.3 Simulation

//TODO rewrite with scheme

The project contains multiple containers, each built from scratch. Each container has its purpose. To create all the images for the containers there is a script, 0_build_images.sh, that will build all the images for the containers. The start_terminals.sh script will take the argument to spawn n UAVs. This will start multiple containers and it gives the user a terminal tab for each container. The code in each container is in a catkin package. Catkin is the official build system of ROS.
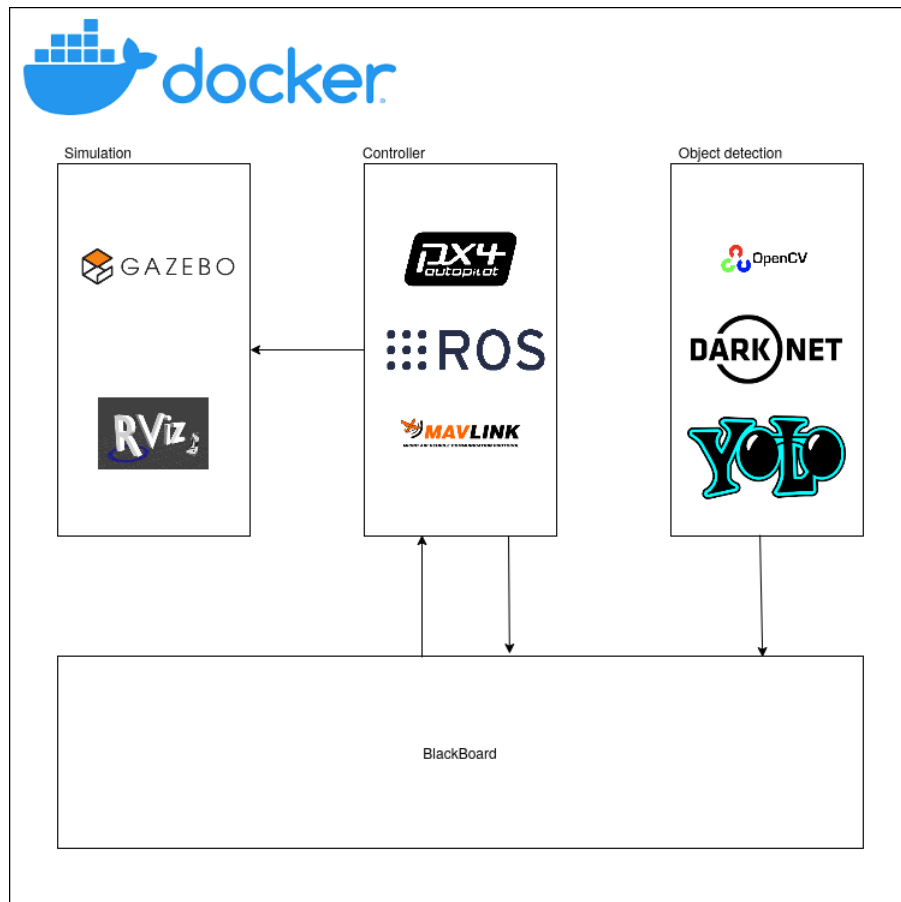
Figure 17: dockerscheme[17].

The first container is responsible for running Gazebo, the simulation. The container is built with the image *smart_uav_px4* from another project built by fellow colleagues Vic Segers and Borcherd Van Brakel. ROS, MAVLink, MAVROS, PX4 firmware are all installed on this image. The simulation container is responsible for spawning all the UAVs in a world. As previously, the script *generate_launch.sh* creates a launch file by generating a string and then writing this string to the *smartuavs.launch* file.

Inside this launch file a xarco file is passed as an argument for each UAV. The xarco file, *iris_base.xarco*, that is passed is in the folder *rotors_description/urdf*. The file takes two arguments, a TCP and a UDP port. Giving each UAV different UDP and TCP ports gives the ability to control each UAV separately.

Inside the *iris_base.xarco* file all the components for the iris UAV are added. Before adding a component the file where it is originated must be included first.

```
<xacro:include filename="$(arg rotors_description_dir)
                         /urdf/component_snippets.xacro" />
```

After including the xarco file, a component can be added to the UAV. Some components can take arguments like the 3D camera, the *vi_sensor_depth_macro*, can take some arguments on where to place the camera, how to orient the camera, a namespace, a parent link, a camera suffix (for the topic where it publishes its data to) and a framerate.

```
<xacro:vi_sensor_depth_macro
    namespace="${namespace}"
    parent_link="base_link"
    camera_suffix="iris_depth"
    frame_rate="30.0">
<origin xyz="0 0 0.01" rpy="0 1.57079632679 0"/>
</xacro:vi_sensor_depth_macro>
```

If there is only one UAV, SDF files can be used to simplify the process. The reason xarco files are used UAV is that they can take arguments whereas SDF files can not. This means the UDP and TCP ports can not be adjusted so only one UAV can be controlled [15].

The simulation container also contains Rviz, a 3D simulation tool for ROS. A developer can add topics, for example a point cloud from the camera, and Rviz will visualize this data. Finally to run the simulation and Rviz there is a script called *1_run_launch.sh* inside the container. As a first argument (-a) it takes n amount of UAVs. The default value is the n amount of UAVs previously iven by starting the containers. The second argument (-w) is a path to a world file. The third argument (-g) is a boolean if Gazebo needs to have a visualization window. Without a window for gazebo the performance is a lot higher because of the limitation in gazebo rendering all the data, thus giving a better real-time factor.
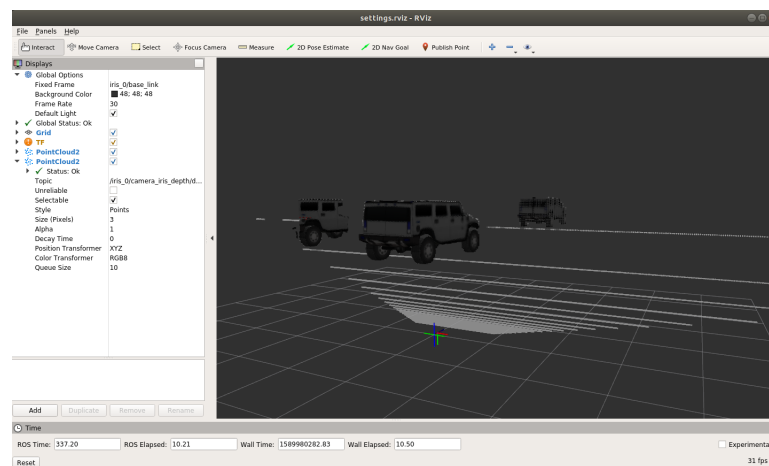

Figure 18: RVIZ[18].

Rviz, abbreviation for ROS visualization, is a visualization software that allows a developer to view data from Gazebo (or real-world). It can capture data, capture sensor information, replay captured data and provide a viewpoint of a robot. All this information comes in the form of topics. A developer can choose multiple topics in Rviz to visualize. Rviz has the ability to visualize multiple topics at the same time while maintaining high performance. By viewing the standpoint from a robot debugging is made substantially easier for a developer [17].

Logging information is a crucial part of developing with ROS. Rosbags provide an easy way of logging information. These files can be quite large so ROS provided a way of capturing only crucial data. One or more topics can be listed after the command "rosbag record", capturing only data given by the user. When playing back this rosbag will loop over the data, while Rviz visualizes it. A developer can use this for tracking down bugs, optimizing code, implementing features, etc.

2.3.4 The blackboard system

The second container, the blackboard system, enables each agent to communicate with eachother. The container uses the image *smartuav_ros_mavros*. MAVLink, MAVROS and pymavlink are installed in this image. Using these libraries a developer can construct custom messages that can be updated in the blackboard system. The blackboard can then publish these custom messages as a bundled package to a custom topic. A problem can be placed inside this package with flags and actions for the agents.

When handling a problem, each agent will take a portion of the problem and iteratively update the blackboard node. For example searching for an object in a large field. Each agent will take a portion of that field and search for the object. When an agent finds the object it updates the blackboard, therefore notifying the other agents that the object is found [13].

When implementing a blackboard system in the project it unlocks a whole new set of possibilities. UAV collision prevention between each other, for example. Next to just GPS information agents are now able to distribute a problem over multiple agents. Communication between agents introduces a multi-agent system.

The limitation of 10 UAVs is in this container. To add information to the blackboard a subsriber has to be made for each UAV. unfortunatly each substriber callback needs to have its own function hard coded to be thread safe. If this is not done each callback would refer to the same function and it would scramble to blackboard data.

2.3.5 Object detection

The third container is responsible for detecting objects for each agent.

The detector uses OpenCV 4.3.0 built from source. To optimize performance, OpenCV has to built with CUDA backend. When building from source additional parameters can be given with *cmake*. For cuda compatibility :

```
-DWITH_CUDA=ON \
-DWITH_CUBLAS=ON \
-DWITH_CUDNN=ON \
-DOPENCV_DNN_CUDA=ON \
```

All these parameters have to be set to ON for OpenCV with GPU compatibility. When running the container the GPU hass to be given :

```
--gpus all
```

And inside the code the prefered dnn backend has to be set.

```
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
```

When starting the object detection, a multi process will start. For each UAV a detector will be made with a pre-trained tiny-yolov3 loaded into the constructor. The detector will then use a subsriber to link the camera feed from the UAV to the callback where frames from the feed will be processed with tiny-yolov3.

When tiny-yolov3 detects an object it will draw a rectangle around the object in a color assigned to this object. the nex image can be displayed using opencv. Next to displaying the new image with the detected objects, the detector also publishes the coordinates of the rectangle drawn around the found object for later usage, for example human tracking.

//TODO add screenshot of detection

2.3.6 Controlling the agents

The last container, the controller, is responsible for starting the multi-agent system. Each UAV will run as a parallel process in the simulation. To start a UAV, a specific pattern of commands has to be followed. Before starting a connection, a heartbeat must be made with each UAV that must be maintained throughout its flight. When a connection is lost, that specific UAV will stop its current action and land on its home point.

When a connection is established, a developer is able to arm the UAV. It is possible to visually confirm that a UAV is armed when its rotors are spinning. After arming a home point has to be set; this is a safe place where the UAV can land in case of connection loss. When a UAV is armed and has a home point, it is ready for further instructions. These agents fly autonomously. For a UAV to fly without user input it has to be put into the "OFFBOARD" mode. Finally, when a UAV is in this mode it can be added to a team, coalition, or hologenic hierarchy. Depending on the hierarchy a UAV will take on an action and help to try and solve the given problem.

2.3.7 Docker Swarm

A docker swarm is a group of virtual or physical machines operating together in a cluster. In the cluster, there is a master where other machines can join as workers. This master nodes functions as the manager node, with the primary function being assigning tasks to worker nodes [2].

In the project, there is a subfolder with scripts to construct a docker swarm. For ROS nodes to be able to communicate with each other, they need to be connected to the same ROS master. When executing the script *0a_create_docke_rswarm.sh* a docker swarm is initialized. With the swarm initialized the script automatically constructs an overlay network called *my_ros_overlay_network*. This overlay will be passed as network arguments when starting a new docker container. These containers will then be attached to the overlay network. When the script completed initializing the swarm and the overlay network it will return a command to join the swarm.

A container is attached to the network when the network argument is passed with the overlay network :

```
--network=my_ros_overlay_network
```

A linked volume is created for the container to be able to access generated files later.

```
-v ‘pwd‘/../Commands/bin:/home/user/bin \
```

On the master node the simulation is started with the script *2_start_simulation_attached.sh*. This will start a container attached to the network. before starting the simulation, the *ROS_MASTER_URI* has to be set accordingly. *0c_generate_ros_config_command.sh* first checks if the worker is the master or not. When a machine is the master the IP address is set to *localhost*.

```
IP=$(docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
smartuav_simulation_attached) > ../Commands/bin/set_swarm_settings.bash

echo "export ROS_HOSTNAME=$IP" >> ../Commands/bin/set_swarm_settings.bash
echo "export ROS_IP=$IP" >> ../Commands/bin/set_swarm_settings.bash
echo "export ROS_MASTER_URI=http://localhost:11311"
>> ../Commands/bin/set_swarm_settings.bash
```

These variables are saved in a file *set_swarm_settings.bash* and can now be sourced from inside the container.

To enable workers to communicate with the ROS master on the master node of the swarm, the overlay IP address has to be known. The script *0b_generate_overlay.sh*, on the master node (outside of running container), will return the IP address of the running container.

On a worker machine, *0c_generate_ros_config_command.sh* first checks if the machine is the host or not. When it is not the script will need an input; the IP. Instead of setting the IP to localhost, the script will now generate the *ROS_MASTER_URI* correctly for each worker.

```
read -p "IP on overlay network of the master container: " master_ip
read -p "Name of the running container: " running_container

readarray -d _ -t strarr <<<"$running_container"

dir=${strarr[1]}
#empty out settings
echo "" > ../src/$dir/Commands/bin/set_swarm_settings.bash

IP=$(docker inspect -f
'{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
$running_container)

echo "export ROS_HOSTNAME=$IP"
>> ../src/$dir/Commands/bin/set_swarm_settings.bash

echo "export ROS_IP=$IP"
>> ../src/$dir/Commands/bin/set_swarm_settings.bash

echo "export ROS_MASTER_URI=http://$master_ip:11311/"
>> ../src/$dir/Commands/bin/set_swarm_settings.bash
```

A worker needs to the IP of the master container running the simulation and the name of the current running container. The current running container name is needed to get the IP of that container to set the variable *ROS_IP*. If this is not done the subribers would not be able to receive messages from this node. The script will then automatically place the *set_swarm_settings.bash* in the correct linked volume folder. This file can now be sourced from inside the container.

To test if each worker is connected to the ROS master the simulation has to be started with *1_start_simulation_attached.sh*. This will start Gazebo, Rviz, and roscore. If the command *rostopic list* from inside each worker container returns the topics, the worker is connected to the master and is able to communicate with it.

2.4 Future Work

2.4.1 Multithreading object detection

In the current state of the project, a normal multiprocess is started with the function *pool* from the multiprocessing module. OpenCV 4.3.0 built with CUDA 10.2 backend does not support this. To fix this problem te library *pytorch* has to be used. Pythorch is an open source machine learning library based on the Torch library, used for applications such as computer vision. In this library there also is a multiprocessing module providing support for CUDA-10.2. By running the object detection on the GPU as a multiprocess it processes up to 500x faster.

2.4.2 Optimizing docker swarm

When currently starting a swarm a lot of scripts have to be manually executed. One of the optimizations would be to automize this using python. The python script should generate all the files needed and distrubute them over the docker containers using the overlay network. The user would no longer execute every script on every worker, thus reducing the chances of failing.

When running the docker swarm on a LAN connection it is possible it will fail. This is due to the fact that some ports refuse connection from other machines. When this happens the system will throw a *IOerror* in the controller container.

2.4.3 Adding other components

Currently there are not a lot of addition components in the project. There is a possibility to add different components using extra docker container or using the existing ones. One example of an extra component is the SLAM (Simultaneous localization and mapping) algorithm. It can be added using an extra container. The code to run this in a multiprocess environement still has to be written.

When adding new components that need to be run in the swarm the name has to be justified has : *workspace_linkedvolumefoldername_**. It is very import that the linkedvolumefoldername is correct because the script *0c_generate_ros_config_command.sh* splits this string and takes the second argument to place the *set_swarm_settings.bash* on the correct place.

2.4.4 Implementing a decision making model

An agent decides its task purely on what is available in its architecture. It can be optimized by implementing a decesion making model in the multi-agent system. This would introduce a whole other concept on itself. The model would need training and testing for different scenarios. When a good model is implemented it would yield far better results than the current decision making.

3 Conclusion

type conclusion here

Bibliographical references

Bibliography

[1] Ros. URL `https://www.ros.org/about-ros/`.

[2] GAZEBO, 05/05/2020. URL `http://gazebosim.org/`.

[3] M. W. Lin Padgham. *Developing Intelligent Agent Systems: A Practical Guide.* John Wiley And Sons, 2005.

[4] MAVLink, 31/04/2020. URL `https://mavlink.io/en/`.

[5] mavlink.io, 17/04/2020. URL `https://mavlink.io/en/services/heartbeat.html`.

[6] MAVROS, 31/04/2020. URL `http://wiki.ros.org/mavros`.

[7] M. V. P. Stone. *Multiagent systems: A survey from a machine learning perspective.* Autonomous Robots, 2000.

[8] PX4, 31/04/2020. URL `https://docs.px4.io/master/en/index.html`.

[9] ROS, 20/04/2020. URL `http://wiki.ros.org/micros_mars_task_alloc`.

[10] P. N. S. Russel. *"Artificial intelligence – A modern approach.* Prentice Hall, 1995.

[11] R. H. Stefan Poslad, Phil Buckle. *The FIPA-OS agent platform: Open Source for OpenStandards.* Imperial College of Science, Technology and Medicine, Exhibition Road, London, SW7 2BZ, 2000?

[12] todo, 05/05/2020. URL `https://en.wikipedia.org/wiki/Python_(programming_language)`.

[13] todo, 20/03/2020. URL `https://en.wikipedia.org/wiki/Blackboard_system`.

[14] todo, 31/05/2020. URL `http://www.arducopter.co.uk/iris-quadcopter-uav.html`).

[15] todo, 31/05/2020. URL `https://github.com/PX4/Firmware/issues/14436`.

[16] S. J. Vaughan-Nichols, 31/04/2020. URL `https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/`.

[17] R. wiki. URL `http://wiki.ros.org/rviz`.

[18] R. wiki, 20/03/2020. URL `http://wiki.ros.org/Nodes`.

[19] R. wiki, 20/03/2020. URL `http://wiki.ros.org/Topics`.

[20] R. wiki, 31/05/2020. URL `http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning`.

[21] R. wiki, 31/05/2020. URL `http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes`.

4 Appendices

4.1 Description Appendix A

4.2 Description Appendix B

4.3 Description Appendix C