Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Automazione

Dipartimento di Elettronica, Informazione e Bioingegneria


# Thesis
# Title


Advisor: Prof. Name SURNAME

Co-Advisor: Ing. Name SURNAME

Thesis by:

Name SURNAME Matr. 000000

Academic Year 2016–2017

*To someone very special...*

# Acknowledgments

# Abstract

# Sommario

# Contents

# List of Figures

# List of Tables

# Introduction

# Chapter 1

# State of the art

The literature about the "Consensus" has grown significantly in the latest years because of the increasing presence of autonomous vehicles. This, in accordance with the new improvements in robotics, has brought to growing interest in consensus between multiple agents which have to accomplish a mission in cooperative or adversarial scenarios.

The "Consensus" theory has its roots in Graph Theory and Automatics and it can be used to coordinate a mission in order to achieve the synchronization between the vehicles even during unforeseen events which force one or more components of the mission to change the planned trajectory or task. In this case, the other components identify this variation and they act to preserve the synchronization.

The "Consensus" algorithm is a distributed algorithm which can sometimes be simulated in a centralized fashion because of the reduced computational power of the machines involved in the mission, which are equipped with low power hardware to account for the crucial issue of power consumption. In this scenario, a centralized server simulates the algorithm and communicates with all the machines.

The most common "Consensus" application is a spatial and timing consensus: in this implementation, each vehicle of the formation has to travel along a specified trajectory and the completion of the mission occurs when all the agents reach the final positions of their spatial paths. The algorithm has to guarantee that the difference between the ending time at which all the vehicles finish their tasks is minimized and asymptotically goes to zero, when the execution time goes to infinity and no other unforeseen events happens. We also consider formations of Unmanned Aerial Vehicles, but the key concepts can be freely applied to other categories of robots when they are able to follow a trajectory. This study is focused on this kind of application, and further simulation results and experimental achievements are presented in chapters 5 and 6.

In the next sections we refer to the main work of Venanzio Cichella [3] in the field of UAV consensus, in order to provide an homogeneous state of the art about the "Consensus". The paper considers all the details needed to build a "Consensus" system. The main components of this kind of system are listed and explained below:

- Parametrized trajectory
- Virtual time
- Consensus law
- Network topology

## 1.1 Parametrized trajectory

The trajectory is a spatial path with an associated timing law and it is used to identify the position of the center of mass of our agent at a given time. We can start with the following definition of a generic trajectory $p_{d,i}(t_d)$ for $N$ vehicles:

$$p_{d,i} : [0, t^f_{d,i}] \rightarrow \mathbb{R}^3, \quad i = 1, 2, \ldots, N \tag{1.1}$$

where $t_d \in [0, T_d]$, with $T_d := max\{t^f_{d,1}, \ldots, t^f_{d,N}\}$, is the time variable of the trajectory, while $t^f_{d,i} \in \mathbb{R}^+$ are the individual final mission times of the vehicles obtained during the planning phase. Usually all this final times are equal and therefore $t^f_{d,1} = \cdots = t^f_{d,N} = T_d$, but we introduced the notation for the sake of completeness. Obviously, the trajectories need to be collision free and must comply with spatial and temporal constraints due to the dimension of the vehicle and its maximum velocities and accelerations. The trajectory can account also for the orientation, and therefore, the function might take images in $\mathbb{R}^m$, where $m$ is the number of dimensions considered. In the following sections, we will refer only to the position, but a more general theory can also be developed.

We now parametrize the trajectory using a dimensionless variable $\zeta_i \in [0, 1]$, related to the time $t_d$. In this way we can specify a function $\theta(\cdot)$, which represents the timing law associated with the spatial path $p_{d,i}(\zeta_i)$. We can specify this timing law using the dynamic relation of the form:

$$\theta(t_d) = \frac{d\zeta_i}{dt_d} \tag{1.2}$$

where $\theta(t_d)$ is a smooth and positive (the parameter increases when the time increases) function.

As it allows a one-to-one correspondence between the time variable $t_d$ and the parameter $\zeta_i$, an analytical expression for the function $\zeta_i(t_d)$ is desirable. Using the timing law defined in (1.2), the map $\zeta_i(t_d)$ is given by the integral:

$$\zeta(t_d) = \int_0^{t_d} \theta_i(\tau) d\tau \tag{1.3}$$

Usually, all this functions are defined as polynomials in order to make quicker and easier the evaluation process, since multiplication and addition are the basic operations in a digital processor. However, it is not mandatory to use them and a generic shape for the functions can be designed. We have defined all the elements of a trajectory and we go into detail about the trajectory generation phase. Further information about boundary conditions and flyable trajectory, which satisfy the dynamic constraints of the vehicles, can be found in [3] and is extensively analyzed in [2], [4], [5].

## 1.2 Virtual time

Given $N$ collision free trajectories, we want each vehicle to follow a *virtual target*, moving along the path computed offline by the trajectory generation algorithm. The objective can be achieved introducing a *virtual time*, $\gamma_i$, which is used to evaluate the trajectory and can be adjusted online to reach the synchronization even when external disturbances occur. Thus, the position of the $i^{th}$ virtual target is denoted by $p_{d,i}(\gamma_i(t))$ and the $i^{th}$ vehicle tries to follow it, by reducing to zero a suitably defined error vector using control inputs.

Considering the trajectory $p_{d,i}(t_d)$ produced by the trajectory generation algorithm, we consider the virtual time $\gamma_i$ as a function of time $t$, which relates the actual time $t$ to mission planning time $t_d$.

$$\gamma_i : \mathbb{R}^+ \to [0, T_d], \qquad for\ all \quad i = 1, 2, \ldots, N \tag{1.4}$$

We can now define the virtual target's position, velocity and acceleration, which have to be followed by the $i^{th}$ vehicle at time $t$

$$\begin{aligned} p_{c,i}(t) &= p_{d,i}(\gamma_i(t)) \\ v_{c,i}(t) &= \dot{p}_{d,i}(\gamma_i(t), \dot{\gamma}_i(t)) \\ a_{c,i}(t) &= \ddot{p}_{d,i}(\gamma_i(t), \dot{\gamma}_i(t), \ddot{\gamma}_i(t)) \end{aligned} \tag{1.5}$$

With the above formulation, if $\dot{\gamma} = 1$, then the speed profile of the virtual target is equal to the desired speed profile computed at trajectory generation level. Indeed, if $\dot{\gamma} = 1$, for all $t \in [0, T_d]$, with $\gamma_i(0) = 0$, it implies that $\gamma_i(t) = t_d$ for all $t$ and thus:

$$p_{c,i}(t) = p_{d,i}(\gamma_i(t)) = p_{d,i}(t) = p_{d,i}(t_d)$$

In this particular case, the desired and commanded trajectories coincide in every time instant and also the velocity profiles coincide with the ones chosen at the trajectory generation time. If instead $\dot{\gamma}_i > 1$, it implies a faster execution of the mission; on the other hand, $\dot{\gamma}_i < 1$ implies a slower one.

We can now normalize $\gamma_i$ in order to have a range which is $[0, 1]$. We simply need to divide all by $T_d$. In this way, we could use Bezier curves in order to represent the spatial path. This kind of curves offer interesting properties for computing minimum distances between two of them and allow the computation of smooth trajectories. In any case, it is not mandatory to use them and a general function could be used instead.

The second order derivative of $\gamma_i$, $\ddot{\gamma}_i$, is a free parameter used to achieve the consensus. In the next section we will introduce the control law which commands its evolution during time and we will explain how it is possible to implement a distributed algorithm.

## 1.3 Consensus law

Now, we formally state the path following problem. We define as $p_i(t) \in \mathbb{R}^3$ the position of the center of mass of the $i^{th}$ agent and since $p_{c,i}(t)$ describes the commanded position to be followed by the agent at time $t$, the errors are defined as:

$$\begin{aligned} e_{p,i}(t) &= p_{c,i}(t) - p_i(t) \in \mathbb{R}^3 \\ e_{v,i}(t) &= v_{c,i}(t) - \dot{p}_i(t) \in \mathbb{R}^3 \end{aligned} \tag{1.6}$$

Then, the objective reduces to that of regulating the error defined in (1.6) to a neighbourhood of zero. This task is solved with an autopilot capable of following the set points computed from the desired trajectory at specified instances of time.

The virtual time is the parameter used to reach consensus between multiple vehicles. In fact, since the the trajectories are parametrized by $\gamma_i$, the agents are synchronized at time $t$ when:

$$\gamma_i(t) - \gamma_j(t) = 0 \quad for\ all \quad i, j \in 1, \ldots, N, \quad i \neq j \tag{1.7}$$

We can also control the rate of progression of the mission using a parameter $\dot{\gamma}_d \in \mathbb{R}$, which represents the velocity of the virtual time with respect to the real time. All the agents share this variable and they proceed at the same rate of progression if:

$$\dot{\gamma}_i(t) - \dot{\gamma}_d(t) = 0 \quad for\ all \quad i \in 1, \ldots, N \tag{1.8}$$

Adjusting $\dot{\gamma}_d$ we can decide the speed of the mission: for instance, if we set $\dot{\gamma}_d = 1$ and (1.7) and (1.8) are satisfied for all the vehicles, then the mission is executed at the speed originally planned in the trajectory generation phase. If instead we use $\dot{\gamma}_d > 1$ or $\dot{\gamma}_d < 1$ we carry out the mission faster or slower. This term can be changed in real time in order to avoid moving objects or unplanned obstacles, which make it necessary to change one of the path of the agents. For the purpose of "Consensus", the parameter is only a reference command, rather than a control input.

Now we introduce the coordination control law which regulates the evolution of $\ddot{\gamma}_i(t)$ during the time and determines $\gamma_i(t)$:

$$\begin{aligned}
&\ddot{\gamma}_i(t) = \ddot{\gamma}_d(t) - b(\dot{\gamma}_i(t) - \dot{\gamma}_d(t)) - a \sum_{j \in \aleph_i} (\gamma_i(t) - \gamma_j(t)) - \overline{\alpha}_i(e_{p,i}(t)) \\
&\dot{\gamma}_i(0) = \dot{\gamma}_d(0) = 1 \\
&\gamma_i(0) = \gamma_d(0) = 0
\end{aligned} \tag{1.9}$$

where $a$ and $b$ are positive coordination control gains, while $\overline{\alpha}_i(e_{p,i}(t)$ is defined as:

$$\overline{\alpha}_i(e_{p,i}(t)) = \frac{v_{c,i}^T(t)e_{p,i}(t)}{||v_{c,i}(t)|| + \epsilon} \tag{1.10}$$

with $\epsilon$ being a positive design parameter, $e_{p,i}$ the position error vector defined in (1.6) and $\aleph_i$ the set of the neighbors which can communicate with the $i^{th}$ vehicle (we will see details later). In the equation (1.9) we have four terms. The feedforward term $\ddot{\gamma}_d$ allows the virtual target to follow the acceleration profile of $\gamma_d$. The second term $-b(\dot{\gamma}_i(t) - \dot{\gamma}_d(t))$ reduces the error between the speed profile imposed by $\dot{\gamma}_d(t)$ and $\dot{\gamma}_i(t)$, which corresponds to the control objective given in (1.8). In particular, if $\dot{\gamma}_d(t)$ is one, then the virtual target converges to the desired speed profile chosen in the trajectory generation phase. The third term $-a\sum_{j\in\aleph_i}(\gamma_i(t) - \gamma_j(t))$ ensures that all the vehicles are coordinated with their neighbors as specified in (1.7). Finally, the fourth term $-\overline{\alpha}_i(e_{p,i}(t))$ is a correction term used to take into account for the path following errors of the agent. Indeed, if the vehicle is behind its target, the term is not zero and the target slows down in order to wait for the real vehicle.

With this control law, we want our vehicles to be synchronized and to proceed at a desired rate of progression, in order to accomplish the mission even when some unforeseen disturbances occur during the execution phase.

## 1.4 Network topology

To achieve time-coordination objective, agents must exchange information over a supporting communication network. To analyze the information flow, we need to consider some tools from algebraic graph theory, whose key concepts can be found in [1].

We assume that a vehicle $i$ exchanges information with only a subset of all vehicles, denoted as $\aleph_i(t)$. We assume that arcs of the network are bidirectional and that there are no network delays. The information exchanged is composed by the virtual time of the agents, $\gamma_i(t)$.

The topology of the graph $\Gamma(t)$ that represents the communication network must comply with the following constraint in order to guarantee the convergence of the "Consensus" algorithm:

$$\frac{1}{NT}\int_t^{t+T} QL(\tau)Q^T d\tau \geq \mu I_{N-1}, \quad for\ all \quad t \geq 0 \tag{1.11}$$

where $L(t) \in \mathbb{R}^{N\times N}$ is the Laplacian of the graph $\Gamma(t)$ and $Q \in \mathbb{R}^{(N-1)\times N}$ is a matrix such that $Q1_N = 0$ and $QQ^T = I_{N-1}$, with $1_N$ being a vector in $\mathbb{R}^N$ whose components are all 1. In (1.11), the parameters $T > 0$ and $\mu \in (0, 1]$ represent a measure of the level of connectivity of the communication graph. This condition requires the graph $\Gamma(t)$ to be connected only in an integral sense, not pointwise in time. Therefore, even if the graph were disconnected during the mission at some interval of time, the convergence of the "Consensus" algorithm would still be possible. With this condition, we can capture also packets dropouts, loss of communication and switching topologies, which can all occur during the mission, but these events do not necessary break the convergence property.

## 1.5 Convergence property

The control law given by (1.9) guarantees that the error of the "Consensus" algorithm converges to zero exponentially. It can be shown that the maximum convergence rate is given by the sum of the convergence rate of the path following error and the term

$$\frac{a}{b}\frac{N\mu}{T(1+(a/b)NT)^2} \tag{1.12}$$

which depends on the control gains $a$ and $b$, the number of vehicles $N$ and the quality of service of the communication network, characterized by the parameters

$T$ and $\mu$. If we fix the gains and the number of the vehicles, the convergence rate depends only on the amount of information which the agents exchange each other over time.

# Chapter 2

# Consensus

# Chapter 3

# System architecture

In this chapter we describe the hardware and software architecture of the system. Most of the software is completely decoupled by the hardware part and can be run on different machines, allowing portability and reusability. On the contrary, some code is developed only for the specific hardware, mostly the code more related with the specific functionalities of the hardware itself. We will have an overview of the hardware used and then we show a general software architecture and its deployment on the machines involved.

## 3.1 Hardware

The hardware used to run the system is heterogeneous and we will show in details the machines involved in the project.

First of all, we used a desktop pc as a ground station. It specifics are listed below:

- Processor: Intel Core(TM) i5-6500 CPU @ 3.20GHz
- Memory: 16 GB
- Network: Intel Gigabit CT Network Adapter

- Storage: 230 GB

On this machine, we run a virtual machine which has the following specifics:

- Processor: 1 Core
- Memory: 4 GB
- Storage 25 GB
- Network: Virtual adapter

The flying machines involved in the mission are of two kinds, but both adopt the same general configuration, even with different hardware. Indeed, they are equipped with a flight control unit connected with a companion microcomputer by the serial port The microcomputer communicates with the ground station using the Wifi.

### 3.1.1 Pixfalcon

The flight control unit adopted it the Pixfalcon (figure 3.1) which is from the family of -TODO-.


Figure 3.1: Pixfalcon board

Its specifications are the following:

- Main System-on-Chip: STM32F427

  - CPU: 180 MHz ARM Cortex M4 with single-precision FPU
  - RAM: 256 KB SRAM (L1)
- Failsafe System-on-Chip: STM32F100
  - CPU: 24 MHz ARM Cortex M3
  - RAM: 8 KB SRAM
- Wifi: ESP8266 external
- GPS: U-Blox 7/8 (Hobbyking) / U-Blox 6 (3D Robotics)
- Connectivity:
  - 1x I2C
  - 1x CAN (2x optional)
  - 1x ADC
  - 4x UART (2x with flow control)
  - 1x Console
  - 8x PWM with manual override
  - 6x PWM / GPIO / PWM input
  - S.BUS / PPM / Spektrum input
  - S.BUS output

### 3.1.2 Intel Edison

The companion computers are of two types. The first kind is the Intel Edison (figure 3.2)which is a general purpose computer.

The specifications are the following:

- Atom 2-Core (Silvermont) x86 @ 500 MHz

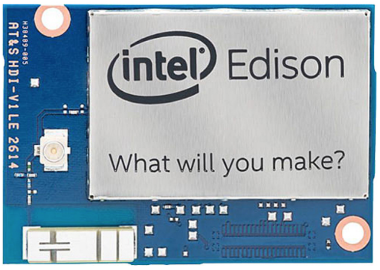


Figure 3.2: Edison board

- Memory: LPDDR3 1 GB
- Storage: 4 GB EMMC

### 3.1.3 RaspberryPi Zero

The second kind of companion is the RaspberryPi Zero (figure 3.3)

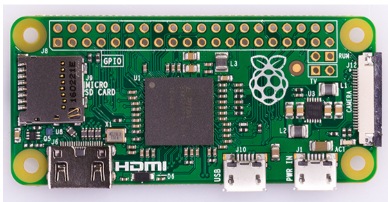


Figure 3.3: RaspberryPi Zero board

The specifications are the following:

- Processor:
  - Broadcom BCM2835
  - contains an ARM1176JZFS (ARM11 using an ARMv6-architecture core)
- Memory: 512MB LPDDR2 SDRAM
- USB On-The-Go port

- Mini HDMI
- 40pin GPIO header
- CSI camera connector (newer version from May 2016)

### 3.1.4 Motive Optitrack

The motion capture system is Motive Optitrack. We use eight Optitrack Prime 13 cameras arranged on a square as show in the figure.

The cameras are connected to a Netgear Prosafe 28PT GE POE switch using Gigabit Ethernet cables.

In order to be visible by the Optitrack system, every drone must be equipped with markers which make possible the recognition.

# Chapter 4

# Consensus node

In this chapter, we will examine the structure of the consensus node: we will see its main components, then, we will show some snippets of code in order to make clearer which parts are involved.

As shown in the general architecture, the consensus node is developed as a ROS node which subscribes and publishes messages to different topics. Moreover, the node offers some ROS services used to start and stop the trajectory following algorithm or the consensus one.

The structure of the node takes into account the main architectural patterns used in the software development field and it was designed to allow the maximum degree of usability and customization, even though one of the most important metric taken into account is the efficiency of the code, because it has to be executed on machines with limited amount of resources.

First of all, the node functionalities are enclosed into a C++ class which initializes all the ROS elements and prepares the node to receive the start and stop commands. The initialization is done by the class constructor when the object is created. In order to apply the consensus dynamic equation, we need the current position of the UAV. The Px4 board already publishes the estimated local position on a topic; so, we need to subscribe to that topic to retrieve the messages with the needed information. Second, we want to publish the consensus variable of the drone in the topic used by all the other UAVs, because, having obtained the others' consensus variables from the same topic, we are able to compute the proportional consensus error. Third, we need the next set point and the next desired velocity profile, because we want to compute the position error and weight it for the target velocity. At the end, we compute the acceleration of the consensus parameter using the consensus equation and we publish the next set point. We will see the details through the code. All these elements can be summarized and shown in figure 4.1.

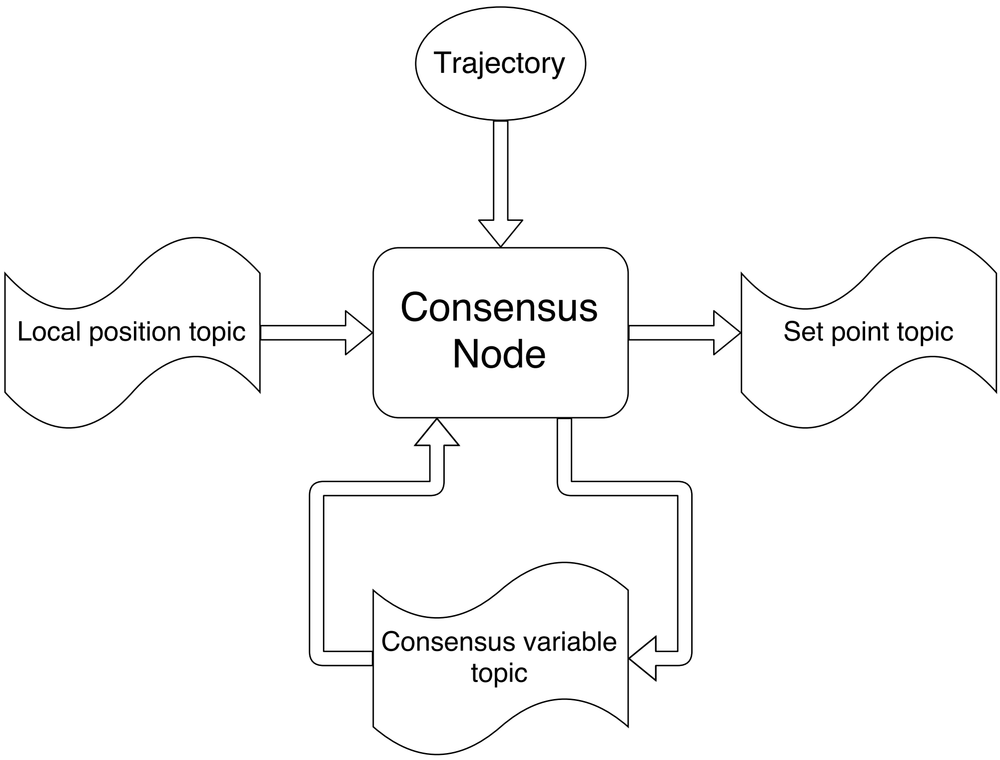


Figure 4.1: Input and output of the node

The subscription of a ROS topic works with a callback function, which accepts as parameter the pointer to the new message. Since in our case we have multiple subscriptions and we must advertise the start and stop services, we need to implement a multithreading architecture which takes care of the concurrent accesses to the state of our object. The number of thread used is three and these are their functions:

- Start and stop services
- Consensus variable callback
- Local position callback

## 4.1 Start and stop services

First of all, the node can work in two different modes:

- trajectory-only
- consensus

In the trajectory-only mode, the node computes the next set point and sends it to the UAV, without taking care if there are others UAVs in the mission. Its only objective is to follow the trajectory and reach its final position trying to respect the time constraints imposed by the trajectory. Instead, in the consensus mode, the node does the same computation as before, but it also publishes its consensus variable and reads all the other ones. It then considers this information and adjusts its variable. The consensus mode includes the trajectory-only, and can be started even if the trajectory-only is already started, while the opposite is not true. When you stop the consensus mode, also the trajectory-only is stopped. When the mission is accomplished, the current active mode is stopped automatically, so that you can freely restart one of the two modes without the need of stopping the previous one.

The services are implemented using the ROS Service class which manages the whole infrastructure needed for calling the service. The call to the service is synchronous and the caller is blocked until the service function is terminated.

In this case, we offer two services: one for starting and stoping the trajectory-only mode and the other for the consensus mode. It is possible to customize the service call in order to pass different number and types of arguments to the service function and the response can also be defined. For the two services we have defined the same parameters that are shown in the figure 4.2.

```
# Command for enabling or disabling
# trajectory and consensus tasks

#####################
## Request fields
bool cmd
---
#########################
### exit code constants
# everything ok
uint8 SUCCESS = 0
# trajectory following already
# active and consensus activated
uint8 ESCALATION = 1
# nothing to do
uint8 ALREADY_DONE = 2
# other errors
uint8 FAILURE = 3

######################
# Response fields
bool success
uint8 exit_code
```

Figure 4.2: Custom service structure

The message is composed by two parts: the request and the response. In the request we need a boolean field in order to know if we want to start or stop the algorithm. The response consists in a boolean variable which represents the success of the operation and an exit code which identifies the eventual problems occurred. The constants for the exit codes are directly specified in the definition of the service.

The trajectory-only service starts or stops the thread which, taken a local position, computes the next set point; while the consensus one starts or stops the same thread as before and the one which retrieves the consensus variables from the other quadrotors.

## 4.2 Consensus variable callback

The thread responsible for collecting the consensus variables of all the other UAVs, is managed by ROS and it executes a callback function when a new message is published on a specific topic. This topic is used by all the drones to publish their consensus variable, $\gamma_i$, and it accepts a custom message which contains only a string with the name of the owner of the variable and the value itself. The message has also a header, which contains general information such as timestamp or message id. The structure of the message can be seen in the figure 4.3.

```
Header header
string owner
float64 gamma
```

Figure 4.3: Custom message structure

The callback function receives the information from the topic and updates a local view of the variables of the neighbors. This information has a timeout validity, because we do not want to consider too old values. Indeed, if we consider too old values and a problem in the network causes a loss of packets, our drone might think which the other drones have a significantly different values of the consensus variables and so, wait for them. On the contrary, it is better to discard the values and remove the neighbors after a timeout.

In order to store the information, we use a support class, thread safe, which provides a procedure to check if the variable is expired or not. The signatures of the methods of the class are presented in the figure 4.4. We use a container to store the values of the neighbors and we always check if the value is expired or not before using it.

These variables are used in the consensus law as $\gamma_j$ of the neighbors and are used to compute the proportional error. We can see that the the expiration interval of the values can model the fact that the network topology can change. Indeed, if a link between two drones vanished because, for instance, they are too far from each other, after a timeout (which it is equal to the expiration time), the neighbor will be removed from the container and the drone will not take into account the old neighbor. Even a failure of a drone is ignored using the timeout: if a machine has a critical problem and does not send its consensus variable, the other drones remove it from their neighbors and, therefore, they can continue their mission without problems.

## 4.3 Local position callback

In the position callback function we apply the consensus law. First of all, we store the actual position of the drone in an object of a custom class, which signature is presented in figure 4.5. In this class, we also include, besides x, y and z, the yaw of our vehicle. Indeed, all the operations defined over the class consider also the orientation.

Now, we compute the synchronization term which corresponds to the sum of the difference between the current $\gamma_i$ and all the $\gamma_j$ of the neighbors. In order to do this, we iterate over a container, which stores the values, and we incrementally form the synchronization term.

The next step is to form the $\overline{\alpha}_i$ term. First of all, we need the next set point, which must be obtained evaluating the trajectory with the actual value of $\gamma_i$. Our trajectory is represented by a class, and its signature is shown in the figure 4.6. Now, we simply compute the *position_error* as *set_point* − *position*. We also need the desired velocity, which can be obtained using the suitable function of the trajectory class. At this point, we have all the terms needed to compute $\overline{\alpha}_i$ as defined in -TODO-.

Since we have all the elements, it is time to apply the consensus law and find $\ddot{\gamma}_i$. We simply need to have the coefficients $a$ and $b$, as shown in -TODO-, and the references $\ddot{\gamma}_d$ and $\dot{\gamma}_d$.

One of the last step needed is the update of $\dot{\gamma}_i$ using $\ddot{\gamma}_i$ and $\gamma_i$ using $\dot{\gamma}_i$. We compute the interval of time, $dt$, between the last update and the current update and we do the math as:

```
dgamma += ddgamma * dt;
gamma += dgamma * dt;
```

At the end, we need to publish the value of $\gamma_i$ to the right topic only if we are operating in consensus mode, otherwise we ignore it. An operation which is always needed is the publication of the set point message to the autopilot of the UAV, in order to allow it to follow the trajectory and reach its final destination.

```
class GammaParameter {
private:
  std::string owner;
  double gamma;
  double acquisition_time; //In seconds
  std::mutex mtx;
  static double expiration_interval; //In seconds

public:
  GammaParameter();
  GammaParameter(const GammaParameter& gp);
  GammaParameter(std::string owner,
                 double gamma,
                 double acquisition_time);
  ~GammaParameter();

  static void setExpirationInterval(double expiration_interval);
  static double getExpirationInterval();
  GammaParameter& setOwner(std::string owner);
  GammaParameter& setGamma(double gamma, double acquisition_time);
  GammaParameter& setData(std::string owner,
                          double gamma,
                          double acquisition_time);
  std::string getOwner();
  double getGamma();
  bool isExpired();
  GammaParameter& getData(std::string *owner_ptr,
                          double * gamma_ptr,
                          bool * exp_ptr);
  GammaParameter& operator= (const GammaParameter &gp);
};
```

Figure 4.4: Consensus variable class

```
class DronePose {
 private:
   double x, y, z, yaw;
 public:
   DronePose();
   DronePose(double x, double y, double z, double yaw);
   DronePose(const DronePose &dp);
   ~DronePose();

   double getX() const;
   double getY() const;
   double getZ() const;
   double getYaw() const;
   void setX(double x);
   void setY(double y);
   void setZ(double z);
   void setYaw(double yaw);

   double module() const;
   DronePose& operator= (const DronePose &dp); //Assignment
   DronePose operator+ (const DronePose &dp) const; //Sum
   DronePose& operator+= (const DronePose &dp);
   DronePose operator- (const DronePose &dp) const; //Difference
   DronePose& operator-= (const DronePose &dp);
   DronePose operator- () const; //Unary minus
   double operator* (const DronePose &dp) const; //Scalar product
   DronePose operator* (double scal) const; //Product with constant
   DronePose& operator*= (double scal);
   DronePose operator/ (double scal) const; //Ratio with constant
   DronePose& operator/= (double scal);
 };
```

Figure 4.5: Class used to manage the position and velocity of the drones

```
class Trajectory {
private:
  std::vector<TrajectorySegment *> segments;
  std::string drone_id;

  double filterAndConvertTime(double t) const;
  TrajectorySegment* getRightSegment (double t) const;

public:
  double min_time, max_time;

  bool loadXML(std::string dorne_ns, char * document);
  void cleanAll();

  DronePose evaluateNED(double t) const;
  DronePose evaluateENU(double t) const;
  DronePose operator[] (double t) const; //Default return ENU

  DronePose evaluateVelNED(double t) const;
  DronePose evaluateVelENU(double t) const;
};
```

Figure 4.6: Class used to manage a generic trajectory

# Chapter 5

# Simulation results

# Chapter 6

# Experimental results

# Conclusions

# Bibliography


[1] N. Biggs. *Algebraic Graph Theory.* New York: Cambridge Univ. Press, 1993.

[2] R. Choe, J. Puig-Navarro, V. Cichella, E. Xargay, and N. Hovakimyan. Trajectory generation using spatial Pythagorean Hodograph Bezier curves. *in Proc. AIAA Guidance, Navigation and Control Conf., Kissimmee, FL*, 2015.

[3] V. Cichella, R. Choe, S. B. Mehdi, E. Xargay, N. Hovakimyan, V. Dobrokhodov, I. Kaminer, A. M. Pascoal, and A. P. Aguiar. Safe coordinated manuvering of teams of multirotor unmanned aerial vehicles. *IEE Control systems magazine*, August 2016.

[4] V. T. Taranenko. Experience of Employment of Ritz's, Poincare's, and Lyapunov's Methods for Solving Flight Dynamics Problems. *Moscow: Air Force Engineering Academy Press*, 1968.

[5] O. A. Yakimenko. Direct method for rapid prototyping of near-optimal aircraft trajectories. *J. Guidance, Control Dyn.*, 23(5):865–875, Sept.-Oct. 2000.