ESCOLA NAVAL

talant de bien faire

Alexandre Valério Rodrigues

Implementação de um tradutor entre STANAG 4586 e MAVLink

Dissertação para a obtenção do grau de Mestre em Ciências Militares Navais, na especialidade de Engenharia Naval, Ramo de Armas e Eletrónica

Alfeite
2017

Alexandre Valério Rodrigues

Implementação de um tradutor entre STANAG 4586 e MAVLink

Dissertação para a obtenção do grau de Mestre em Ciências Militares Navais, na especialidade de Engenharia Naval, Ramo de Armas e Eletrónica

Orientação de: Mario Rui Monteiro Marques

O Aluno Mestrando	O Orientador
Alexandre Valério Rodrigues	Mario Rui Monteiro Marques

Alfeite
2017

"*Far and away the best prize that life has to offer is the chance to work hard at work worth doing*"

Theodore Roosevelt

AGRADECIMENTOS

Esta investigação contou com o apoio de várias pessoas que foram fundamentais para a sua realização. Assim, gostaria de expressar algumas palavras de agradecimento.

Ao meu orientador CTEN EN-AEL Monteiro Marques, pelos conselhos, direções e oportunidades que forneceu. A sua disponibilidade em qualquer altura foi uma constante ao longo de toda a investigação.

À empresa *UAVision*, por toda a disponibilidade.

Ao Professor Victor Lobo, diretor do Centro de Investigação Naval, por proporcionar os meios necessários para a realização desta investigação.

À minha família, mais concretamente aos meus pais, à minha irmã e à minha madrasta, por toda a compreensão durante alguns períodos de ausência que a realização desta dissertação obrigou, pelo apoio incondicional e por todos os ensinamentos e valores que foram transmitidos ao longo de toda a minha vida.

À Inês, por toda a disponibilidade e compreensão durante este período. Um especial agradecimento por toda a força e confiança transmitidas.

Por último mas não menos importante, aos camaradas do Curso D. Maria II, que me têm acompanhado ao longo destes cinco anos, sendo que a amizade e entreajuda demonstradas fizeram a diferença.

RESUMO

O papel dos *Unmanned Aerial Vehicles* (UAVs) tem vindo a ganhar destaque nos últimos anos. Hoje em dia, estes veículos podem ser utilizados nas mais variadas missões, tanto para fins militares como para fins civis. Os UAVs dependem de uma estação de controlo, responsável pelo envio de comandos e pela receção dos dados obtidos pelo veículo. Para existir comunicação entre o UAV e a estação de controlo, é necessária a utilização de protocolos de comunicação. Contudo, ao existirem múltiplos protocolos deste tipo, é verificada uma grande falta de padronização nesta área. A consequência deste problema é a inexistência de interoperabilidade entre sistemas que utilizam protocolos diferentes.

Esta dissertação visa responder a este problema através da criação de um tradutor entre dois dos protocolos de comunicação mais frequentemente utilizados: STANAG 4586 e MAVLink. O STANAG 4586 é um *standard* NATO e procura estabelecer uma uniformização para os UAVs dos países membros. O MAVLink, por sua vez, é um protocolo de comunicação bastante utilizado por vários pilotos automáticos, sendo assim uma referência na sua área. O desenvolvimento de um tradutor permitirá a interoperabilidade entre uma estação de controlo que utilize o *standard* STANAG 4586 e um UAV que utilize o protocolo MAVLink. Com a utilização do tradutor desenvolvido, não será necessário alterar a estrutura de um UAV para que este seja compatível com o *standard* STANAG 4586.

O sistema proposto é desenvolvido com a linguagem de programação *Python* e com a utilização de um *Raspberry Pi*, *hardware* de fácil inserção em qualquer veículo. Esta investigação conta ainda com a criação de uma biblioteca de mensagens STANAG 4586, que permite a validação e obtenção de resultados nos cenários criados. Desta forma, são então testados vários cenários, através da utilização do simulador *Software In the Loop* (SITL). Os resultados demonstram que o tradutor realiza a conversão entre os dois protocolos de forma viável, não afetando as comunicações entre o UAV e a estação de controlo. No final são apresentadas conclusões, assim como propostas de trabalho futuro.

Palavras-chave: *Unmanned Aerial Vehicle*, STANAG 4586, MAVLink, Tradutor.

ABSTRACT

The role of Unmanned Aerial Vehicles (UAVs) has been gaining prominence in recent years. Nowadays, these vehicles can be used in the most varied missions, both for military or civil purposes. The UAVs rely on a control station, responsible for sending commands and for receiving data obtained by the vehicle. In order to have communication between the UAV and the control station, communication protocols are required. However, since there are multiple protocols of this type, there is a great lack of standardization in this area. The consequence of this problem is the deficiency of interoperability between systems using different protocols.

This dissertation aims to address this problem by creating a translator between two widely used communication protocols: STANAG 4586 e MAVLink. STANAG 4586 is a NATO standard and seeks to establish a standardization for UAVs country members. MAVLink, however, is a communication protocol broadly used by several autopilots, which makes it a reference in its area. The development of a translator will allow interoperability between a control station using the STANAG 4586 standard and a UAV using the MAVLink protocol. By using the developed translator, it will not be necessary to change the structure of the UAV to make it compatible with STANAG 4586.

The proposed system is developed with Python programming language and with the use of a Raspberry Pi, which is easy to insert into any vehicle. In this research, a STANAG 4586 message library is also created, which allows to validate and obtain results from the created scenarios. Accordingly, several scenarios are then tested through the use of the Software In the Loop simulator (SITL). The results prove that the translator performs the conversion between the two protocols in a viable way, without affecting the communications between the UAV and the control station. Conclusions are presented at the end as well as future work proposals.

Keywords: *Unmanned Aerial Vehicle*, STANAG 4586, MAVLink, Translator.

ÍNDICE GERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÓNIMOS

AESA	Agência Europeia para a Segurança da Aviação
APA	*American Psychological Association*
API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
BSD	*Berkeley Software Distribution*
C4I	*Command, Control, Communications, Computers and Intelligence*
CBRN	*Chemical, Biological, Radiological and Nuclear*
CCI	*Command Control Interface*
CCISM	*Command and Control Interface Specific Module*
CDT	*Control Data Terminal*
CINAV	Centro de Investigação Naval
CRC	*Cyclic Redundancy Check*
DLI	*Data Link Interface*
EHF	*Extremely High Frequency*
EUA	Estados Unidos da América
GCS	*Ground Control Station*
GT-VENT	Grupo de Trabalho para os Veículos Não Tripulados
GUI	*Graphical User Interface*
HCI	*Human Computer Interface*

HF *High Frequency*

ICARUS *Integrated Components for Assisted Rescue and Unmanned Search Operations*

ID Identificação

IEEE *Institute of Electrical and Electronics Engineers*

JAUS *Joint Architecture for Unmanned System*

LF *Low Frequency*

LGPL *GNU Lesser General Public License*

MAVLink *Micro Aerial Vehicle Communication Protocol*

MF *Medium Frequency*

NATO *North Atlantic Treaty Organization*

NSA NATO *Standardization Agency*

OSI *Open System Interconnection*

PR *Personal Robots*

RAM *Random Access Memory*

REX *Robotics Exercise*

ROS *Robot Operative System*

SANT Sistema Aéreo Não Tripulado

SESAR *Single European Sky ATM Research*

STANAG NATO *Standardization Agreements*

SHF *Super High Frequency*

SITL	*Software In The Loop*
STAIR	*Stanford Al Robot*
UAS	*Unmanned Aircraft System*
UAV	*Unmanned Aerial Vehicles*
UCS	UAV *Control System*
UDP	*User Datagram Protocol*
UHF	*Ultra High Frequency*
USB	*Universal Serial Bus*
VDT	*Vehicle Data Terminal*
VHF	*Very High Frequency*
VLF	*Very Low Frequency*
VSM	*Vehicle Specific Module*
XML	*eXtensible Markup Language*

Introdução

Motivação

Objetivo

Metodologia

Estrutura da Investigação

Os sistemas não tripulados são sistemas compostas por veículos, sejam eles aéreos, marítimos ou terrestres, que não possuem um operador humano a bordo. Estes veículos são controlados remotamente e podem ter vários tipos de carga, dependendo dos objetivos da missão a que são sujeitos. Os veículos que operam no meio aéreo são normalmente designados *Unmanned Aerial Vehicles* (UAVs).

São várias as tarefas que estes veículos podem desempenhar, oferecendo grandes vantagens com a sua utilização. Reconhecimento aéreo, patrulhas e inspeções em ambientes complexos e perigosos são algumas dessas tarefas. Um dos pontos fortes na utilização dos UAVs consiste no facto de não existir risco para a vida humana, sendo que no caso de se perderem, apenas existem prejuízos materiais. As elevadas taxas de sucesso já demonstradas em cenários reais, a capacidade de operarem em meios difíceis e os grandes avanços da tecnologia são alguns dos fatores que motivam significativamente a expansão destes sistemas. Deste modo, as imensas capacidades dos UAVs permitem que estes veículos sejam utilizados tanto no meio civil, como no militar (Nonami, 2010).

Atualmente, os UAVs têm tido cada vez mais impacto no meio civil. Nos últimos anos estimou-se que estes sistemas tenham obtido um crescimento anual de aproximadamente 100% (SESAR, 2016). As suas aplicações são várias, tais como monitorização e inspeção de edifícios, mapeamento do território, monitorização do ambiente, filmagens para fins de jornalismo, entre outros. Existem vários tipos de UAVs, sendo que diferem nas suas dimensões, desempenho e objetivos. A Tabela 1 representa uma classificação feita para UAVs que operam no meio civil e está apresentada num estudo criado pelo Parlamento Europeu (Marzocchi, 2015).

Tabela 1 - Classificação de UAVs no meio civil

Tipo de UAV e preços praticados	Aplicações	Legislação
Micro (<20/25 kg)	Lazer e uso comercial. Ex: fotografia e inspeções.	Regulado de acordo com cada país.
Pequeno/Médio (20/25 – 150kg)	Patrulhas em grandes áreas, busca e salvamento, controlo de fronteiras.	Regulado de acordo com cada país.
Grande (>150kg)	Normalmente no meio militar, pode carregar grandes cargas e até passageiros.	UAVs civis seguem a legislação da Agência Europeia para a Segurança da Aviação.

No entanto, a utilização de UAVs começou no meio militar e foi aqui que estes veículos começaram a ganhar destaque. São várias as vantagens que tornam estes veículos cada vez mais importantes nas Forças Armadas. Estima-se que existam aproximadamente 1000 UAVs entre todas as Forças Armadas Europeias. Adicionalmente, é ainda estimado que exista um crescimento anual de sensivelmente 5% no número de UAVs inseridos neste meio. Como consequência, é importante referir que com o aumento do número de UAVs é expectável uma diminuição na aviação tripulada (SESAR, 2016).

Deste modo, a introdução deste tipo de sistemas na Marinha Portuguesa é iminente e é seguro afirmar que no futuro estes sistemas farão parte dos seus meios operacionais. Participar em missões de busca e salvamento ou fazer reconhecimento de áreas perigosas são algumas das tarefas que os UAVs poderão vir a desempenhar. A utilização destes sistemas poderá vir a reduzir os custos das missões, assim como o risco associado.

O Grupo de Trabalho para os Veículos Não Tripulados (GT-VENT) foi criado em 2015. O objetivo deste grupo é assegurar todo o enquadramento relativamente à utilização de veículos não tripulados na Marinha Portuguesa. A definição do enquadramento estratégico relativamente à utilização de UAVs ou a definição de linhas orientadoras para a sua operação, são algumas das incumbências que o GT-VENT possui.

A Marinha Portuguesa promove ainda vários exercícios com veículos autónomos,

através do Centro de Investigação Naval (CINAV). Um deles é o *Robotics Exercise* (REX), que visa dar oportunidade à comunidade empresarial e académica para demonstrar e testar os seus sistemas no meio marítimo. Este exercício constitui também uma boa oportunidade para a Marinha Portuguesa estar em contacto com os mais recentes avanços na área da robótica móvel.

De forma a estar próxima do desenvolvimento e investigação dos veículos autónomos, a Marinha Portuguesa assinou protocolos com as empresas *UAVision* e *Tekever*[1]. Deste modo, é possível verificar que a Marinha Portuguesa, mais especificamente o CINAV, tem uma ligação direta neste âmbito de investigação.

Como referido anteriormente, os UAVs são veículos aéreos não tripulados, mas para operarem necessitam de um sistema mais complexo, onde estão englobados elementos tais como a *Ground Control Station* (GCS) ou estação de controlo, o *data link*, entre outros. A GCS é responsável por monitorizar e controlar o veículo, transmitindo os comandos necessários para realizar esta tarefa. É ainda na GCS que se recebem os dados provenientes do UAV, tais como imagens, vídeo, ou outros dados que se pretenda adquirir. A função do *data link* é estabelecer a comunicação entre o UAV e a GCS (Nonami, 2010).

Para estabelecer o *data link* é necessário estabelecer protocolos de comunicação. Estes protocolos são normalmente baseados na troca de mensagens formatadas, entre o veículo e a estação de controlo. No entanto, uma consequência da grande expansão dos veículos autónomos é a criação de vários protocolos e arquiteturas diferentes, o que leva a uma falta de padronização. Alguns destes protocolos de comunicação são o *Micro Aerial Vehicle Communication Protocol* (MAVLink), o *Joint Architecture for Unmanned Systems* (JAUS) ou o *Standardization Agreement* (STANAG) 4586.

Um dos protocolos de comunicação entre veículos autónomos mais utilizadas no meio militar é o STANAG 4586. Contudo, a maioria dos sistemas existentes no mercado nacional não é compatível com este protocolo. No entanto, um dos protocolos mais comuns no meio civil é o MAVLink, uma vez que são vários os pilotos automáticos que

[1] Empresas aeronáuticas responsáveis pela criação e exportação de vários veículos aéreos não tripulados.

o utilizam.

O STANAG 4586 é um *standard*[2] desenvolvido pela *North Atlantic Treaty Organization* (NATO), para a comunicação entre uma estação de controlo e um UAV. O objetivo é garantir a interoperabilidade entre este tipo de sistemas autónomos, de modo a aumentar as capacidades de forças conjuntas aquando da operação com UAVs entre os países membros. Este *standard* inclui a arquitetura do *data link*, arquiteturas para os sistemas de controlo e interface com os operadores e ainda uma biblioteca onde são definidos os formatos de várias mensagens.

O MAVLink é um protocolo que tem um conjunto de bibliotecas específicas para veículos aéreos autónomos de pequenas dimensões. Este protocolo é composto por um conjunto de mensagens pré-definidas, em que cada uma tem a sua função específica.

Considerando que é um *standard* NATO, utilizado em ambientes militares, não é fácil encontrar sistemas que sejam compatíveis com o STANAG 4586. Uma das causas deste fator é a grande complexidade que este tipo de missões possui, o que exige que este *standard* seja bastante completo, dotado de doutrina militar. Por outro lado, o MAVLink é utilizado para ter comunicações simples e rápidas entre UAVs de pequenas dimensões e não foi desenvolvido para ser empregue num meio militar.

A falta de padronização leva a que sistemas que utilizem diferentes protocolos de comunicação não possam trocar informação entre si, o que é essencial em operações com vários veículos autónomos. A interoperabilidade é um dos pilares de qualquer missão conjunta. Esta capacidade é bastante importante, especialmente na NATO, visto que é uma organização onde vários países operam em conjunto (NATO, 2006).

De forma a tirar o máximo proveito sobre os veículos autónomos, é necessário garantir a interoperabilidade entre os mesmos. No entanto, a existência de diversos protocolos de comunicação entre estes sistemas torna esta tarefa difícil, visto que uma GCS que utilize um certo protocolo não pode comunicar com um veículo que utilize outro

[2] Por ser mais abrangente, o STANAG 4586 não é apenas visto como um protocolo de comunicação, mas como um *standard*. Assim, neste caso, o termo *standard* engloba não só formatos de mensagens, como também a arquitetura do sistema aéreo não tripulado e interfaces entre o operador e o veículo aéreo.

diferente. Criar um sistema que faça a conversão entre diferentes protocolos de comunicação é uma possível solução para este problema.

Motivação

Os veículos autónomos fazem parte de uma área que está em constante desenvolvimento. Devido às suas inúmeras vantagens, a introdução dos UAVs na Marinha Portuguesa é cada vez mais uma realidade e é esperado que no futuro estes sistemas sejam introduzidos nos meios operacionais atualmente existentes. Para tal, é necessário que estas tenham compatibilidade com os sistemas atualmente utilizados pelas Forças Armadas dos vários países que integram a NATO. A possibilidade de trabalhar nesta área, contribuindo para a utilização dos UAVs na Marinha Portuguesa é uma grande motivação pessoal para a realização desta dissertação de mestrado.

Adicionalmente, o desenvolvimento desta dissertação proporcionará uma oportunidade de aprender vários conceitos e métodos novos. Desta forma, será possível aprender a trabalhar em ambientes Linux, assim como aprender a programar numa linguagem não familiar, o *Python*[3]. Estes fatores tornam esta dissertação de mestrado mais desafiante, aumentando o nível de motivação.

Objetivo

A falta de padronização na área da robótica móvel é uma das principais preocupações atuais dos investigadores. O objetivo desta dissertação é a realização de um sistema que faça a tradução entre dois protocolos de comunicação existentes: STANAG 4586 e MAVLink. Este sistema deverá permitir que o envio de mensagens com o formato especificado pelo *standard* STANAG 4586 seja compatível com o formato das mensagens do protocolo MAVLink. Com a utilização do tradutor desenvolvido, não será necessário alterar a estrutura de um UAV para que este fique compatível com o *standard* STANAG 4586. Esta investigação assume maior relevo uma vez que a importância do *standard* STANAG 4586 tem vindo a aumentar, tornando-se assim num documento

[3] Python é uma linguagem de programação de alto nível, desenvolvida em 1991. Podendo ser utilizado em vários sistemas operativos, é uma linguagem simples e com várias bibliotecas disponíveis, tornando-se uma grande vantagem.

padrão entre os países membros da NATO.

Criar um tradutor entre dois protocolos de comunicação engloba vários passos e requisitos que devem ser cumpridos. Para tal, é necessário estabelecer quais as mensagens que devem ser traduzidas de modo a que o veículo desempenhe a função pretendida. Simultaneamente, é necessário que o conversor seja bem estruturado, possuindo o menor tempo de resposta possível de modo a não afetar as comunicações.

Metodologia

A presente dissertação foi proposta pela empresa *UAVision* e visa avaliar a viabilidade no desenvolvimento de um tradutor entre o STANAG 4586 e o MAVLink. O modelo procurará colmatar a falta de interoperabilidade existente atualmente, permitindo a comunicação e cooperação entre veículos MAVLink e estações de controlo STANAG 4586.

Para adquirir conhecimentos para a realização desta investigação, foi realizada uma revisão da literatura, onde foram pesquisados vários livros de referência na área da robótica móvel. Adicionalmente, foram também pesquisados vários artigos científicos, de modo a fundamentar a dissertação. O estilo bibliográfico escolhido para ser utilizado ao longo desta dissertação é o *American Psychological Association* (APA).

A implementação deste tradutor será feita através de um *hardware Rapsberry Pi*. Este *hardware* é um computador, todo integrado numa só placa, capaz de fazer inúmeras tarefas que um computador normal faz. Este dispositivo utiliza um sistema operativo Unix e a sua programação é feita em *Python*.

Estrutura da dissertação

Esta dissertação de mestrado é composta por introdução, quatro capítulos e conclusão. Ao longo dos quatro capítulos será feito um enquadramento teórico sobre o tema, dando a conhecer o problema, metodologias de investigação, um modelo de resolução e, por fim, validação e discussão de resultados.

A introdução apresenta os veículos autónomos e como estes são vantajosos não só no meio civil, como também no meio militar. O problema associado à falta de padronização entre estes veículos é igualmente identificado, clarificando o objetivo desta dissertação de mestrado.

O primeiro capítulo representa o enquadramento teórico sobre o tema. Desta forma, começa por identificar o conceito de interoperabilidade sob diferentes perspetivas. Posto isto, é representado o Sistema Aéreo Não Tripulado (SANT), onde é apresentada a evolução histórica destes sistemas, algumas arquiteturas utilizadas e também alguns exemplos da sua aplicação. Finalmente, são introduzidos protocolos de comunicação, entre os quais o MAVLink, STANAG 4586 e o ROS, que irão ser utilizados nesta investigação, expondo ainda determinados trabalhos realizados na área.

As metodologias de investigação são abordadas ao longo do segundo capítulo. No desenvolvimento de uma dissertação, é fundamental a escolha do método de investigação adequado à área científica em questão. Assim, serão introduzidos alguns conceitos sobre o método de investigação clássico, assim como uma análise sobre diferentes tipos de metodologias de criação de *software*.

O processo de desenvolvimento e implementação do modelo é apresentado no terceiro capítulo. Inicialmente, é explicado o processo de escolha sobre a arquitetura que o sistema deve possuir, de modo a descobrir uma solução para o problema encontrado. Posteriormente, é realizado um estudo sobre a escolha do *hardware* a utilizar. Por fim, são definidos os passos para a realização do *software* que faz a tradução entre o STANAG 4586 e o MAVLink.

O quarto capítulo introduz a validação e discussão de resultados. Esta etapa é fundamental para o estudo da viabilidade, uma vez que demonstra se o problema foi solucionado através do uso do tradutor concebido. Deste modo, serão definidos inicialmente os cenários de validação, incluindo os seus objetivos e requisitos. Posteriormente, serão analisados os dados obtidos, de modo a perceber se o tradutor afeta ou não as comunicações.

No final da dissertação serão apresentadas as conclusões, onde será feito um

resumo dos principais pontos abordados ao longo da dissertação de mestrado e serão apresentadas as últimas reflexões sobre o tema. Como nota final, serão propostos possíveis trabalhos futuros.

1. Revisão da Literatura

1.1 Interoperabilidade

1.2 Sistema Aéreo Não Tripulado

1.3 STANAG 4586

1.4 MAVLink

1.5 ROS

1.6 Trabalhos anteriores

O primeiro capítulo desta investigação consiste na revisão da literatura. Primeiramente, é definido o conceito de interoperabilidade, exemplificando-o em diferentes áreas e explicando a sua importância. De seguida, é representado o sistema aéreo não tripulado, incluindo uma breve introdução histórica, assim como missões que estes veículos podem desempenhar e a arquitetura geral do sistema. Posto isto, são representados e explicados os protocolos de comunicação entre UAVs e estações de controlo que serão utilizados ao longo desta investigação. Finalmente, é apresentado um estudo contendo trabalhos anteriores que foram realizados nesta área.

1.1 Interoperabilidade

A interoperabilidade é um conceito amplo, podendo ter várias definições de acordo com a respetiva área de trabalho. De acordo com o *Institute of Electrical and Electronics Engineers* (IEEE), a interoperabilidade pode ser definida como a capacidade que dois ou mais sistemas têm para trocarem informação entre si e cooperarem com sucesso (IEEE, 1990). Noutra perspetiva, para um ambiente de operações conjuntas, a NATO define interoperabilidade como a capacidade que diferentes organizações militares possuem para conduzir operações em conjunto. Estas operações podem ser realizadas com organizações de diferentes países e em ambientes distintos, tais como terrestres, navais ou aéreos. Nesta definição, a interoperabilidade permite que diferentes forças, unidades ou sistemas operem em conjunto, aumentando as suas capacidades e possibilitando, por vezes, a redução de custos. Para isso é necessário que seja partilhado doutrinas e procedimentos comuns através da utilização de *standards* (NATO, 2006).

De acordo com o IEEE, os *standards* são documentos que contêm conjuntos de requerimentos, boas práticas e regras, criados de modo a existir um procedimento disciplinado e uniforme para o que se pretende atingir (IEEE, 1990). A utilização de *standards* é um fator fundamental para que a cooperação de vários sistemas ou forças seja realizada com sucesso. A NATO *Standardization Agency* (NSA) é uma organização que faz parte da NATO e que tem como objetivo a elaboração da política de *standards*. Esta política é alcançada através da criação de STANAGs, que visam fornecer às diferentes nações doutrinas, procedimentos e conjuntos de regras de modo a que alcancem o nível de interoperabilidade desejado (NATO, 2006).

Desta forma, foi criado um *standard*, STANAG 4586 com o propósito de promover a interoperabilidade entre os UAVs que operam num ambiente de operações conjuntas NATO. A conceção deste *standard* aumentou significativamente a capacidade de operação e cooperação dos UAVs, permitindo o aumento da probabilidade de sucesso das missões. A arquitetura funcional da estação de controlo, a arquitetura do *data link*, interfaces para o operador ou mensagens que devem ser trocadas entre sistemas são algumas das especificações que este *standard* define (NATO Standardization Agency, 2012). O STANAG 4586 é um *standard* de referência para sistemas aéreos não tripulados utilizados por nações NATO.

1.2 Sistema Aéreo Não Tripulado

Antes de se iniciar o estudo sobre UAVs é necessário clarificar alguns conceitos chave. A terminologia relacionada com estes sistemas tem variado ao longo do tempo, sendo que, por vezes, o mesmo sistema tem diferentes termos associados. Este facto deve-se principalmente à existência de diferentes requisitos e conceitos de utilização, como por exemplo, entre sistemas militares e civis. Deste modo, um UAV pode ser definido como um veículo aéreo não tripulado, ou seja, um veículo que não tem um piloto humano nem passageiros a bordo (K. Valavanis, 2015). A Agência Europeia para a Segurança da Aviação (AESA) define também o termo SANT, ou *Unmanned Aircraft System* (UAS), como sendo um conjunto de elementos, incluindo um veículo não tripulado, uma estação de controlo e quaisquer outros elementos necessários para permitir o voo, tais como elementos que auxiliem no lançamento e recolha do veículo (European Aviation Safety Agency, 2009).

Desta forma, este subcapítulo está organizado de modo a fornecer primeiramente uma breve história sobre a utilização dos UAVs. De seguida, são representadas missões que os UAVs podem desempenhar, quer no meio civil ou militar. Finalmente, é representada a arquitetura geral que cada sistema deste tipo emprega, identificando os seus elementos principais.

1.2.1 História dos UAVs

A história da utilização de UAVs não é recente, sendo que o seu grande desenvolvimento começou a partir da Primeira Guerra Mundial. No entanto, o aparecimento deste tipo de veículos deu-se há mais de 2500 anos.

O primeiro mecanismo utilizado como máquina voadora autónoma de que há registo foi criado por um grego chamado Archytas. Aplicando uma série de conhecimentos sobre geometria, mecânica e estruturas, bastante evoluídos na época, Archytas criou *O Pombo*, em 425 a.C.. O mecanismo, que está representado na Figura 1, foi produzido em madeira e com a sua forma de ave, voava movendo as asas através de energia gerada pelo vapor produzido por uma caldeira aquecida. Os registos indicam que este veículo voou cerca de 200 metros antes de perder a energia, sendo que se despenhou, não voltando a voar. Esta invenção é considerada por alguns autores como sendo o primeiro UAV da história (Valavanis, 2007).

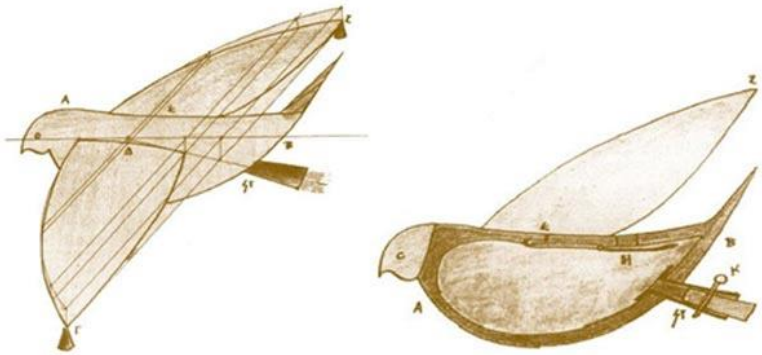

Figura 1- O Pombo, criado por Archytas. Fonte:http://www.ancient-origins.net/ancient-technology

Desde o aparecimento de *O Pombo*, outros mecanismos deste tipo foram desenhados e criados, sendo que a sua complexidade foi aumentando. Alguns dos veículos mais importantes e conhecidos estão representados na Tabela 2, que representa a evolução histórica dos UAVs. Os primeiros sistemas concebidos foram meros protótipos, enquanto que os veículos apresentados no final do século XX já apresentavam semelhanças com os existentes atualmente.

Tabela 2- História dos UAVs

Data e Designação	Descrição
425 a.C. – *O Pombo*	Veículo em forma de ave, movido através de energia gerada por vapor de uma caldeira.
400 a.C. – *Bamboo Copter*	Voo vertical com rotores construídos com bambu.
1483 – *Aerial screw*	Veículo com 5 metros de diâmetro construído com tecido e madeira, capaz de pairar.
1508 – *The Glider*	Veículo em forma de ave, que movia as asas para voar.
1916 – *Hewitt-Sperry Automatic Airplane*	Desenvolvido para ser uma bomba voadora.
1918 – *Kettering Bugg*	Bomba voadora desenhada para atingir alvos a 121km de distância.
1933 – *Queen Bee*	Aeronave não tripulada construída para ajudar nos treinos de artilharia.
1939 – *Radioplane OQ-2*	Primeiro UAV produzido em grandes quantidades nos Estados Unidos da América (EUA).
1944 – *Buzzbomb*	Primeira bomba voadora de sucesso, empregue e construída em grande escala.
1955 - *Falconer*	UAV utilizado para reconhecimento, equipado com uma câmara.
1959 – *QH-50 DASH*	Helicóptero não pilotado, criado para lançar torpedos anti-submarinos.
1964 – *Lightning Bug*	UAV utilizado para reconhecimento, com elevada taxa de sucesso.
1994 – *Predator*	Inicialmente concebido para reconhecimento aéreo, posteriormente atualizado para lançar mísseis.

O aparecimento dos UAVs no meio militar deu-se durante a Primeira Guerra Mundial com as bombas voadoras produzidas nos EUA. A primeira bomba voadora foi denominada *Hewitt-Sperry Automatic Airplane* e os seus testes começaram em 1917. Contudo, o projeto, que foi requerido pela Marinha dos EUA, foi cancelado, visto não ter obtido resultados positivos. Apesar disso, o desejo de ter uma bomba voadora manteve-se e, em 1918, foram efetuados testes com a *Kettering Bug*. Este mecanismo foi apenas testado, não sendo utilizado em ambiente real (Cook, 2007).

Foi durante a Segunda Guerra Mundial que apareceu a primeira bomba voadora de sucesso. Desenvolvida pela Força Aérea Alemã, a *Buzzbomb* foi responsável por mais de 6200 baixas no Reino Unido. O *Lightning Bug* foi outro UAV empregue com uma elevada taxa de sucesso. Foi bastante utilizado pela Força Aérea dos EUA na guerra do Vietname e foi concebido principalmente para realizar missões de reconhecimento.

O *Predator* foi desenvolvido para a Força Aérea dos EUA e começou a ser utilizado na década de 90. Inicialmente concebido para efetuar missões de reconhecimento, foi posteriormente modificado de modo a ser equipado com mísseis *Hellfire.* Este UAV continua a ser utilizado atualmente, apesar de já ter sofrido atualizações desde o seu lançamento.

A evolução dos UAVs tem sido notória e estes veículos estão a ser cada vez mais utilizados, quer para propósitos militares ou civis. No início do século XX, os testes realizados com UAVs apenas duravam no máximo 20 minutos. Cerca de 50 anos depois, estes sistemas já eram utilizados com sucesso para missões de reconhecimento. Deste modo, a complexidade e capacidade associada a estes sistemas tem vindo a aumentar significativamente. Atualmente, estes sistemas são cada vez mais importantes nas operações militares e têm vindo a ser integrados nas Forças Armadas de vários países.

1.2.2 Missões

Os UAVs podem desempenhar um conjunto de missões bastante vasto, uma vez que são sistemas bastante dinâmicos, podendo ter arquiteturas diferentes consoante a sua função. Desta forma, existem diversas características que fazem destes veículos opções viáveis para diferentes tipos de missões. Algumas das características mais importantes são mencionadas de seguida (Skrzypietz, 2012):

- Resistência, uma vez que existem UAVs que podem operar durante dias, não dependendo do descanso dos pilotos;
- Segurança que estes veículos proporcionam, uma vez que o piloto está localizado na estação de controlo;

- Flexibilidade na sua operação, visto que devido às suas dimensões e características aerodinâmicas estes veículos têm uma grande capacidade de manobra.

Assim, os UAVs podem desempenhar vários tipos de missões, que podem ser divididas entre missões de âmbito civil ou militar. De seguida estão mencionadas algumas missões de âmbito civil que estes veículos podem realizar (Skrzypietz, 2012) (Austin, 2010):

- Explorações científicas, tais como explorações geológicas, estudo de vulcões, estudo de furacões, estudo da atmosfera, entre outros;
- Monitorização de pescas ilegais;
- Monitorização de fontes emissoras de poluição;
- Observação de campos petrolíferos;
- Prevenção e monitorização de incêndios florestais;
- Monitorização de cheias;
- Avaliação de danos causados por desastres naturais, como por exemplo terramotos;
- Avaliação de áreas contaminadas;
- Procura por sobreviventes em incidentes, tais como naufrágios, quedas de aviões, entre outros;
- Proteção e monitorização das fronteiras;
- Vigilância da costa marítima;
- Monitorização e segurança de multidões em grandes eventos;
- Inspeção a infraestruturas;
- Recolha de dados para serviços meteorológicos;
- Fotografias e filmagens, como por exemplo para emissões desportivas ou filmes;
- Transporte de cargas;
- Agricultura, sendo que pode ser utilizado para rega, colocação de fertilizantes ou até monitorização através de imagem ou sensores específicos.

Como referido anteriormente, as características dos UAVs tornam-nos excelentes opções para a realização de certas missões de âmbito militar, tais como (Alkire, 2010) (Austin, 2010) (David, 2000) (Coffey, 2002):

- Informação e reconhecimento, bastante utilizado nas forças especiais, na recolha e análise de dados, proteção e disseminação de informação sobre o inimigo;
- Transporte de cargas em locais de difícil acesso;
- Segurança marítima, como por exemplo na proteção de portos, bases, navios ou outras infraestruturas;
- Guerra eletrónica passiva ou ativa, visto que estes veículos podem estar equipados com sensores e transmissores de modo a efetuar *jamming* ou deceção eletromagnética;
- Guerra aérea, na aquisição, designação e ataques a alvos;
- Guerra antissubmarina, com a colocação de sonoboias;
- Comunicações, funcionando como um retransmissor, aumentando o seu alcance e providenciando comunicações para forças;
- Monitorização de contaminação *Chemical, Biological, Radiological and Nuclear* (CBRN);
- Busca e salvamento, em situações de perigo;
- Combate à pirataria, sendo utilizados na monitorização e vigilância;
- Avaliação de danos causados.

1.2.3 Arquitetura do Sistema

Existem várias conceções quanto à arquitetura de um sistema aéreo não tripulado ou UAS. Deste modo, alguns autores dividem o sistema da seguinte forma: veículo aéreo (UAV), estação de controlo (GCS) e ligação de dados (Gupta, 2013). Por outro lado, existem classificações onde os elementos são mais especificados, tais como: veículo aéreo (UAV), estação de controlo (GCS), equipamento de lançamento e recolha do veículo, carga útil e ligação de dados (Austin, 2010) (Fahlstrom, 2012)(NATO Standardization Agency, 2012). As duas classificações expostas abrangem a mesma informação, dividindo, no entanto, o sistema em partes diferentes.

A arquitetura do sistema abordada nesta dissertação será dividida do seguinte modo: veículo aéreo, estação de controlo e ligação de dados. Esta escolha deve-se ao facto de não ser necessária uma abordagem detalhada em relação aos elementos da carga útil e do lançamento e recolha do veículo, por não ser esse o âmbito de investigação desta dissertação.

1.2.3.1 Veículo Aéreo

No veículo aéreo está compreendida a fuselagem, o sistema de propulsão, o sistema de navegação, o sistema de produção e distribuição de energia e carga útil. Na carga útil estão contidos os elementos que não são necessários para o voo, controlo e navegação, mas que são requisitos necessários para que o UAV cumpra a missão. Estes elementos podem ser diversos, tais como câmaras eletro-óticas, câmaras de imagem térmica, radares, sistemas de armas, entre outros.

Existem vários tipos de veículos aéreos, dependendo das suas características aerodinâmicas e configurações, podendo ser classificados em quatro categorias: asa fixa, asa rotativa, dirigíveis (Figura 2 a)), e *flapping wings* (Figura 2 b)). Contudo, os UAVs mais utilizados são os de asa fixa e asa rotativa, visto que apresentam um conjunto de características mais adequadas para a maioria das aplicações (Nonami et al., 2010).

Figura 2- a) UAV Dirigível (Boon, 2004); b) Flapping Wing UAV (Austin, 2010)

Veículo Aéreo de Asa Fixa

Os UAVs de asa fixa são bastante utilizados em aplicações que necessitem de grande autonomia. Estas plataformas são geralmente caracterizadas por conseguirem percorrer grandes distâncias, mantendo-se a voar durante largos períodos de tempo, podendo estes chegar até a alguns dias. Apresentam ainda outras vantagens, tais como o facto de poderem carregar cargas úteis pesadas, conseguirem atingir velocidades elevadas e de possuírem estruturas mais simples, em comparação com as aeronaves de asa rotativa (Austin, 2010). A Figura 3 representa o *Wingo*, um UAV de asa fixa desenvolvido pela empresa *UAVision.*

Figura 3 – Wingo. Fonte: UAVision

Uma das desvantagens que a maioria das aeronaves de asa fixa apresenta é o facto de necessitar de uma pista de lançamento ou de outro tipo de materiais que auxiliem neste processo, tais como lançadores hidráulicos ou o lançamento assistido por rocket (Eriksson, 2013). No entanto, os veículos mais leves podem não precisar deste tipo de lançamento, levantando voo com uma projeção manual. Outra desvantagem é o facto de este tipo de aeronaves não conseguirem pairar, visto que precisam constantemente de ar a passar sobre as suas asas de modo a poderem voar. Esta desvantagem faz com que este tipo de veículos não seja a melhor solução para aplicações estacionárias, tais como realizar inspeções em edifícios, ou manter o contacto visual sobre um determinado alvo durante um período de tempo alargado.

Veículo Aéreo de Asa Rotativa

As principais características dos veículos aéreos de asa rotativa são a capacidade de levantamento de voo vertical e a sua grande manobrabilidade. Adicionalmente, têm ainda a capacidade de pairar, o que não acontece nos veículos de asa fixa. Estas

características fazem com que este tipo de UAVs seja bastante utilizado em aplicações civis, devido aos seus fáceis processos de levantamento de voo e aterragem.

Os veículos de asa rotativa podem ter vários tipos de classificações, dependendo dos seus rotores. Podem apresentar um rotor apenas, rotores coaxiais, quatro rotores, entre outras configurações. Na Figura 4 está representado o *Spyro*, um veículo aéreo de asa rotativa com quatro rotores, desenvolvido pela empresa *UAVision*. A utilização de rotores faz com que não seja necessário o constante movimento do veículo para gerar a passagem de ar indispensável à sua sustentação. Deste modo, este tipo de veículos consegue pairar, ao contrário dos UAVs de asa fixa.

Figura 4 - Spyro. Fonte: UAVision

No entanto, existem algumas desvantagens associadas à utilização deste tipo de UAVs. Em primeiro lugar, as suas estruturas envolvem uma complexidade maior, comparativamente aos veículos de asa fixa. Consequentemente, as suas estruturas fazem com que estes veículos ofereçam uma maior resistência aerodinâmica, diminuindo a sua autonomia. Deste modo, devido às suas configurações, este tipo de veículos não consegue atingir grandes velocidades e as cargas úteis suportadas são normalmente menores, quando em comparação com as aeronaves de asa fixa.

1.2.3.2 Estação de Controlo

A estação de controlo pode estar localizada em terra, no mar, ou pode até ser uma estação móvel (Segor, 2010). Pode ser representada como o centro de controlo e monitorização de um UAV, onde as missões são planeadas e carregadas. No entanto, pode também fazer parte de um sistema maior, onde existe a partilha de informação por várias estações de controlo. Neste último, o controlo e monitorização do veículo pode ser distribuído por várias estações.

Do mesmo modo, representa a interface entre o operador e o sistema, podendo transmitir comandos ou informação para o veículo através de um *up-link*, sistema de comunicação que irá ser explicado no seguinte subcapítulo. Recebe ainda informação do veículo, através de um *down-link*. A informação recebida está normalmente associada à carga útil, a sistemas de posição geográficos ou a outros elementos que indiquem o estado do UAV (Jovanovic, 2008).

A estação de controlo pode também estar ligada a estações externas, de modo a recolher ou fornecer informação relevante para o desempenho da missão, tais como meios de aquisição de dados meteorológicos ou a transmissão de informação importante para outro sistema na rede (Austin, 2010). Na Figura 5 está representado umm exemplo da troca de informação entre a estação de controlo e outros sistemas.

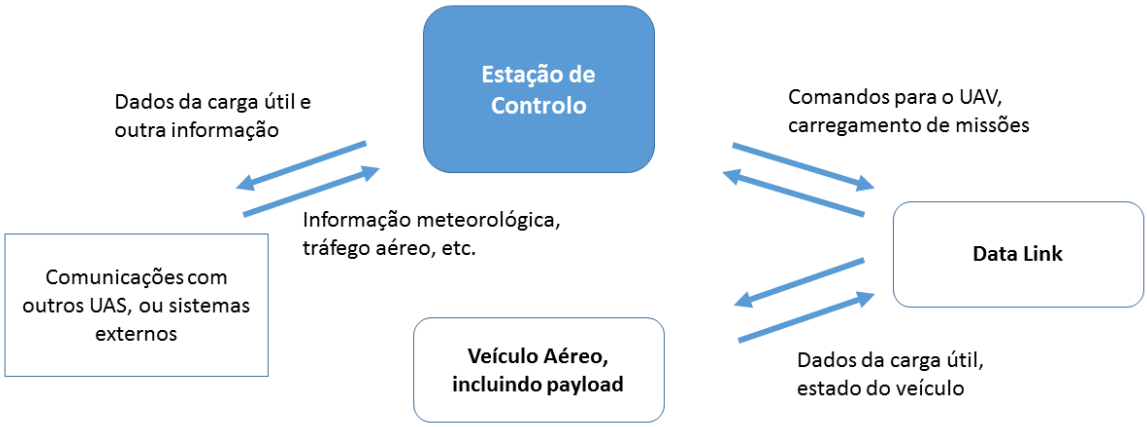

Figura 5 - Exemplo básico de troca de informação de um sistema aéreo não tripulado

Dependendo da grandeza de todo o sistema e do nível de detalhe pretendido para um certo tipo de missão, a estação de controlo pode ser um simples computador ou pode ser mais complexa (Figura 6), estando dividida nos seguintes subsistemas:

- Controlos de voo do UAV, onde está também incluída a monitorização do estado atual do veículo. Neste subsistema pode estar incluído o comando manual do veículo mas também a programação dos comandos automáticos de voo;
- Um subsistema que reconheça automaticamente o tipo de carga útil existente no veículo, ajustando-se também para a sua operação. Este subsistema monitoriza igualmente o estado e informação da carga útil;
- Os elementos terrestres de comunicação entre o UAV e a estação de controlo, incluindo transmissores e recetores. Neste subsistema podem também estar contidos componentes que fazem a monitorização do estado das comunicações;
- Sistemas para a monitorização da navegação, mostrando o percurso e a posição do UAV em tempo real;
- Sistemas com mapas do ambiente ao redor, de modo a poder ser feito o planeamento da missão e cálculos necessários para que esta seja desempenhada;
- Sistemas de comunicação com outras estações de controlo, UAVs, ou estações externas, de modo a efetuar a partilha de informação.

Figura 6 – Exemplo de estação de controlo. Fonte:www.uasvision.com

1.2.3.3 Ligação de Dados

A ligação de dados, ou *data link*, é o elemento responsável por estabelecer a comunicação entre o veículo e a estação de controlo e vice-versa. Pode ser divida em duas:

- *Up-link*, representando a ligação entre a estação de controlo e o UAV, tipicamente associada a comandos transmitidos pelo operador;
- *Down-link*, representando a ligação no sentido oposto, ou seja, entre o UAV e a estação de controlo, associada a dados da carga útil, tais como imagens de vídeo, podendo também transmitir dados sobre o estado do veículo, entre outros.

Os requisitos utilizados na ligação de dados estão normalmente associados a dois parâmetros fundamentais:

- Fluxo de transmissão, que representa a informação transmitida por segundo num canal de comunicações, sendo que é medido em bits por segundo;
- Largura de banda que corresponde ao espetro de frequências ocupado por um determinado sinal, ou seja, a diferença entre a frequência mais alta e mais baixa de um canal de comunicações, medida em Hz.

A comunicação pode ser realizada a partir de três diferentes meios: laser, fibra ótica e rádio. O meio de comunicação via laser não tem sido utilizado para este tipo de aplicações, principalmente devido à absorção atmosférica, que limita bastante o alcance e a robustez da ligação de dados (Ricklin, 2002). Por outro lado, a fibra ótica apresenta diferentes características, tais como alta *data rate* e também segurança de comunicações. No entanto, por ser uma ligação física, são muitas as desvantagens deste meio. A fibra ótica teria de estar armazenada no veículo ou na estação de controlo e poderia facilmente ficar presa por obstáculos, limitado simultaneamente o seu alcance. A utilização da fibra ótica como meio de ligação de dados de um UAS foi apenas um protótipo, sendo que nunca chegou a ser testado. Assim, a grande maioria dos sistemas UAS utiliza comunicação rádio na ligação de dados (Austin, 2010).

Como referido anteriormente, a ligação de dados nos sistemas aéreos não tripulados é realizada através de ondas de rádio. Este meio de comunicação é conseguido

através da propagação de ondas eletromagnéticas no espaço livre. As frequências compreendidas entre os 3kHz e os 3GHz são consideradas radiofrequências. Entre os 3GHz e os 300GHz, as frequências são designadas micro-ondas. Apesar de serem usadas para transmitir sinais rádio, estas ondas são normalmente utilizadas para comunicações em linha de vista, uma vez que não são refratadas[4] na atmosfera (Pereira, 2014). A Tabela 3 representa o espetro das ondas de rádio.

Tabela 3 - Espetro das ondas rádio

Nome da banda de frequências	Sigla	Frequências	Exemplos de aplicações
Very Low Frequency	VLF	3-30kHz	Comunicações com submarinos ou ajudas à navegação.
Low Frequency	LF	30-300kHz	Radiodifusão ou ajudas à navegação.
Medium Frequency	MF	300-3000kHz	Radiodifusão ou comunicações com navios.
High Frequency	HF	3-30MHz	Radiodifusão, rádios amadores ou comunicações militares.
Very High Frequency	VHF	30-300MHz	Radiodifusão FM, estações de televisão, comunicações militares.
Ultra High Frequency	UHF	300-3000MHz	Estações de televisão, telecomunicações ou comunicações aéreas.
Super High Frequency	SHF	3-30GHz	Telecomunicações, radar, comunicações via satélite.
Extremely High Frequency	EHF	30-300GHz	Radar, comunicações via satélite.

As radiofrequências são refratadas na atmosfera e, por este motivo, conseguem ter longos alcances. No entanto, à medida que a frequência diminui, o fluxo de transmissão também diminui. Por outro lado, com frequências mais altas é possível obter um fluxo de transmissão mais elevado, diminuindo no entanto o alcance das comunicações. É necessário obter um compromisso entre o fluxo de transmissão e o alcance das

[4] A refração atmosférica causa o desvio de uma onda eletromagnética de linha reta devido à variação na densidade do ar. É essencial para existirem comunicações para além da linha de vista.

comunicações, aquando da escolha das frequências ideais para um certo *data link*.

Como referido anteriormente, o *datalink* pode ser dividido entre *uplink* e *downlink.* Visto que o *uplink* é tipicamente utilizado para transmitir comandos para o UAV, a largura de banda necessária pode tomar valores na ordem de poucos kHz. Pelo contrário, o *downlink* é normalmente utilizado para transmitir imagens ou vídeo desde o UAV para a estação de controlo. Esta transmissão requer uma largura de banda maior, podendo tomar valores que normalmente estão compreendidos entre os 300kHz e os 10MHz (Austin, 2010).

De modo a haver comunicação entre o UAV e a GCS e entre UAVs é necessária a existência de protocolos de comunicação. A função destes protocolos é definir como é feita a troca de informação entre sistemas, sendo esta normalmente realizada através de mensagens formatadas. Assim, existem vários protocolos que definem esta comunicação, sendo que diferem no meio em que operam, tais como aéreo, terrestre ou marítimo, no tipo de veículos que utilizam, no tipo de segurança de comunicações que proporcionam, entre outros.

O STANAG 4586 é um exemplo deste tipo de protocolos. Como irá ser explicado no próximo subcapítulo, este *standard* é mais abrangente, uma vez que não define apenas os formatos de mensagens que devem ser utilizados, mas também as arquiteturas e interfaces com o operador. Outro exemplo é o MAVLink, que é utilizado para veículos aéreos não tripulados de pequenas dimensões. Trata-se de um protocolo de comunicações simples, sendo bastante utilizado em UAVs no meio civil. Outros protocolos são o JAUS, que pode ser utilizado com qualquer tipo de veículo autónomo, ou o *Robot Operative System* (ROS), que foi inicialmente desenvolvido para veículos terrestres mas que atualmente já é capaz de ser utilizado em qualquer tipo de veículos.

1.3 STANAG 4586

O STANAG 4586 é um *standard* que define as arquiteturas, protocolos, elementos de dados e interfaces entre os vários sistemas de controlo de um sistema aéreo não tripulado. Este documento foi criado em 2003 e desde então tem sofrido várias atualizações. Atualmente consta na 3ª Edição, que foi promulgada em 2012. O objetivo

deste *standard* é a promoção de interoperabilidade entre os UAVs e todos os seus sistemas, permitindo operações conjuntas entre várias forças NATO. Desta forma, são definidas as características e interfaces que os vários elementos do sistema aéreo não tripulado devem possuir de modo a poderem atingir um certo nível de interoperabilidade. A GCS, o UAV e os elementos de comando, controlo e comunicação são alguns dos elementos que compõem o sistema aéreo não tripulado segundo o STANAG 4586 (NATO Standardization Agency, 2012).

O STANAG 4586 está dividido em dois anexos: anexo A, que contém todos os termos e definições utilizadas e anexo B, que representa a maior parte do *standard* e é onde estão representadas as interfaces, protocolos e bibliotecas de mensagens que são essenciais para esta investigação. Este *standard* está sobre uma atenção continuada por parte da sua equipa de suporte *Custodian Support Team*.

Nas operações conjuntas, os UAVs podem ser utilizados como ferramenta de apoio nas decisões do comando tático. Deste modo, a necessidade da interoperabilidade entre sistemas advém da falta de comunicação que existia antes de ser criado o STANAG 4586. Este facto está representado na Figura 7, onde os sistemas aéreos não tripulados de cada nação não realizavam trocas de informação entre si. Cada SANT era composto pelo seu próprio *data link*, protocolo de comunicação e formatos de mensagens na troca de informação entre o UAV e a GCS. A disseminação da informação era realizada por meios indiretos, por exemplo a partir da GCS para um sistema próprio e finalmente para o utilizador, o que gerava sobrecarga de dados aquando da sua recolha. Como consequência, existiam perdas de tempo, em operações que normalmente requerem tempos de resposta praticamente imediatos.

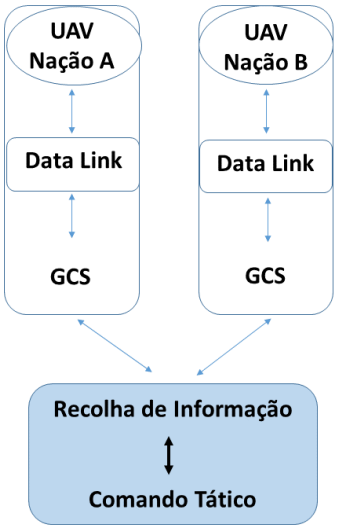

Figura 7- Panorama sem interoperabilidade

Este *standard* começa por definir conceitos básicos sobre a arquitetura de um sistema aéreo não tripulado. Desta forma, e como referido no subcapítulo 1.2.2, que representou várias abordagens de arquiteturas sobre estes sistemas, o STANAG 4586 divide o sistema aéreo não tripulado em 5 elementos:

- UAV, onde estão incluídas as unidades de propulsão e da plataforma;
- Carga útil ou *payload*, onde podem estar incluídos vários sistemas que sejam necessários para a condução da missão, tais como sistemas de armas ou sistemas de recolha de imagem;
- *Data link*, que pode ser dividido em duas unidades, o *Vehicle Data Terminal* (VDT), que está inserido no veículo e o *Control Data Terminal* (CDT) que está inserido na estação de controlo;
- UAV *Control System* (UCS) ou estação de controlo, onde são feitas todas as interfaces com os sistemas de controlo do UAV, com a carga útil e com o operador;
- Lançamento e Recolha, que representa os elementos responsáveis por esta função.

A Figura 8 representa um exemplo de arquitetura com os vários elementos que fazem parte do sistema aéreo não tripulado segundo o STANAG 4586 (Stansburys, 2009). Na Figura estão também representados sistemas externos de *Command, Control, Communications, Cumputers and Intelligence* (C4I), que podem estar ligados à estação de controlo de modo a que existam trocas de informação pertinentes.

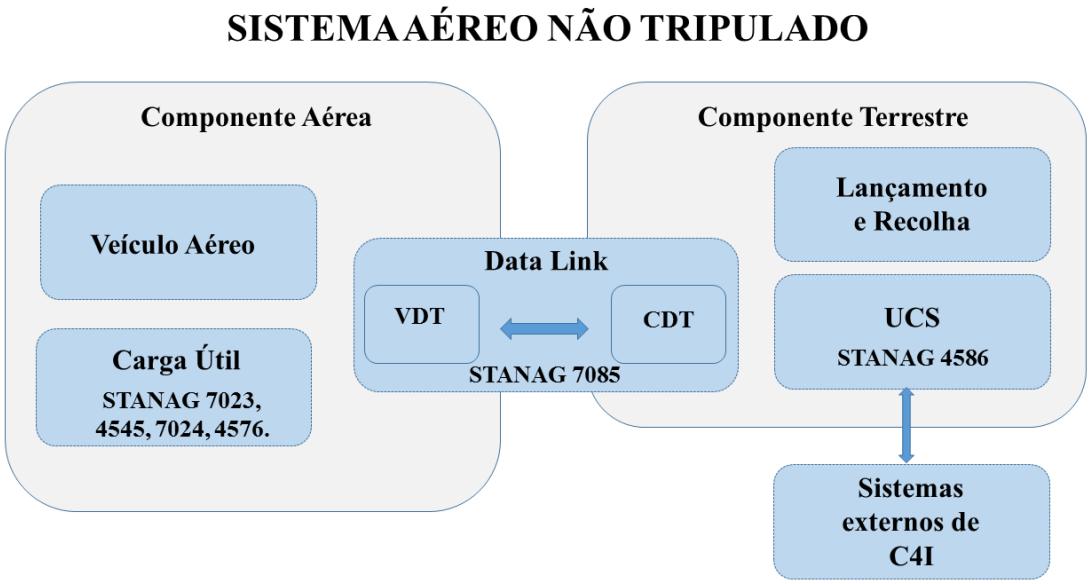

Figura 8 - Arquitetura de um sistema aéreo não tripulado

Na Figura 8 estão ainda identificados outros *standards* que podem ser aplicados aos sistemas aéreos não tripulados. Assim, o STANAG 7085 representa um *standard* para a interoperabilidade entre *data links*, enquanto que o STANAG 7023, 4545, 7024, 4575, entre outros, permitem a interoperabilidade entre a carga útil e o veículo aéreo. Contudo, não existiam *standards* que definissem as interfaces entre o veículo aéreo e a GCS ou que identificassem que informação deveria ser exibida ao operador. O STANAG 4586 veio colmatar estes problemas, definindo também níveis de interoperabilidade.

1.3.1 Níveis de Interoperabilidade

O STANAG 4586 define cinco níveis de interoperabilidade, que descrevem a capacidade que um sistema aéreo não tripulado deve ter de modo a ser identificado como interoperável. Para atingir um certo nível de interoperabilidade é necessário estabelecer

as interfaces que o STANAG 4586 introduz. Assim, os níveis de interoperabilidade são os seguintes:

- Nível 1, receção e/ou transmissão indiretas de dados associados aos sensores;
- Nível 2, receção direta no UAV de dados de sensores. Contudo, não existe controlo sobre o UAV;
- Nível 3, controlo e monitorização sobre a carga útil do UAV, a menos que especificado como apenas monitorização;
- Nível 4, controlo e monitorização sobre o UAV, a menos que especificado como apenas monitorização. O lançamento e recolha do UAV não fazem parte deste nível;
- Nível 5, controlo e monitorização sobre o UAV e também sobre a sua recolha e lançamento, a menos que especificado como apenas monitorização.

1.3.2 Arquitetura funcional da GCS

O STANAG 4586 define os seguintes elementos para a arquitetura funcional de uma GCS:

- *Vehicle Specific Module* (VSM), que está entre o DLI e o veículo aéreo e é utilizado para facilitar a compatibilidade dos formatos de dados entre o *data link* e o UAV;
- *Data Link* Interface (DLI), que providencia mensagens formatadas de modo a estabelecer a comunicação entre a *Core* GCS e o veículo;
- *Core* GCS, que providencia ao operador a capacidade de conduzir todas as fases da missão, assim como interfaces gráficas de modo a controlar o UAV e a sua carga útil, de acordo com o nível de interoperabilidade pretendido;
- *Command Control* Interface (CCI), faz a interface entre o *Core* GCS e sistemas de C4I externos, especificando os requerimentos que devem ser adotados para que exista comunicação;

- *Command and Control Interface Specific Module* (CCISM), onde existe um *software* ou *hardware* que faz a conversão entre a CCI e os sistemas de C4I externos que não sejam compatíveis;
- *Human Computer* Interface (HCI), que representa a interface gráfica que deve ser estabelecida na Core GCS, de modo a ser utilizada por um operador.

Um exemplo de arquitetura funcional de uma GCS está representado na Figura 9. O método que o sistema aéreo não tripulado utiliza para fazer a comunicação entre o UAV e a *Core* GCS é baseado na troca de mensagens formatadas. Estas mensagens estão definidas no STANAG 4586.

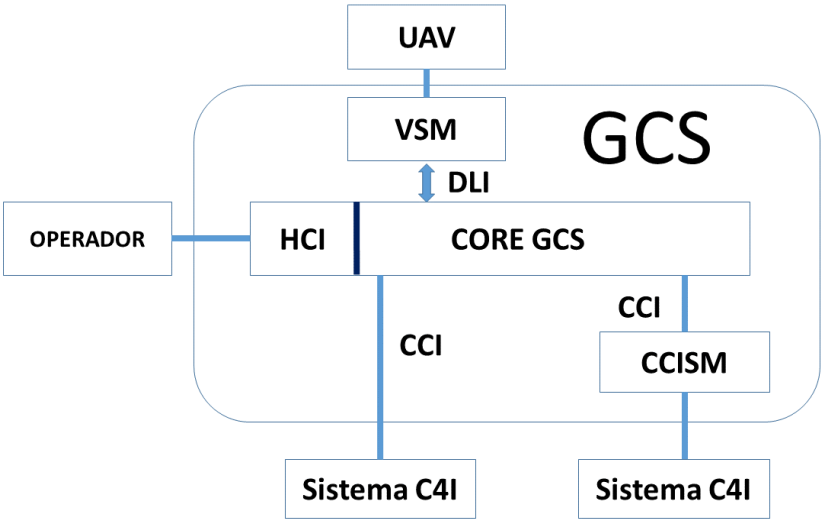

Figura 9 - Arquitetura funcional de uma GCS

1.3.3 Mensagens Formatadas

As mensagens formatadas são responsáveis pela troca de informação entre o veículo e a GCS. Desta forma, devem corresponder a um certo formato que está descrito no *standard*. A Figura 10 representa o formato que todas as mensagens devem seguir.

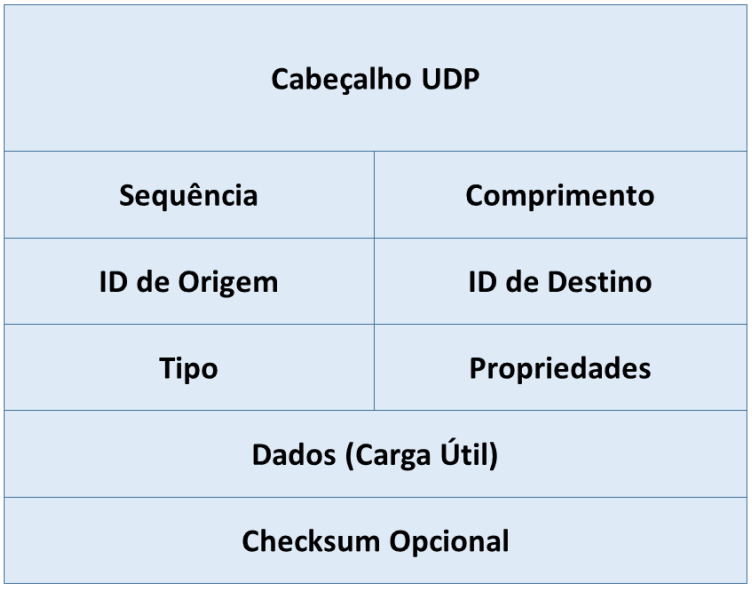

Figura 10 - Formato de mensagens STANAG 4586

O cabeçalho *User Datagram Protocol* (UDP) é composto por campos que fazem parte deste protocolo. O UDP é um protocolo da camada de transporte do modelo *Open System Interconnection* (OSI) e é utilizado para estabelecer ligações entre dois sistemas, não existindo garantia de receção do pacote a enviar. Esta investigação não aborda os campos do cabeçalho UDP, uma vez que não é esse o objetivo principal, mas sim todos os outros campos que fazem parte de uma mensagem STANAG 4586.

A sequência servia para segmentar os dados a partir de uma mensagem, gerando vários blocos de dados. Este campo não é utilizado atualmente e deve estar preenchido com "-1".

O comprimento da mensagem deve ser um valor inteiro de 16 bits, representado com um valor entre 1 e 528. O objetivo deste campo é fornecer o comprimento do campo de dados da mensagem, ou carga útil.

O ID de origem corresponde ao número de identificação de um determinado elemento do sistema aéreo não tripulado. Um exemplo é o veículo aéreo, quando envia uma mensagem de dados para a GCS.

Pelo contrário, o ID de destino corresponde ao número que identifica o elemento

de destino. No exemplo dado no parágrafo anterior, o ID de destino seria o número de identificação da GCS.

O tipo da mensagem representa o valor que identifica a mensagem. Todas as mensagens têm um tipo diferente e este campo é representado com 16 bits.

O campo correspondente às propriedades da mensagem é composto por 16 bits e está dividido em quatro partes. O bit mais significativo indica se a mensagem deve ser respondida com uma mensagem de *acknowledgement* ou não. Deste modo, se este bit tiver o valor "0" não é necessário *acknowledgement*, caso contrário, se for necessário, deve ter o valor "1". As duas partes seguintes identificam a versão do *standard* pela qual se estão a basear e se é utilizado o *checksum* opcional. A última parte não é utilizada, sendo que está reservada para uso futuro.

A carga útil representa os elementos de dados que se querem transmitir, tais como *waypoints* desde a GCS para o UAV, de modo a que este desempenhe uma missão. Os formatos deste campo estão descritos no *standard* e variam consoante o tipo de mensagem.

Finalmente, o *checksum* opcional é um código que pode ser utilizado de modo a identificar erros que possam ter ocorrido durante a transmissão da mensagem. Este campo é opcional porque já está presente um *checksum* no cabeçalho UDP. Por este motivo, não é abordado ao longo desta investigação.

O facto de ser um *standard* desenhado para ser utilizado no meio militar, contendo por isso a sua doutrina, é uma vantagem para os UAVs das Forças Armadas que utilizam o STANAG 4586, uma vez que é bastante completo. Contudo, uma das desvantagens está na dificuldade de implementação do *standard* visto que não existem exemplos *open-source*. Consequentemente, os custos de aquisição de sistemas compatíveis com o STANAG 4586 são elevados.

Em suma, o STANAG 4586 é um *standard* que visa promover a interoperabilidade entre os UAVs e os seus sistemas de controlo, garantindo a padronização entre este tipo de sistemas nos países membros NATO. Este objetivo é atingido através da definição de interfaces entre os vários elementos que compõem um

sistema aéreo não tripulado e também com o operador.

1.4 MAVLink

O MAVLink é um protocolo de comunicação *open-source* concebido para UAVs de pequenas dimensões. Apresentado pela primeira vez em 2009, este protocolo foi desenvolvido por Lorenz Meier e é baseado na troca de mensagens entre o veículo e a estação de controlo. Desde o seu lançamento, o MAVLink já sofreu várias atualizações, que irão ser mencionadas neste capítulo. Este protocolo é licenciado com a *GNU Lesser General Public License* (LGPL), uma licença de *software* livre, permitindo a utilização de várias bibliotecas disponíveis.

As mensagens MAVLink são formatadas em código binário e são bastante eficientes. Esta característica deve-se ao facto de as mensagens possuírem um cabeçalho reduzido de 8 bytes, existindo simultaneamente controlo de erros. Devido ao seu bom desempenho, este protocolo é bastante utilizado em vários pilotos automáticos, tais como o *ArduPilotMega* ou o *PxIMU Autopilot* e também por vários *softwares*, incluindo estações de controlo, tais como o *QGroundControl* ou o *APM Planner* (Coombes, 2012).

Este protocolo de comunicação utiliza mensagens que são criadas a partir de ficheiros *eXtensible Markup Language* (XML). No entanto é possível gerar código com outras linguagens de programação tais como C, *Python* e *Java*, possibilitando assim o uso do protocolo MAVLink com diferentes linguagens. O código é obtido através de programas que fazem a conversão entre XML e as linguagens de programação anteriormente referidas. Um exemplo desta conversão é o *Pymavlink*, que permite ler e criar mensagens MAVLink utilizando *Python*. Outra vantagem deste protocolo é a possibilidade que o utilizador tem de criar mensagens personalizadas (Meier, 2011).

1.4.1 Formato das Mensagens

Como referido anteriormente, as mensagens MAVLink possuem um formato específico. Uma mensagem pode ser dividida em duas partes principais: cabeçalho e carga útil. O cabeçalho contém a informação necessária para enviar e receber a mensagem, possuindo uma dimensão fixa de 8 bytes. A carga útil representa os dados que se pretendem transmitir e tem uma dimensão variável, dependendo do tipo de mensagem, sendo que no mínimo pode não ter qualquer byte e no máximo pode ter 255 bytes. Deste modo, a dimensão de uma mensagem MAVLink pode ir desde os 8 até aos 263 bytes (QGroundControl, s.d.). A Figura 11 representa o formato de uma mensagem, sendo que a explicação dos seus campos está resumida na Tabela 4.

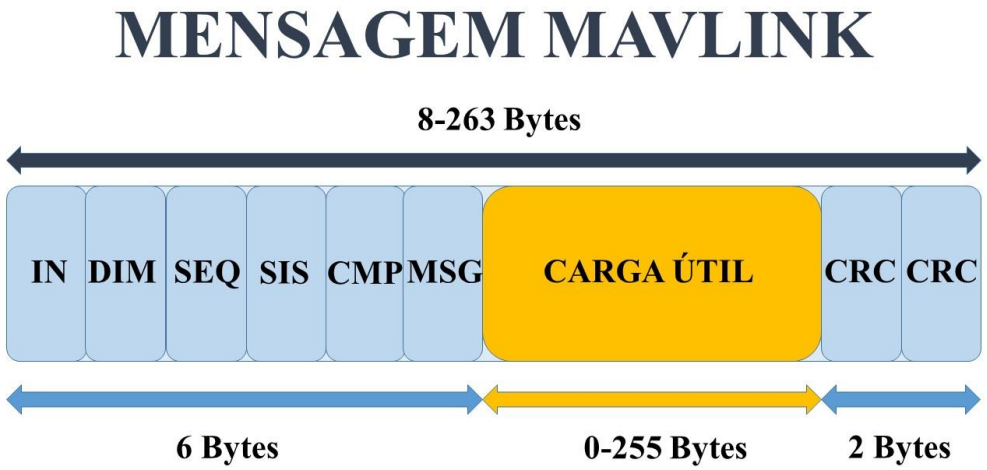

Figura 11 - Formato de mensagem MAVLINK

Para identificar o início de cada mensagem, é inserido um indicador com o valor hexadecimal de 0xFE. Após este indicador, são apresentados valores sobre a dimensão da carga útil a transmitir, a sequência da mensagem, a identificação do sistema, identificação do componente, identificação da mensagem, carga útil e, por fim, códigos de verificação de erros. A identificação do sistema pode tomar valores entre 1 e 255, permitindo assim a existência de 255 veículos diferentes na rede. A Tabela 4 representa o formato das mensagens, onde é atribuído o índice do *byte*, o conteúdo, o valor e a sua explicação.

Tabela 4 - Explicação dos campos da mensagem MAVLINK

Índice do Byte	Conteúdo	Valor	Explicação
0	Identificador de início de mensagem (IN)	0xFE	Indica o início de uma mensagem.
1	Dimensão da carga útil (DIM)	0-255	Indica a dimensão da carga útil.
2	Sequência da mensagem (SEQ)	0-255	Número da mensagem, de modo a verificar se alguma mensagem foi perdida.
3	Identificação do sistema (SIS)	1-255	Identifica o sistema que envia a mensagem, permitindo múltiplos veículos na rede.
4	Identificação do componente (CMP)	0-255	Identifica o componente do sistema que envia a mensagem (Ex: Câmara, piloto automático).
5	Identificação da mensagem (MSG)	0-255	Representa a identificação da mensagem e como a carga útil deve ser descodificada.
6 a (n+6)	Carga útil	Bytes (0-255)	Dados a transmitir, dependendo do tipo de mensagem.
(n+7) a (n+8)	Código de verificação de erros, *Checksum*	-	Cálculo baseado em códigos CRC.

Os códigos de verificação de erros utilizados são *Cyclic Redundancy Check* (CRC) e são utilizados nas comunicações digitais, de modo a identificar perdas de dados durante a transmissão. O cálculo dos códigos CRC acontece em dois momentos. O primeiro cálculo ocorre antes da transmissão e o segundo no momento da receção da mensagem. Se o código calculado na receção for igual ao código da transmissão, a comunicação ocorreu sem falhas. Se os códigos forem diferentes é necessário retransmitir a mensagem.

Este protocolo é vocacionado para comunicações rápidas e controlo de erros. Deste modo, a segurança das comunicações não é uma das suas prioridades, o que é uma

desvantagem principalmente para aplicações militares. Para perceber e comprovar esta realidade, foi realizado um estudo onde foram feitas várias simulações de ataques. Assim, foram testadas a confidencialidade e integridade das mensagens e também do canal de ligação de dados entre um UAV e uma GCS (Marty, 2013).

Existem vários motivos e formas de realizar ataques a um sistema aéreo não tripulado. Determinar a área que o UAV está a patrulhar, ou tomar controlo sobre este podem ser alguns dos objetivos dos ataques. Os testes realizados demonstram que o MAVLink é vulnerável a ataques, uma vez que foi possível obter informações sobre as posições, intenções do UAV e ainda tomar controlo sobre o mesmo. Foi violada assim a confidencialidade e integridade do sistema aéreo não tripulado. Um método de proteger as comunicações foi ainda apresentado no estudo, apesar de não ter sido testado (Marty, 2013).

O protocolo MAVLink tem sofrido várias atualizações e entre elas destacam-se a v.1.0.0 para v.1.1.0 e v.1.1.0 para v.2.0.0. Atualmente, o protocolo está mantido com a versão v.2.0.0, lançada em 2016, mas ainda suporta as versões v.1.0.0 e v.1.1.0. Estas atualizações ao protocolo compreendem normalmente a alteração ou adição de certas mensagens, mas podem também consistir em variações nos formatos, codificações ou até nos tipos de CRC utilizados.

A atualização para a versão v.1.1.0 consistiu numa série de atualizações que serão explicadas de seguida. A partir desta versão, cada sistema passa a ter de suportar uma série de mensagens definidas de modo a ser reconhecida a sua compatibilidade com o MAVLink. Alguns exemplos destas mensagens consistem no *Heartbeat*, *System Status*, entre outras. Várias mensagens foram também alteradas mantendo, no entanto, a compatibilidade com a versão anterior (QGroundControl, s.d.).

A atualização da versão v.1.1.0 para v.2.0.0 foi a mais recente e trouxe vários benefícios para a utilização do protocolo:

- Alteração do número de mensagens do protocolo, uma vez que estavam apenas disponíveis 8 bits (256 mensagens no máximo) para este campo e passaram a estar 16 bits;

- Adição da opção de colocar uma *password* de modo aumentar a segurança entre a ligação do MAVLink com o piloto automático;
- É mantido o suporte com as bibliotecas MAVLink da versão anterior;
- Suporte de *arrays* com dimensão variável.

1.4.2 *APM Planner* e MAVProxy

Tal como referido anteriormente, existem diversos *softwares open-source* que utilizam o MAVLink. O *APM Planner* e o MAVProxy são alguns deles, sendo que representam uma excelente ferramenta para ser utilizada, não só em simulações mas também em ambiente real.

O *APM Planner* é uma aplicação de estação de controlo, utilizada para enviar comandos de controlo e monitorizar o UAV. Esta aplicação pode integrar mapas do *Google Maps* e a sua utilização é bastante acessível para qualquer operador. A Figura 12 representa uma missão, onde foram introduzidos vários *waypoints*. O *APM Planner* permite ainda que o utilizador forneça ao UAV vários comandos e modos de operação, tais como o *Start Mission*, que permite começar a missão introduzida ou o *Return to Launch*, onde o UAV vai voltar ao ponto de descolagem. Uma das grandes vantagens desta aplicação é o facto de também ser compatível com um *software* de simulação, *Software In The Loop* (SITL). O SITL é bastante utilizado para testar pilotos automáticos ou programas, antes de serem testados em UAVs num ambiente real, visto que faz simulações com uma grande aproximação da realidade (Coombes et al., 2012).

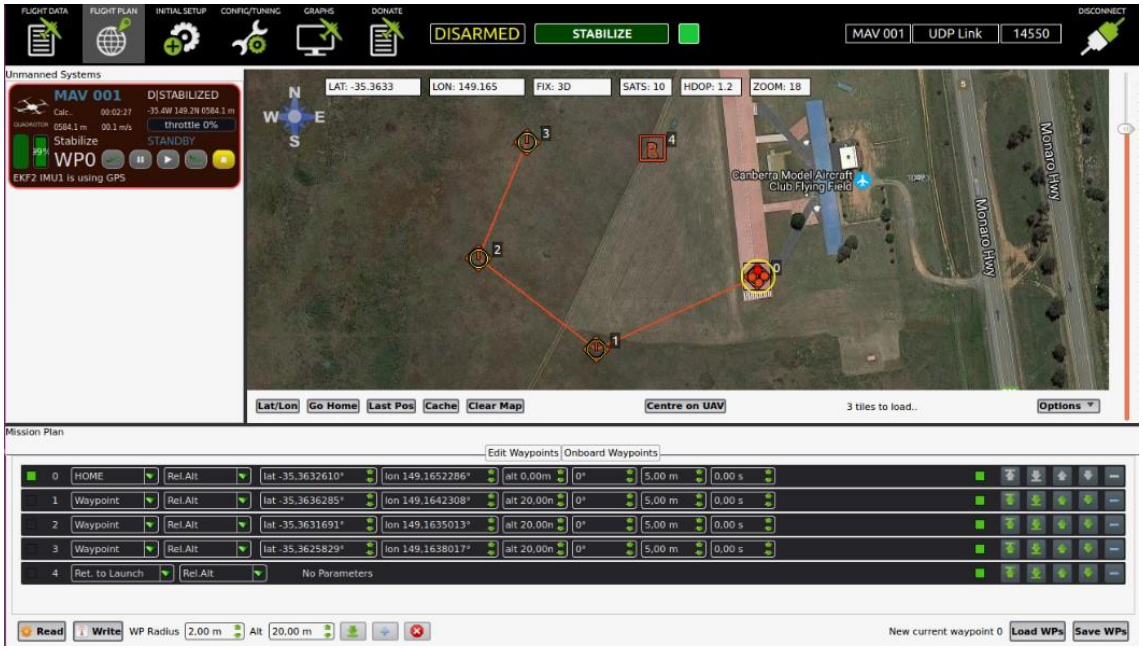

Figura 12 - APM Planner

O MAVProxy é uma estação de controlo *open-source* desenvolvida em *Python*, onde o veículo é controlado através de linha de comandos. Este *software* permite a utilização de qualquer comando utilizado no *APM Planner* e uma das suas vantagens é o facto de permitir a utilização de *Python*. Deste modo, qualquer utilizador pode desenvolver os seus programas nesta linguagem de programação, testando-os no SITL, visto que este simulador faz uso do MAVProxy. A Figura 13 ilustra a linha de comandos do MAVProxy.

```
alexandre@alexandre: ~/ardupilot/ArduCopter
SIM_VEHICLE: Run ArduCopter
SIM_VEHICLE: "/home/alexandre/ardupilot/Tools/autotest/run_in_terminal_window.sh
" "ArduCopter" "/home/alexandre/ardupilot/build/sitl/bin/arducopter" "-S" "-I0"
"--home" "-35.363261,149.165230,584,353" "--model" "+" "--speedup" "1" "--defaul
ts" "/home/alexandre/ardupilot/Tools/autotest/default_params/copter.parm"
SIM_VEHICLE: Run MavProxy
SIM_VEHICLE: "mavproxy.py" "--master" "tcp:127.0.0.1:5760" "--sitl" "127.0.0.1:5
501" "--out" "127.0.0.1:14550" "--out" "127.0.0.1:14551" "--console"
RiTW: Starting ArduCopter : /home/alexandre/ardupilot/build/sitl/bin/arducopter
-S -I0 --home -35.363261,149.165230,584,353 --model + --speedup 1 --defaults /ho
me/alexandre/ardupilot/Tools/autotest/default_params/copter.parm
Connect tcp:127.0.0.1:5760 source_system=255
Loaded module console
Log Directory:
Telemetry log: mav.tlog
Waiting for heartbeat from tcp:127.0.0.1:5760
MAV> STABILIZE> Received 809 parameters
Saved 809 parameters to mav.parm
mode guided
STABILIZE> GUIDED> arm throttle
GUIDED> arm throttle
GUIDED> takeoff 40
GUIDED> Take Off started
```

Figura 13 - MAVProxy

O MAVLink apresenta algumas vantagens e desvantagens, dependendo dos requisitos pretendidos e dos objetivos do UAV. Desta forma, algumas das suas vantagens são o facto de ser um protocolo simples, desenvolvido para obter comunicações rápidas e leves e também o facto de o acesso a bibliotecas e tutoriais ser fácil. Contudo, para aplicações relacionadas com o meio militar, este protocolo pode não ser o mais apropriado, visto que não tem o nível de complexidade inerente a este tipo de operações.

1.5 *Robot Operating System* (ROS)

O ROS, ou sistema operativo para veículos autónomos, é um *framework*[5] utilizado para o desenvolvimento de *software* para estes sistemas. A história do ROS remonta para 2007 e 2008, anos em que foram criados os projetos *Stanford Al Robot* (STAIR), pela Universidade de *Stanford*, e *Personal Robots* (PR), pelo laboratório de pesquisa de robótica *Willow Garage*. Estes projetos implementaram arquiteturas consideradas visionárias para a altura, sendo que contribuíram para o desenvolvimento do ROS. Assim, este *framework* foi criado com uma licença *open-source Berkeley Software Distribution* (BSD[6]), o que permitiu o seu rápido crescimento na comunidade de investigação da robótica móvel. A entidade gestora do ROS é da *Open Source Robotics Foundation* (Quigley, 2009).

É composto por várias bibliotecas e ferramentas que visam simplificar o processo de criação de *software* para a utilização em veículos autónomos. Desta forma, foi desenvolvido com a colaboração de várias entidades, cuja matéria de interesse são veículos autónomos, que foram contribuindo nas suas áreas de especialidade. A utilização do ROS permite a implementação de código de programação já criado e testado, que de outra forma demoraria tempo a desenvolver (Quigley, 2015).

O ROS é baseado no sistema operativo *Unix*, sendo que pode ser utilizado com outros sistemas operativos tais como o *Microsoft Windows* ou *Mac OS X*, mas apenas a título experimental. Existem cinco pontos que podem ser definidos como ideologias do

[5] Um *framework* é um *software* que providencia funções genéricas para o desenvolvimento de aplicações, a partir da utilização de programas e bibliotecas desenvolvidos pelos utilizadores.

[6] A licença BSD é permissiva, possibilitando a utilização de código existente sem custos, para fins comerciais e não comerciais.

ROS (Quigley, 2009):

- Ligações ponto-a-ponto;
- Baseado em ferramentas;
- Várias linguagens de programação;
- Ideologia *thin*;
- Sem custos e *open-source*.

Um *framework* com ligações ponto a ponto consiste num modelo descentralizado (Figura 14), onde cada sistema tem as mesmas capacidades, podendo iniciar um processo de comunicação. Por outro lado, um *framework* com comunicações baseadas num servidor centralizado não é uma opção viável aquando da utilização de sistemas heterogéneos, uma vez que estes possuem processos diferentes. Desta forma, os sistemas ROS consistem num número extenso de serviços que estão ligados entre si através da troca de mensagens, sendo que não existe um sistema de roteamento central.

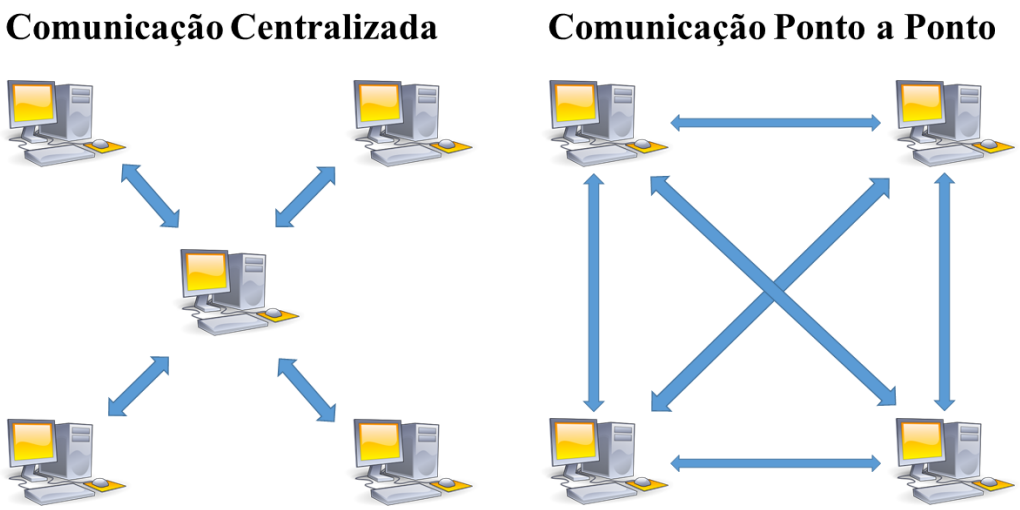

Figura 14 - Comparação entre comunicação centralizada e ponto a ponto

Ao contrário de outros *frameworks* de desenvolvimento de veículos autónomos, o ROS não é baseado num ambiente de gestão integrado. Assim, as várias funcionalidades, como por exemplo, a visualização das ligações de uma arquitetura, a visualização de dados em diferentes gráficos, o estabelecimento de ligações, entre outras, são realizadas

através de módulos distintos, criados a partir de múltiplas ferramentas que o ROS proporciona. A utilização destas ferramentas permite o desenvolvimento de várias implementações diferentes que podem ser adaptadas de acordo com a tarefa em questão.

Uma das ideologias do ROS consiste no facto de se puderem utilizar várias linguagens de programação. Esta ideologia proporciona uma grande vantagem para qualquer *framework*, uma vez que permite a utilização de linguagens de programação diferentes de acordo com a preferência do utilizador ou de acordo com os requisitos do programa. Assim, existem bibliotecas disponíveis em linguagens tais como *Python*, C++, *Lisp*, *Java*, *Ruby*, entre outras. Os módulos ROS comunicam entre si segundo uma convenção própria, que descreve como devem ser elaboradas as mensagens antes de serem transmitidas para a rede.

Tal como referido anteriormente, o ROS encoraja os investigadores a desenvolverem as suas próprias bibliotecas. No entanto, é expectável que estas tenham a capacidade de trocar mensagens com outros módulos. Para que tal aconteça, é necessário que as bibliotecas tenham um certo nível de abstração embutido, ou seja, não devem ser dependentes de uma determinada aplicação ou *hardware*. Esta ideologia pode ser denominada por *thin* e permite a reutilização de *software*.

A utilização do ROS incide sobre uma licença permissiva BSD, que permite a sua utilização a nível comercial e não comercial. Desta forma, a utilização deste *framework* não tem custos, sendo que é *open-source*. Esta é uma grande vantagem, uma vez que facilita o controlo de erros e testes de uma determinada biblioteca ou módulo.

Existem alguns conceitos fundamentais para entender a arquitetura do ROS (Quigley, 2015):

- Nós;
- Mensagens;
- Tópicos;
- Serviços.

O ROS é desenvolvido segundo uma conceção modular. Assim, um sistema pode ser dividido em vários nós. A ligação entre os vários nós é feita ponto a ponto e pode ser

visualizada através de grafos.

A comunicação entre diferentes nós é conseguida através da troca de mensagens. Estas mensagens consistem em formatos de dados estruturados, permitindo vários tipos de dados, tais como inteiros, flutuantes, booleanos, matrizes, entre outros.

Um nó envia uma mensagem a partir da sua publicação num determinado tópico. Por outro lado, quando um nó está interessado em receber uma mensagem, subscreve o tópico em questão. Assim, os tópicos podem ser utilizados para publicar, quando se quer enviar uma mensagem, ou para subscrever, quando se quer receber. É ainda possível a existência de múltiplos nós a publicar ou a subscrever o mesmo tópico.

O modelo de publicar/subscrever um determinado tópico pode ser flexível, mas o seu modelo de *broadcast* não é apropriado para comunicações síncronas. Estas comunicações são feitas a partir dos serviços, que permitem que um nó execute uma função que está contida num outro nó diferente. Desta forma, quando um nó necessita de uma determinada função, executa uma solicitação de serviço. Posteriormente, o nó que providencia a função, responde à solicitação com as definições necessárias para que se dê início ao acesso.

Em suma, o ROS é uma *framework* que apresenta várias vantagens. Em primeiro lugar, é composto por uma arquitetura modular, o que permite facilmente a reutilização de código e a conceção de variados tipos de sistemas. Adicionalmente, é mantido com uma licença BSD, o que permite a utilização do seu código sem custos, quer para fins comerciais ou outros. Contudo, uma das críticas apontadas ao ROS é a sobrecarga excessiva do seu sistema de mensagens, que pode acumular em projetos complexos ou com bastantes veículos autónomos.

1.6 Exemplos de aplicações anteriores

A interoperabilidade entre sistemas aéreos não tripulados é um tema bastante importante, principalmente em grandes projetos, onde sejam utilizados vários tipos diferentes de veículos autónomos. O *Integrated Components for Assisted Rescue and Unmanned Search Operations* project (ICARUS) é um desses projetos e é composto por 24 parceiros de 10 países diferentes. Desta forma, o seu objetivo é o desenvolvimento de

equipas de veículos autónomos, sejam eles aéreos, marítimos ou terrestres, de modo a serem projetados em situações de desastres para efetuarem busca e salvamento (Marques, 2016). Composto por 5 parceiros, o SEAGULL foi outro projeto e promoveu o desenvolvimento de soluções eficientes que colmatassem o défice que existe relativamente ao conhecimento situacional marítimo, fazendo uso de UAVs (Marques, 2015).

Deste modo, é possível concluir que cada vez mais existem projetos que utilizam equipas de veículos autónomos, devido a todas as suas vantagens. No entanto, a consequência da utilização destes veículos reside na falta de padronização que existe nos seus sistemas de controlo e comunicação. Num projeto em larga escala, é necessário existirem trocas de informação constantes entre os sistemas autónomos, de modo a chegar a processos de decisão rápidos. Se existir uma falta de padronização, este objetivo pode não ser atingido.

No entanto, uma solução possível para combater este problema pode ser a criação de tradutores entre diferentes protocolos ou *standards* de comunicação. Atualmente, já existem exemplos de tradutores, tais como entre STANAG 4586 e JAUS, entre JAUS e ROS, entre outros.

Como referido anteriormente, o projeto ICARUS utilizou vários veículos autónomos diferentes de modo a alcançar os seus objetivos. Deste modo, foi necessária a criação de sistemas que fizessem a tradução entre diferentes protocolos ou *standards* de comunicação. A arquitetura do ICARUS foi desenhada utilizando o JAUS e a tradução existente ocorreu com ROS (Marques, 2016). Outro exemplo de conversão é entre o STANAG 4586 e JAUS (Smith, 2012). O STANAG 4586 é bastante utilizado atualmente, principalmente na Marinha dos EUA. No entanto, este *standard* foi desenvolvido apenas para UAVs, sendo que não suporta veículos que operam no meio terrestre ou marítimo. Deste modo, tornou-se necessário realizar a conversão entre o STANAG 4586 e outro *standard* que pudesse ser utilizado por vários tipos de veículos, como é o caso do JAUS.

1.6.1 MAVROS

O MAVROS é um exemplo de *software* que faz a tradução entre o *framework* ROS e o protocolo MAVLink. É um pacote do ROS, fazendo parte das suas bibliotecas.

Providencia comunicação entre o ROS e vários pilotos automáticos que utilizam o MAVLink. Desta forma, o MAVROS está disponível *open-source*, sendo que a Figura 15 representa a sua ligação com o MAVProxy e SITL.

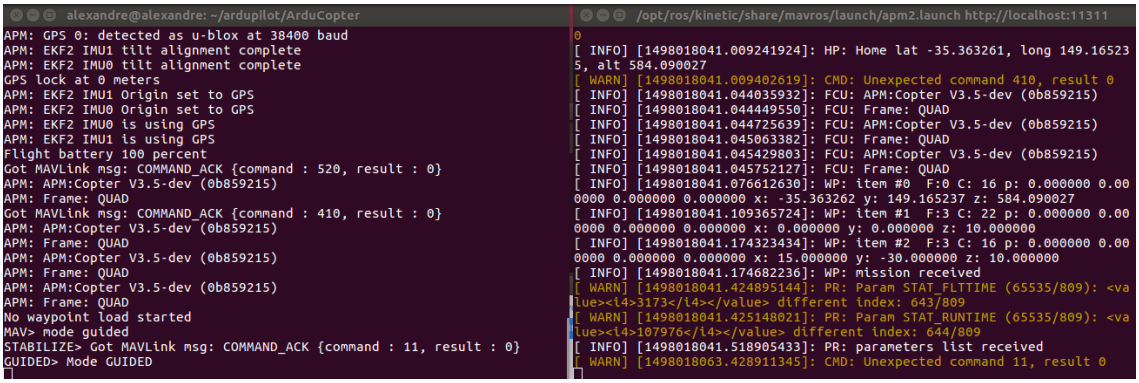

Figura 15 - Ligação entre MAVProxy e SITL

2. **Metodologias de Investigação**

As metodologias de investigação adotadas para esta dissertação são fundamentadas pelo método científico clássico e consistem na execução de várias etapas (Crawford, 1990). Adicionalmente, algumas das etapas estão também aplicadas de acordo com as metodologias de investigação para veículos autónomos (Nehmzow, 2009). As etapas estão ilustradas na Figura 16.

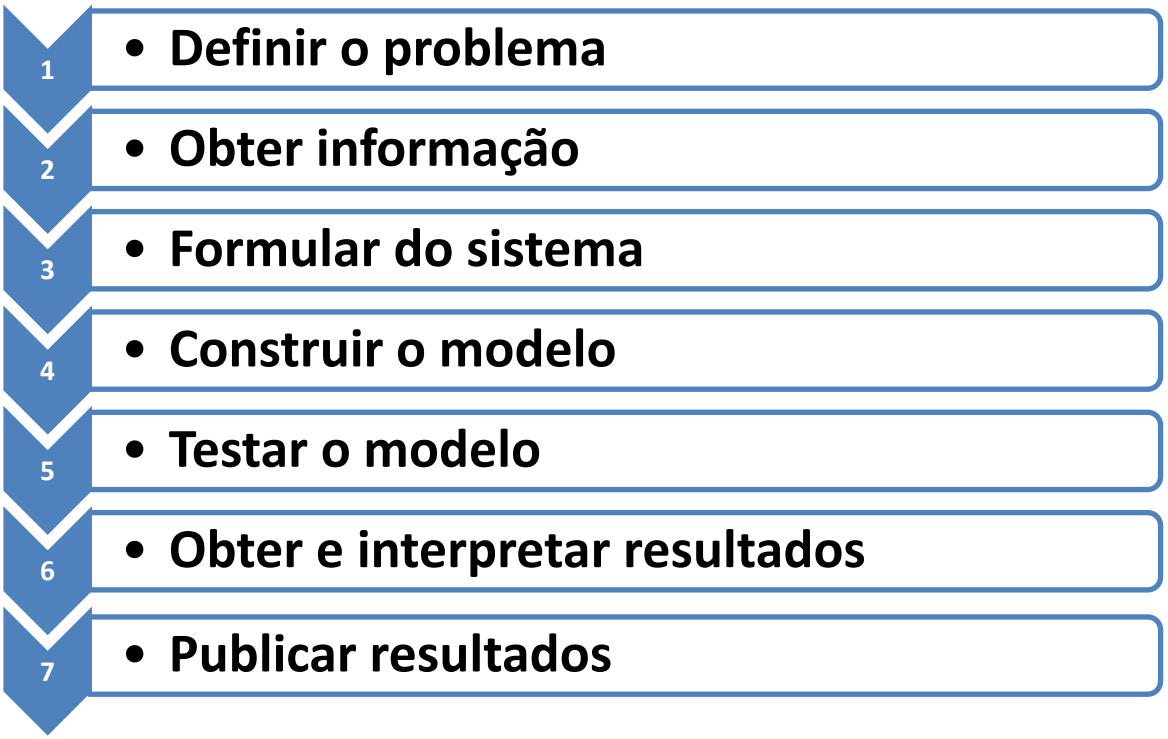

Figura 16 - Etapas adotadas segundo o método científico clássico

As etapas adotadas nesta investigação estão descritas nos parágrafos seguintes.

A primeira etapa consiste na definição do problema. Com base no que foi referido anteriormente, a falta de padronização é um dos constrangimentos atuais no que diz respeito aos veículos autónomos. Assim, o problema determinado é a falta de interoperabilidade entre uma estação de controlo que utilize o protocolo STANAG 4586 e um veículo autónomo que faça uso do protocolo MAVLink.

A obtenção de informação é a segunda etapa na realização da investigação. Esta etapa foi realizada no capítulo anterior e consistiu na pesquisa sobre os veículos autónomos, incluindo a sua arquitetura, protocolos de comunicação e trabalhos realizados anteriormente.

Formular o sistema representa a terceira etapa adotada. Com base na informação que foi obtida na etapa anterior, devem ser recolhidos dados de modo a identificar a melhor forma de solucionar o problema. Desta forma, a hipótese passa pela criação de um programa que faça a tradução entre STANAG 4586 e MAVLink. Este programa deve ser inserido num *hardware*, que será escolhido através de uma análise cuidada.

Contruir o modelo é a quarta etapa e faz parte do processo prático da investigação, fundamentada com a formulação que se realizou na etapa anterior. Inicialmente devem ser encontradas quais as mensagens que se querem converter. Posto isto, deve ser desenvolvido um programa em *Pytho*n, de modo a realizar a tradução, concebendo ainda uma interface gráfica para o utilizador.

A quinta etapa da investigação passa pela realização de testes. Para a realização destes testes foi escolhido um *software*. A escolha foi o SITL, uma vez que é utilizado por vários investigadores, de modo a experimentar os seus programas antes de os utilizar em ambiente real. O facto de proporcionar uma experiência bastante próxima da realidade e adicionalmente ser compatível com os restantes *softwares* que irão ser utilizados, resultou na escolha do SITL como sendo o mais apropriado para a realização de testes.

A sexta etapa consiste na obtenção e análise de resultados. Os resultados serão obtidos através da realização de vários cenários de validação. Adicionalmente, serão ainda comparados os resultados obtidos com outro *software* similar de tradução. Assim será possível avaliar o desempenho do tradutor desenvolvido, identificando a sua viabilidade.

A sétima e última etapa consiste na publicação de resultados. Uma investigação científica deve ser publicada de modo a contribuir ativamente para a área científica em que está inserida. Deste modo, esta etapa consiste não só na escrita desta dissertação de mestrado, como também na publicação de artigos em duas conferências científicas. As conferências científicas são a *SEA-CONF* 17, que decorre anualmente na Escola Naval Romena, e o *OCEANS'17 MTS/IEEE Aberdeen.*

As etapas descritas anteriormente foram fundamentadas pelo método científico clássico e adaptadas para esta investigação. No entanto, é necessário definir como se dará

o processo da conceção do modelo. Deste modo, existem várias metodologias que podem ser utilizadas para a criação do *software* e entre eles podem ser destacadas os seguintes (Balaji, 2012):

- Metodologia em cascata, que representa um processo de desenvolvimento sequencial, onde casa passo inicia após o término do anterior. Algumas das suas vantagens são a fácil implementação e o facto de os requisitos de cada passo serem claros antes de se dar início ao seguinte. Contudo, se existir um problema num dos passos deve ser tratado de imediato, porque se passar para o seguinte pode originar um problema estrutural;
- Metodologia em V, que é uma versão modificada da metodologia em cascata. No entanto, a diferença é que este não foi desenhado para ser linear, mas para permitir o retorno a passos do processo anteriores. Como vantagens tem o facto de permitir a alteração dos requisitos de qualquer passo, ao contrário do modelo em cascata. No entanto, se existirem alterações, todos os passos devem ter os seus requisitos atualizados. Adicionalmente, esta metodologia requer revisões em todos os passos, o que não é vantajoso na criação de modelos pequenos;
- Metodologia ágil, que é uma versão desenhada para obter uma rápida resposta. Deste modo, a grande vantagem desta metodologia é o facto de responder facilmente a alterações de requisitos. Contudo, se o modelo for em grande escala, torna-se difícil definir o tempo necessário para a sua realização.

Na conceção do modelo existem vários requisitos que devem ser considerados. Destes requisitos fazem parte o tempo reduzido que o programa deve demorar a responder, e os formatos fixos que as mensagens devem ter. Desta forma, os requisitos para a construção do modelo não variam, sendo que não é necessário recorrer ao modelo ágil. Como o modelo pode ser descrito como um processo linear, baseado em passos simples, a escolha do modelo em cascata é a opção escolhida para a sua conceção.

A Figura 17 representa a metodologia em cascata adaptada para a construção do modelo.

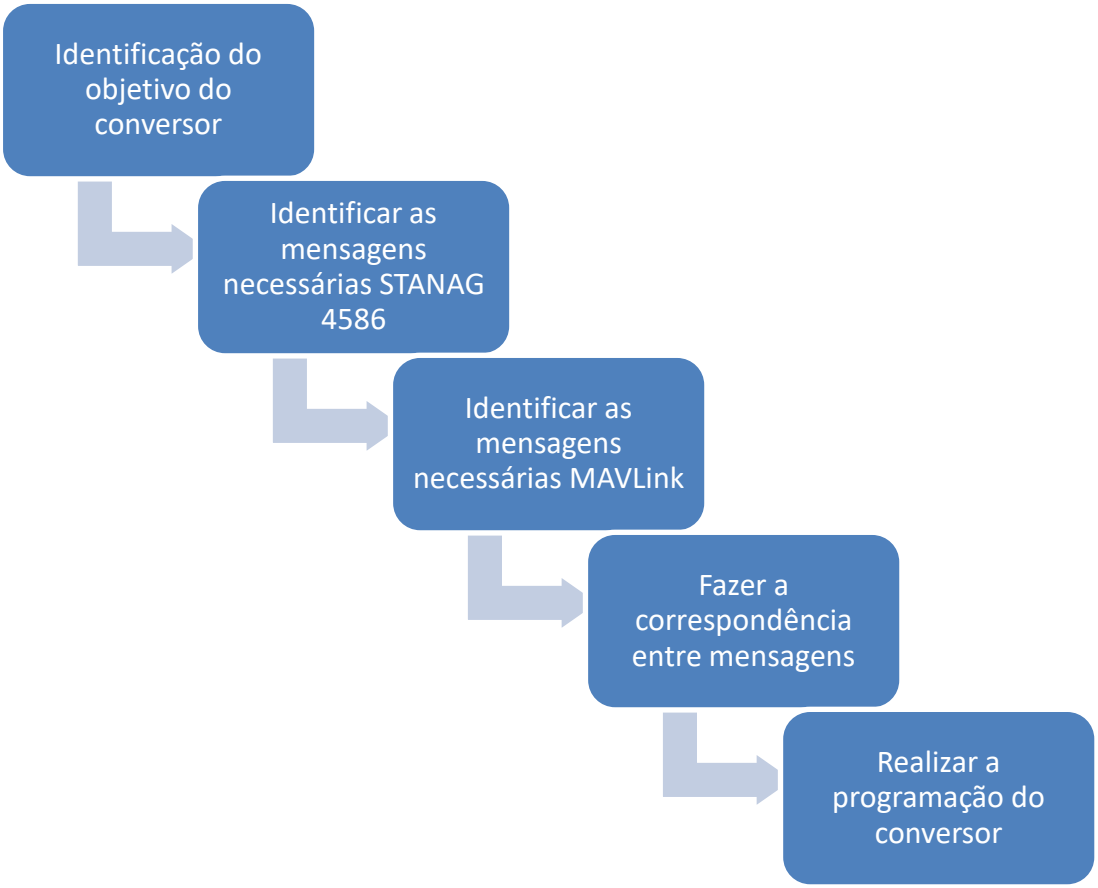

Figura 17 - Modelo em cascata

Como ilustrado na Figura 17, a conceção do modelo pode ser resumida a cinco passos. O primeiro passo passa pela identificação do objetivo do conversor. O objetivo é a transmissão de uma missão, constituída por um conjunto de *waypoints*, da estação de controlo que opera com STANAG 4586 para o veículo aéreo que utiliza MAVLink. Deste modo, o conversor deve ser capaz de traduzir as mensagens que estão associadas à passagem de *waypoints* e ao início da missão.

O segundo e terceiro passo passam pela identificação das mensagens necessárias para transmitir os dados requeridos. Deste modo, as mensagens devem ser identificadas em ambos os protocolos.

O quarto passo passa por corresponder as mensagens que estão no formato STANAG 4586 para o formato MAVLink, de modo a identificar as suas diferenças. Finalmente, o quinto passo representa a construção do conversor que está representada no desenvolvimento desta dissertação.

3. Desenvolvimento e Implementação do Modelo

3.1 Formulação da Hipótese

3.2 *Hardware*

3.3 *Software*

No capítulo anterior foram definidas as metodologias de investigação desta dissertação, onde foram definidos os vários passos necessários para solucionar o problema encontrado. Este capítulo está focado no estudo e desenvolvimento do sistema que fará a tradução de protocolos.

Deste modo, este capítulo está dividido em alguns subcapítulos. Primeiramente, com base nas metodologias adotadas, é formulada a proposta do sistema que dará resposta ao problema encontrado. De seguida, é escolhido qual o *hardware* que será utilizado, identificando as suas características. Posto isto, é representado os programas de *software* que foram criados, identificando a sua arquitetura.

3.1 Formulação do sistema

Tal como referido anteriormente, o problema que esta dissertação procura solucionar é a falta de compatibilidade que existe entre uma estação de controlo STANAG 4586 e um veículo MAVLink. De acordo com o estudo realizado no capítulo anterior, foram encontradas algumas soluções possíveis para o problema. Nos trabalhos realizados anteriormente foram estudados outros tradutores e conversores, entre diferentes linguagens de programação. Estes sistemas foram bem sucedidos, resultando na sua aplicação em projetos de larga escala. Para este caso, é também necessário criar um tradutor que faça a conversão entre os dois protocolos de comunicação: STANAG 4586 e MAVLink.

No que diz respeito à formulação do sistema, a questão seguinte passa por estudar a melhor solução sobre onde deve ser colocado o sistema. Para esta questão, existem duas soluções possíveis:

1. Colocar o tradutor na estação de controlo;
2. Colocar o tradutor no veículo aéreo.

A colocação do tradutor na estação de controlo é a primeira opção apontada. A vantagem de possuir o sistema neste espaço físico é o facto de ser apenas necessário obter um tradutor para existir comunicação com vários veículos que utilizem MAVLink. De outro modo, seria necessário obter vários tradutores, um para cada veículo MAVLink, o que poderia não ser viável para um projeto que contasse com vários veículos com este

protocolo. No entanto, com esta opção, a estação de controlo passaria a transmitir mensagens com o formato MAVLink. Consequentemente, não seria possível existir comunicação entre a estação de controlo e veículos aéreos que utilizassem outro protocolo. Assim, apesar de se resolver o problema de compatibilidade entre a estação de controlo STANAG 4586 e um veículo aéreo a utilizar MAVLink, surgiriam outros problemas de compatibilidade, como por exemplo, com veículos que utilizassem STANAG 4586, o que não seria de todo desejável.

Desta forma, a colocação do tradutor na estação de controlo é uma boa opção se apenas existirem veículos na rede que utilizem MAVLink. De outro modo, esta opção não é a mais indicada, uma vez que resulta numa falta de compatibilidade com veículos que utilizem outros protocolos, incluindo o STANAG 4586. A Figura 18 representa o modelo que foi referido anteriormente.

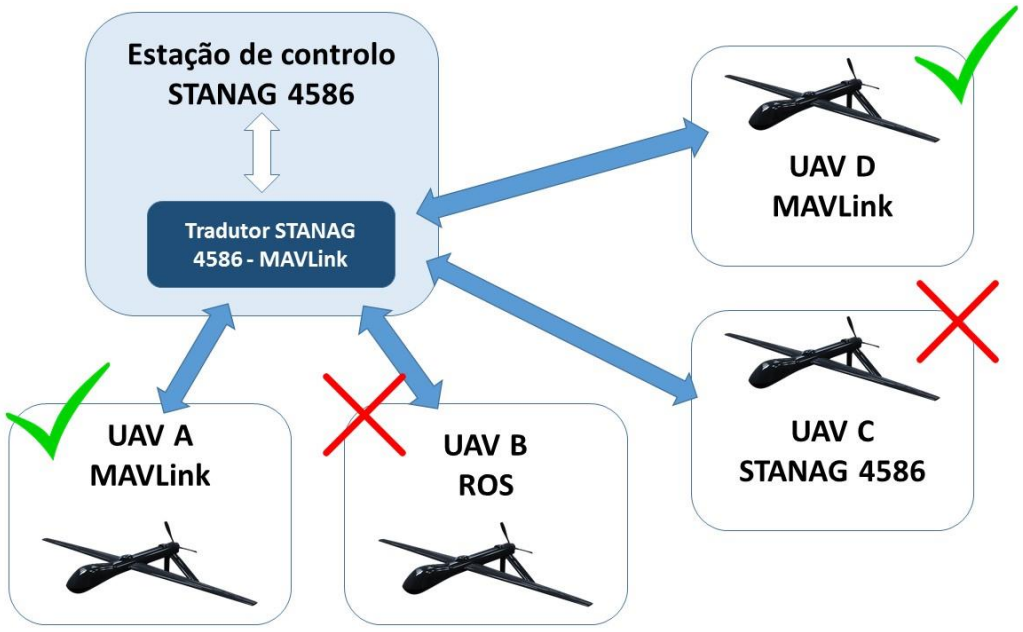

Figura 18 - Opção 1 - Colocação do tradutor na GCS

A segunda opção é a colocação do tradutor no veículo aéreo. A vantagem associada à utilização deste modelo é a possibilidade de existir comunicação entre uma estação de controlo STANAG 4586 e um veículo aéreo com o protocolo MAVLink,

permitindo simultaneamente comunicação com veículos que utilizem STANAG 4586. Esta situação é possível uma vez que a estação de controlo continua a transmitir com o seu protocolo de comunicação de origem, o STANAG 4586. Como desvantagem, com esta opção seria necessário obter um tradutor para cada veículo que utilizasse MAVLink.

Desta forma, a colocação do tradutor no veículo aéreo permite a compatibilidade entre uma estação de controlo STANAG 4586 e veículos que utilizem MAVLink, mantendo simultaneamente a possibilidade de comunicação com veículos STANAG 4586. Esta solução resolve o problema de compatibilidade. Para existir comunicação entre a estação de controlo e outros protocolos de comunicação, tais como ROS ou JAUS, é necessário criar outros tradutores nos respetivos veículos. O modelo que foi referido neste parágrafo está representado na Figura 19.

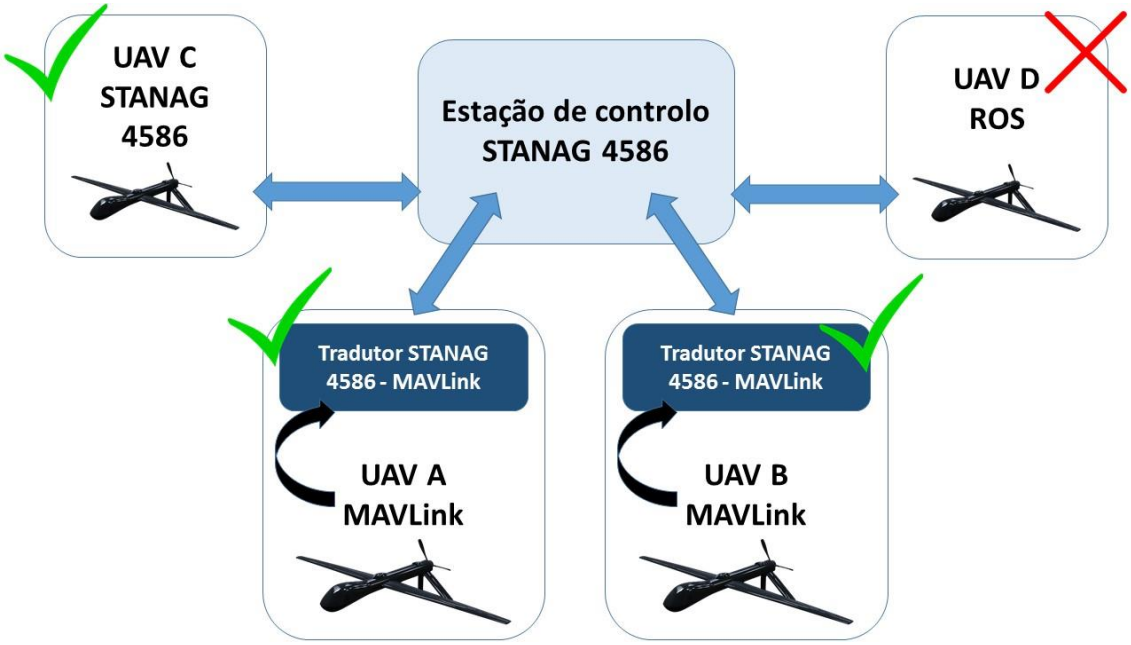

Figura 19 - Opção 2 - Colocação do tradutor no veículo

Após o estudo sobre as vantagens e desvantagens de ambas as opções, foi concluído que a solução mais viável é a colocação do tradutor no veículo. A seleção desta opção deu-se uma vez que o problema associado à compatibilidade da estação de controlo STANAG 4586 e veículos aéreos que operam com MAVLink é solucionado, continuando a existir simultaneamente compatibilidade com veículos que utilizem STANAG 4586, o que não acontece com a opção 1. Para existir compatibilidade entre a estação de controlo

e veículos aéreos que operem com outro protocolo de comunicação torna-se necessário criar outros tradutores que façam a conversão entre o STANAG 4586 e o protocolo em questão.

3.2 *Hardware*

A escolha do *hardware* é um processo bastante importante, uma vez que influenciará diretamente a eficácia do sistema. Assim, torna-se necessário identificar algumas características que o *hardware* deve possuir:

- Rapidez de processamento;
- Compatibilidade do dispositivo com qualquer veículo aéreo, ou com o maior número de veículos aéreos possível;
- Facilidade de introdução do tradutor que será criado;

Rapidez de processamento é a principal característica aquando da escolha do *hardware*, uma vez que a tradução deve ocorrer no menor período temporal possível para que não afete as comunicações. Simultaneamente, é também importante que o *hardware* possa ser facilmente introduzido numa grande variedade de veículos aéreos, de modo a que o tradutor não seja exclusivo para certos veículos. A facilidade de introdução do *software* de tradução no *hardware* é também um fator importante na escolha, uma vez que os *hardwares* existentes podem utilizar sistemas operativos diferentes. Este fator poderá influenciar a linguagem de programação que será utilizada para o desenvolvimento do tradutor.

Desta forma, existem algumas opções para a escolha do *hardware a utilizar*, sendo que cada uma tem as suas vantagens e desvantagens: *Arduino*, *NodeMCU* e *Raspberry Pi*. Assim, é necessário identificar as características que cada um destes sistemas possui, de modo a escolher a melhor opção para a realização do tradutor.

O *Arduino* é um microcontrolador desenhado numa placa de circuito. É um computador simples e tipicamente utilizado para controlar pequenos dispositivos, como sensores ou motores. Este *hardware* pode ser ligado diretamente a um computador ou a uma bateria, sendo possível começar a trabalhar com ele imediatamente. O *Arduino* não

foi desenvolvido para comunicar com outros computadores e por este motivo tem apenas uma porta USB para este processo. Detém um espaço para armazenamento bastante reduzido, apenas o suficiente para guardar o código do programa a utilizar. Desta forma, o *Arduino* é uma boa opção para uma tarefa que seja simples e repetitiva (Rai, 2013).

O *NodeMCU* é um pequeno microcontrolador, que também precisa de muito pouco para ser estabelecido. É apenas necessário escrever o código de programação e enviar para o dispositivo através de uma ligação USB. Neste *hardware*, os programas são escritos em Lua[7]. O *NodeMCU* é ligado a um módulo Wi-Fi ESP8266, pelo que é bastante fácil de estabelecer ligação à internet. Assim, este *hardware* pode ser visto como uma versão melhorada do *Arduino*, uma vez que é compatível com este, sendo que simultaneamente tem acesso a Wi-Fi (Stanovov, 2016) (Silveira, 2016).

O *Raspberry Pi* não é um microcontrolador, mas sim um computador, com o seu próprio sistema operativo. O sistema operativo que este *hardware* utiliza requer algum conhecimento de programação, especialmente *Python*. Tal como um computador típico, o *Raspberry Pi* também demora algum tempo no processo de ligar e desligar. Pode ser ligado a outros dispositivos através de portas *Universal Serial Bus* (USB) ou *Ethernet*. O *Raspberry Pi* necessita de ter um cartão de memória micro SD, sendo que possui bastante espaço de armazenamento, não só para o sistema operativo, mas como para qualquer programa que seja necessário. Desta forma, o *Raspberry Pi* é uma boa opção para tarefas mais complexas, como por exemplo para controlar um robot, ou para desenvolver um sistema de segurança de uma casa (Poole, 2015).

A partir do estudo realizado é possível concluir que o *Arduino* é ideal para programas simples e repetitivos, sendo que não é este o caso, uma vez que o tradutor terá que ser um programa dinâmico, capaz de traduzir várias funções. O *NodeMCU* pode ser visto como uma versão melhorada do *Arduino*, uma vez que é compatível com este, possuindo simultaneamente Wi-Fi. No entanto, o tradutor não necessita desta funcionalidade, uma vez que pode ser ligado diretamente ao veículo. O *Raspberry Pi* pode ser utilizado para programas mais complexos, sendo que pode ser também ligado

[7] Lua foi desenvolvida em 1993 e é uma linguagem de programação concebida para aplicações de *software*.

facilmente ao veículo. Desta forma, o *Raspberry Pi* é o *hardware* escolhido para o desenvolvimento do tradutor.

Existem vários modelos do *Raspberry Pi*, sendo que variam na rapidez de processamento, no consumo de energia, nas ligações, no acesso ao Wi-Fi, entre outros. Os *Raspberry Pi* 3 e *Raspberry Pi Zero W* são os mais recentes, sendo que contam com velocidades de processamento e valores superiores de memória RAM. No entanto, é escolhido o *Raspberry Pi 2 Model B*, uma vez que oferece um processador 900MHz *quad-core* ARM Cortex-A7 e 1GB de *Random Access Memory* (RAM), valores suficientes para que a tradução decorra sem problemas.

Desta forma, o *Raspberry Pi* será inserido no veículo, facilitando a ligação com este. A inserção do *software* que será desenvolvido é também um processo fácil, uma vez que o *Raspberry Pi* utiliza *Python*. A Figura 20 representa o esquema do sistema que será utilizado nesta investigação.

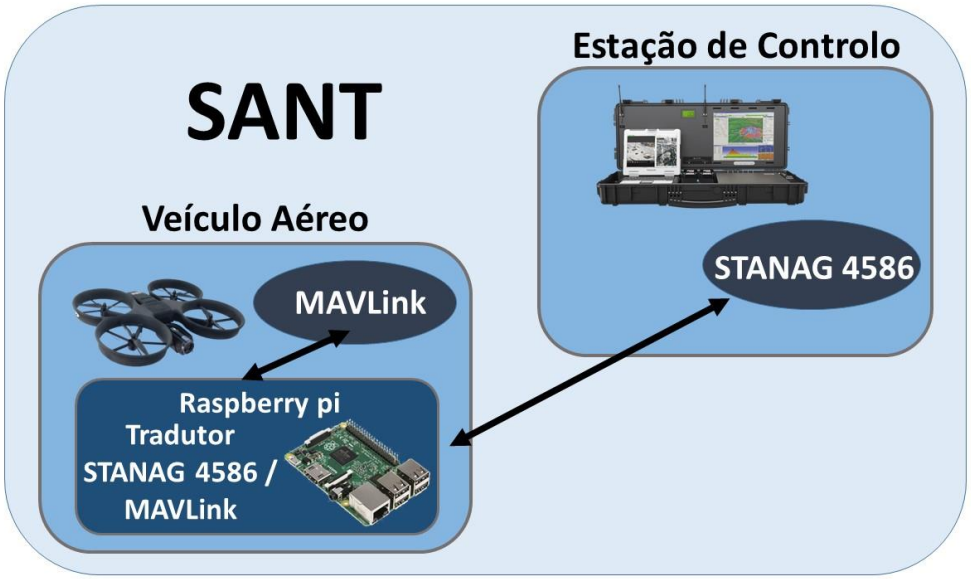

Figura 20 - Arquitetura final do modelo desenvolvido

3.3 *Software*

Após a realização do estudo sobre o *hardware* a utilizar, é necessário desenvolver o s*oftware* que fará a tradução entre os dois protocolos de comunicação. A criação deste *software* é uma etapa fundamental desta investigação. No entanto, a tradução de todo o

standard STANAG 4586 em MAVLink é um processo bastante extenso. Deste modo, é necessário identificar qual o objetivo do tradutor e escolher as mensagens a converter.

Como referido anteriormente, o objetivo do tradutor será converter mensagens de *waypoints* no formato do *standard* STANAG 4586 para o formato do protocolo MAVLink. Consequentemente, vai ser possível enviar uma missão planeada, ou seja, um conjunto de *waypoints*, a partir de uma estação de controlo que utilize STANAG 4586 para um veículo que utilize MAVLink.

Primeiramente, é necessário identificar qual a mensagem STANAG 4586 que tem a função descrita no parágrafo anterior. Essa mensagem é a *AV position waypoint* e tem o tipo 13002. Em segundo lugar, é preciso fazer a correspondência desta mensagem com as mensagens do protocolo MAVLink. Deste modo, a mensagem MAVLink responsável por enviar *waypoints* é a *MAV_CMD_NAV_WAYPOINT*. Finalmente, é necessário verificar a relação entre os campos de ambas as mensagens. Como referido anteriormente, o STANAG 4586 é um *standard* mais completo do que o protocolo MAVLink, e este facto é visível no número de campos que as mensagens de ambos os protocolos possuem. No STANAG 4586, a mensagem *AV position waypoint* possui 19 campos. Por outro lado, a mensagem MAVLink correspondente, *MAV_CMD_NAV_WAYPOINT*, possui 7 campos. Os campos de ambas as mensagens estão representados nas Tabelas 5 e 6.

Tabela 5 - Mensagem STANAG 4586

STANAG 4586
AV position waypoint
1. Presence Vector
2. Time Stamp
3. Waypoint Number
4. Latitude
5. Longitude
6. Location Type
7. Waypoint to Altitude
8. Altitude Type
9. Altitude Change Behavior
10. Speed
11. Speed Type
12. Next Waypoint
13. Turn Type
14. Optional Messages
15. Waypoint Type
16. Limit Type
17. Loop Limit
18. Arrival Limit
19. Activity ID

Tabela 6 - Mensagem MAVLink

MAVLink
MAV_CMD_NAV_WP
1. Hold Time
2. Acceptance Radius
3. Pass Radius
4. Yaw Angle
5. Latitude
6. Longitude
7. Altitude

A partir das tabelas 5 e 6 é possível verificar que os campos não possuem uma correspondência direta. Assim, é necessário verificar quais são os campos que estão relacionados e completar os que não estão. Na Tabela 7 é possível verificar a correspondência que existe entre as mensagens de *waypoints* do STANAG 4586 e do MAVLink.

Tabela 7 - Relação entre mensagens correspondentes

Relação entre mensagens de waypoints	
STANAG 4586	MAVLink
4. Latitude	5. Latitude
5. Longitude	6. Longitude
7. Waypoint to Altitude	7. Altitude
13. Turn Type	4. Yaw Angle

Após feita a correspondência entre as mensagens dos dois protocolos, é necessário identificar os programas de *software* que serão necessários. De modo a desenvolver o tradutor, é essencial possuir bibliotecas de ambos os protocolos para identificar os formatos das suas mensagens. Adicionalmente, é também importante possuir estas bibliotecas de modo a ser possível a realização de testes, verificando e validando o sistema criado. Assim, para realizar o *software* foi necessário passar por algumas etapas, que estão descritas na Figura 21.

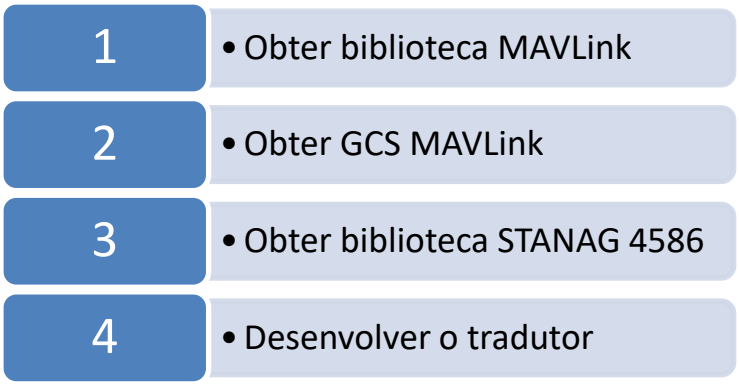

Figura 21 - Etapas para a realização do software

Tal como referido anteriormente, uma vez que tem uma licença LGPL, o MAVLink é um protocolo com suporte, bibliotecas e outras aplicações de fácil acesso. Assim, foi identificada a estação de controlo, que melhor se adaptaria a esta investigação, uma vez que deveria ser desenvolvida em *Python*. A estação de controlo escolhida foi o MAVProxy, que utiliza uma biblioteca de mensagens desenhada em *Python*, chamada *Pymavlink*. A utilização do MAVProxy permite um fácil acesso às mensagens MAVLink,

podendo ainda ser inserido no *Raspberry Pi* de modo a trabalhar em simultâneo com outras estações de controlo.

A obtenção de uma estação de controlo STANAG 4586 é um processo mais difícil, uma vez que este *software* tem que ser adquirido a empresas externas, chegando a tomar valores que podem ser elevados. Uma solução para este problema é o próprio desenvolvimento de umabiblioteca deste tipo. Deste modo, é necessário criar um protótipo de biblioteca com as mensagens necessárias do *standard* STANAG 4586 de modo a desenvolver o *software* de tradução.

3.3.1 Desenvolvimento do protótipo de uma biblioteca STANAG 4586

O desenvolvimento de uma biblioteca de mensagens é um processo complexo. É necessário que seja bem estruturado de modo a que seja fácil e rápido de utilizar. Deste modo, é necessário primeiramente estudar algumas bibliotecas *open-source* já existentes, de modo a entender como é realizado este programa e a tirar conclusões para que o protótipo da biblioteca STANAG 4586 fique bem estruturada.

Após alguma pesquisa, foi possível encontrar algumas bibliotecas de mensagens disponíveis. Inicialmente foi encontrada uma biblioteca STANAG 4586, mas com uma edição antiga. Devido a estar bastante desatualizada e também devido ao facto de ser escrita na linguagem de programação *Perl*, esta biblioteca não foi utilizada. Ainda assim, foi possível estudar a metodologia que foi utilizada para o seu desenvolvimento.

Apesar de serem desenvolvidas para outros protocolos de comunicação, foram ainda estudadas outras bibliotecas de mensagens. Entre elas, destaca-se a *Pymavlink*, que tal como já foi referido anteriormente, é uma biblioteca estabelecida para o protocolo MAVLink escrita em *Python*. O estudo desta biblioteca de mensagens é um processo importante, uma vez que irá também ser utilizada ao longo desta investigação. Assim, o desenvolvimento do protótipo da biblioteca STANAG 4586 teve em conta a biblioteca *Pymavlink*, uma vez que é com que esta que irá ocorrer a tradução. Este facto facilita a criação do tradutor, visto que as duas bibliotecas irão possuir a mesma estrutura lógica.

O protótipo da biblioteca de mensagens STANAG 4586 criada está disponível no Apêndice A. Foi desenvolvido com o objetivo de ser rápido e simples de perceber por

parte do utilizador. Devido à sua estrutura, a inserção de novas mensagens ou atualização de mensagens já existentes é também fácil.

O objetivo deste protótipo de biblioteca passa por realizar duas funções distintas: receber um conjunto de bytes de dados, identificando a mensagem ou mensagens respetivas e convertendo-os para caracteres segundo a tabela *American Standard Code for Information Interchange* (ASCII)[8], de modo a que sejam facilmente interpretados; enviar uma mensagem, onde inicialmente são recolhidos os dados para os campos da mensagem, codificando-os de seguida em código binário. Nas Figuras 22 e 23 estão representados estes dois objetivos, assim como as etapas necessárias para os atingir.

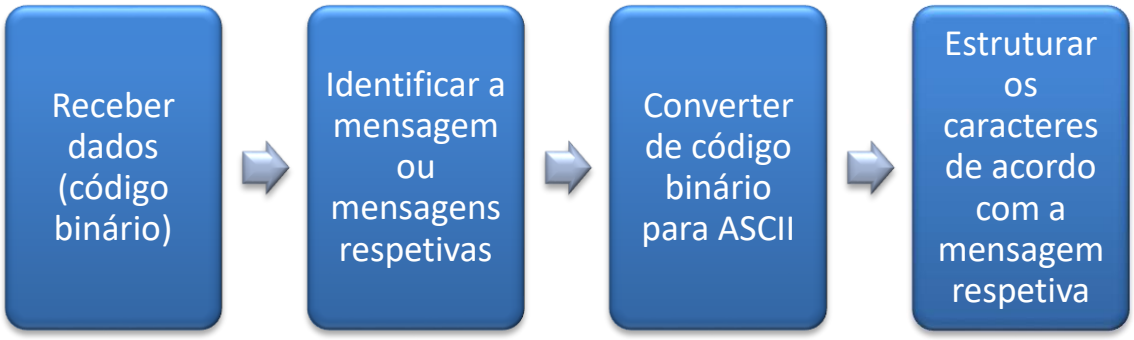

Figura 22 - Receber os Bytes

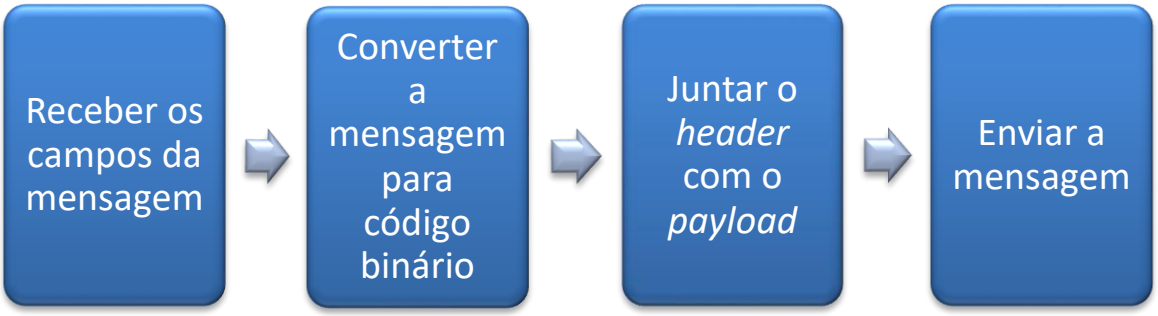

Figura 23 - Codificar e Enviar a mensagem

[8] A tabela ASCII é utilizada para converter valores binários em caracteres.

Assim como em várias outras linguagens de programação, o *Python* contém classes, que podem ser vistas como módulos que agrupam dados e funções. Desta forma, a biblioteca realizada contém algumas classes principais: *STANAG_message, STANAG_header, Payload* e finalmente a classe *STANAG*. De seguida estão explicadas estas classes principais.

3.3.1.1 Classe *STANAG_message*

Como estudado anteriormente, uma mensagem STANAG 4586 é composta por vários campos de dados. Nesta biblioteca, as mensagens foram divididas em dois módulos principais: *header* e *payload.* O objetivo da classe *STANAG_message* é unir estes dois módulos, resultando numa mensagem completa.

Esta classe é composta por duas funções, sendo que a primeira é responsável por inicializar a classe (está presente em todas as classes no *Python*) e a segunda, denominada *pack*, é responsável por fazer a união do *header* com o *payload*. Ao ser inicializada, esta classe recebe os dados necessários para criar uma classe *STANAG_header*, que irá ser descrita de seguida. Como referido anteriormente, a função *pack* é chamada quando é necessário fazer a junção dos dois módulos. No entanto, esta função não é responsável para passar os dados para código binário, uma vez que já os recebe neste formato.

Desta forma, a classe *STANAG_message* é uma classe fundamental no protótipo da biblioteca STANAG 4586. É uma classe central, uma vez que todas as mensagens passam por lá antes de serem enviadas ou após serem recebidas.

3.3.1.2 Classe *STANAG_header*

A classe *STANAG_header* é inicializada a partir da classe *STANAG_message*. Ao ser inicializada, esta classe cria seis campos de dados diferentes. Os seis campos de dados que esta classe cria correspondem aos campos que foram referidos no subcapítulo "STANAG 4586": *seq* corresponde a sequência; *mlen* a comprimento da mensagem; *source_id* a ID de origem; *dest_id* a ID de destino; *msg_type* a tipo da mensagem; *msg_prop* a propriedades da mensagem.

À semelhança da *STANAG_message*, esta classe é composta por duas funções. A primeira é responsável pela inicialização e a segunda, chamada *pack*, é responsável por

fazer a codificação do *header* para código binário (que posteriormente vai para a classe *STANAG_message* de modo a ser unido ao *payload*). Esta codificação é feita através de uma função que está disponível nas bibliotecas *Python*, chamada *struct.pack*.

Desta forma, a classe *STANAG_header* foi criada com o intuito de fazer a divisão da mensagem em duas partes: *header* e *payload*. Consequentemente, a atualização ou inserção de novas mensagens é um processo mais fácil, uma vez que apenas se tem que alterar a componente do *payload*.

3.3.1.3 Classe *Payload*

O *payload* não corresponde a uma só classe, mas sim a todas as mensagens que estão inseridas na biblioteca. Assim, cada uma das mensagens STANAG 4586 que está inserida nesta biblioteca (não estão inseridas todas uma vez que o objetivo do tradutor é converter mensagens de *waypoints* para o veículo) tem uma classe correspondente. Para esta investigação, as mensagens que mais utilizadas são: *STANAG_av_position_wp e STANAG_vehicle_operating_mode_command.* A primeira é responsável por enviar *waypoints* e a segunda por estabelecer o comando de voo (que pode ser, por exemplo, um comando de *Launch*).

Apesar de existir uma classe para cada mensagem, estas estão estruturadas de forma semelhante. Assim, o nome de cada classe é composto pelo prefixo STANAG seguido do nome da mensagem, como por exemplo *class STANAG_av_position_wp*. Cada classe contém certos parâmetros que permitem a sua caracterização, tais como o nome da mensagem, o tipo da mensagem, os campos que a compõem, formatos, entre outros.

Cada classe de mensagens é composta por duas funções, sendo que a primeira é responsável pela inicialização e a segunda pelo *pack* da mensagem. A maioria das mensagens possuem campos que necessitam de ser preenchidos. O número do *waypoint*, a sua longitude, latitude, altitude, tipo de localização ou velocidade que deve ser feita, são alguns dos parâmetros que, por exemplo, a mensagem *STANAG_av_position_wp* possui. No entanto, existem algumas mensagens que não necessitam do preenchimento de campos. Um exemplo disso é a mensagem *STANAG_heartbeat*, que permite perceber se o UAV ou a GCS estão operacionais. Tal como nas classes anteriores, esta classe também

tem uma função *pack*, que permite a codificação do *payload* para código binário através da função *struct.pack*.

Desta forma, o *payload* representa cada mensagem que está inserida nesta biblioteca. Cada mensagem tem uma classe associada, onde são inseridos os seus campos para posteriormente ser codificada. Após o *payload* ser codificado na função *pack*, é chamada a função *pack* da classe *STANAG_message*, onde vai ser feita a união do *header* com o *payload*, resultando na mensagem final.

3.3.1.4 Classe STANAG

As classes anteriores tinham a função de criar o *header* e o *payload*, juntando estes dois módulos de modo a criar a mensagem final. Por outro lado, a classe STANAG tem o objetivo de enviar e receber mensagens, preenchendo o *buffer*. Esta classe é composta por várias funções, sendo que as essenciais são as seguintes: função de inicialização, *encode*, *send*, *get_msg_type*, *get_buf* e *decode*.

Ao ser inicializada, a classe passa a deter um conjunto de propriedades, tais como um *buffer*, que corresponde a um *array* de bytes, o número total de mensagens enviadas e o número total de bytes enviados. À medida que as mensagens são enviadas, estes valores vão incrementando, sendo que o número total de bytes enviados depende do tipo de mensagem.

Para poder enviar as mensagens, é necessário chamar algumas funções das classes que foram expostas anteriormente. Deste modo, as funções *encode* e *send* são responsáveis por codificar a mensagem em código binário (chamando a função *pack* da classe *STANAG_message*), adicionando-a ao *buffer*. Tal como referido anteriormente, são ainda incrementados os respetivos valores de mensagens e bytes enviados.

Foram ainda criadas as funções *get_msg_type* e *get_buf*, cujo objetivo é retornar alguns valores. A função *get_msg_type* permite identificar o tipo de mensagem que está a ser recebida, a partir da descodificação do *header*. A descodificação é realizada através da utilização da função *struct.unpack*, que faz o trabalho inverso da função *struct.pack*, referida anteriormente. A função *get_msg_type* é fundamental, uma vez que a descodificação de uma mensagem depende do seu tipo, visto que diferentes mensagens

possuem diferentes campos e comprimentos. A função *get_buf* é simples e permite o retorno do *buffer*, de modo a que o utilizador possa obter estes dados quando desejar.

A função *decode* é uma das funções mais complexas da biblioteca e permite a passagem das mensagens no formato de código binário para caracteres, de forma a serem percetíveis por parte do utilizador. Deste modo, inicialmente é descodificado o *header*, o que permite obter o tipo da mensagem. Consequentemente, uma vez que já é conhecido o tipo da mensagem, é descodificado o *payload*. Depois da descodificação do *payload*, é apenas necessário juntar os dois módulos, *header* e *payload*, de modo a formar a mensagem que foi recebida. Ao longo desta função é ainda efetuado despiste de erros, como por exemplo, na descodificação do *header,* de modo a verificar se o valor do tipo da mensagem que foi obtido é válido ou não.

Em suma, foram explicadas as principais classes que compõem o protótipo da biblioteca de mensagens STANAG 4586 desenvolvido. A criação desta biblioteca foi um passo essencial para o desenvolvimento desta investigação, uma vez que permitiu a codificação das mensagens no formato correto. Desta forma, e em conjunto com bibliotecas já existentes do protocolo MAVLink, é então possível o desenvolvimento do tradutor pretendido.

3.3.2 Desenvolvimento do Tradutor

Os requisitos para a criação do tradutor passaram pela obtenção da biblioteca e GCS MAVLink, assim como pelo desenvolvimento da biblioteca de mensagens STANAG 4586. As bibliotecas de ambos os protocolos são essenciais para esta investigação, uma vez que permitem a utilização de mensagens com o formato MAVLink e STANAG 4586. Desta forma, é então possível iniciar o desenvolvimento do tradutor.

O objetivo do tradutor é fazer a ligação entre os protocolos STANAG 4586 e MAVLink. Especificamente para esta investigação, as mensagens a traduzir dizem respeito ao envio de *waypoints*, uma vez que se vai simular o envio dos mesmos a partir de uma estação de controlo STANAG 4586 para um veículo MAVLink. Desta forma, o *software* de tradução está ligado a ambas as bibliotecas.

Tal como referido anteriormente, no que diz respeito ao protocolo MAVLink, a

GCS a utilizar é o MAVProxy, que pode atuar paralelamente com outras estações de controlo no veículo, sendo que é uma grande vantagem. Outra vantagem é o facto de esta GCS fazer uso da biblioteca *pymavlink*, desenvolvida em *Python*. Adicionalmente, é ainda utilizada uma *Application Programming Interface* (API) que utiliza MAVLink, o *DroneKit-Python*.

Neste caso, a vantagem de utilizar uma API reside no facto de facilitar a utilização da biblioteca *pymavlink*, através de funções pré existentes. O *DroneKit-Python* permite o desenvolvimento de aplicações que sejam colocadas num *hardware*, sendo que este é ligado ao veículo. São várias as funções que esta API permite realizar, tais como a obtenção de parâmetros, estados do veículo ou o envio de comandos ao piloto automático. Assim como nesta investigação, existem vários outros projetos que também utilizam o conjunto de *software DroneKit-Python*, MAVProxy e *pymavlink*, uma vez que facilitam o desenvolvimento de aplicações para UAVs que utilizem MAVLink (Psirofonia, 2017).

O código de programação desenvolvido para o tradutor consta em Apêndice B. É formado por três módulos principais: a classe *Reception*, a classe *STANAG to MAVLink waypoint* (SM_WP) e a *Graphical User Interface* (GUI).

Como o nome indica, a classe *Reception* é responsável por receber todas as mensagens STANAG 4586 que cheguem ao veículo, reencaminhando-as para a função de conversão correta. Esta classe é composta por duas funções principais: a inicialização e *identification*. Assim como na biblioteca STANAG 4586, a função de inicialização contém uma contabilização dos bytes e do número de mensagens que são recebidas. A função *identification* é responsável por verificar o tipo da mensagem recebida e reencaminhá-la para a classe de tradução respetiva, que neste caso é a classe SM_WP. Nas suas funções, a classe *Reception* realiza ainda um controlo de erros simples, de modo a verificar se a mensagem recebida consta na lista de mensagens existentes na biblioteca e disponíveis para tradução.

A tradução é realizada na classe SM_WP. O processo é iniciado pela receção da mensagem de *waypoints*. Posteriormente os campos correspondentes aos valores de longitude, latitude e altitude são obtidos. No entanto, os *waypoints* STANAG 4586 são recebidos individualmente e são por isso guardados numa lista de comandos que o

DroneKit-Python possui. Assim, é necessário existir uma função responsável por enviar esta lista de comandos (composta por *waypoints*) para o veículo. Esta função é denominada *send_all*.

A GUI é a interface que simula o envio de mensagens STANAG 4586 por parte de um operador, para que possam ser convertidas em MAVLink pelo tradutor. Assim, é constituída por campos para introdução de posição, que é enviada através do botão *Introduce WP* (os tempos de cada introdução são apresentados na GUI). Após a introdução dos *waypoints* é esperado que o utilizador os envie para o veículo. Este procedimento é feito através do botão *Send All to UAV*. Se o utilizador necessitar de apagar todos os *waypoints* já introduzidos, ou apagar todos os *waypoints* que o veículo possui, deve pressionar o botão *Clear All*. Nesta investigação foi também introduzido o botão *Start Mission*. Quando o botão é premido, é enviada uma mensagem *STANAG_vehicle_operating_mode_command,* que estabelece comando de voo. Um dos campos desta mensagem é preenchido com o valor 21, que corresponde ao *launch*. Quando o tradutor recebe esta mensagem, realiza os comandos necessários através do *DroneKit-Python* para que o veículo dê início à missão com os *waypoints* estabelecidos. Finalmente, o botão *Load WPs from UAV* permite ao utilizador a receção dos *waypoints* que o veículo possui, escrevendo-os num ficheiro .txt denominado *mission_load*. Após isso os *waypoints* recebidos são ainda traduzidos para o formato de mensagem STANAG 4586 e os seus valores de latitude, longitude e altitude são apresentados na GUI, que está representada na Figura 24.

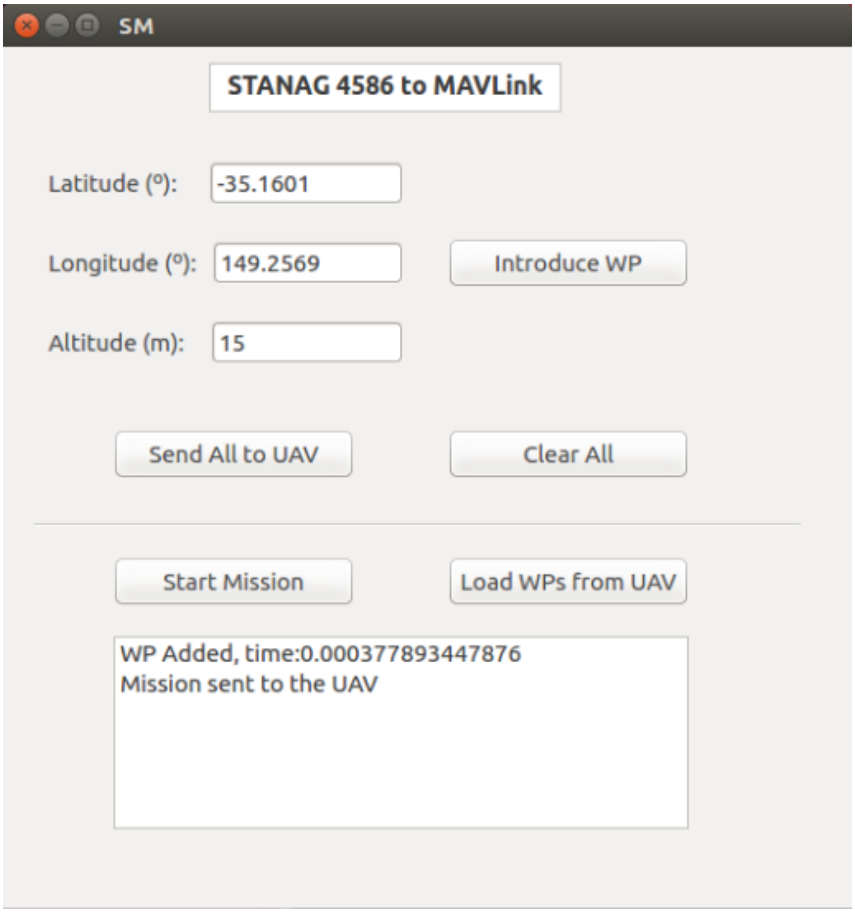

Figura 24 - GUI a simular uma GCS STANAG 4586

Em suma, foi realizado um estudo onde se concluiu que a opção mais viável seria a colocação do tradutor no veículo. Posteriormente, foi escolhido o *hardware* a utilizar, o *Raspberry Pi*. Este *hardware* possui um conjunto de vantagens, como a fácil inserção no veículo, assim como a compatibilidade do seu sistema operativo com a linguagem de programação *Python*. Em seguida, foi desenvolvida uma biblioteca do *standard* STANAG 4586 e foram escolhidas as aplicações de MAVLink a utilizar. O passo final consistiu na criação do tradutor.

Desta forma, é pretendido que o operador introduza *waypoints* numa estação de controlo STANAG 4586, que é simulada pela biblioteca e GUI desenvolvidas. Na biblioteca, os *waypoints* são convertidos para o formato STANAG 4586, onde são codificados por forma a serem enviados para o *Raspberry Pi*, que estará ligado ao UAV. No *Raspberry Pi* é feita a tradução de modo a que a mensagem fique no formato MAVLink. Por fim, os *waypoints* são passados para o UAV. O trabalho efetuado permite

ainda o início da missão e a receção dos *waypoints* que o veículo detém.

3.3.2.1 Limitações do tradutor

O tradutor desenvolvido conta ainda com algumas limitações que devem ser referidas. Como referido anteriormente, foram traduzidas algumas mensagens STANAG 4586, incluindo uma mensagem destinada ao envio de *waypoints*. Este *standard* não tem apenas uma mensagem para este fim, sendo que conta com mais, dependendo do objetivo do veículo e da *payload*. Contudo, o objetivo desta dissertação foi estudar a viabilidade do tradutor, que foi conseguido através das mensagens utilizadas.

Adicionalmente, uma mensagem STANAG 4586 tem mais campos do que uma mensagem MAVLink, como se demonstrou nas Tabelas 5 e 6. Assim, com a execução do tradutor pode ocorrer a perda de alguns dados. Finalmente, o tradutor desenvolvido não conta com formas de verificar a receção da mensagem MAVLink.

4. Validação do Modelo e Análise de Resultados

4.1 Validação do Modelo

4.2 Análise de Resultados

Tal como referido anteriormente, um trabalho de investigação consiste na execução de várias etapas. Após o desenvolvimento e implementação do modelo, torna-se necessário verificar a sua viabilidade. Para tal, a realização de testes e interpretação de resultados é fundamental. Assim sendo, este capítulo está dividido em dois subcapítulos principais: validação do modelo e análise de resultados.

4.1 Validação do Modelo

A validação do modelo desenvolvido é uma etapa fundamental para qualquer investigação. O objetivo do tradutor é converter mensagens de *waypoints*. Estas mensagens serão enviadas por uma estação de controlo que opere com o *standard* STANAG 4586. Desta forma, a função do tradutor é fazer com que estas sejam recebidas e tratadas por um veículo que opere com o protocolo MAVLink.

Neste subcapítulo estão descritos os testes que irão avaliar o programa concebido. Desta forma, serão criados dois cenários distintos:

1. Validação do tradutor no computador, com o objetivo principal de verificar a receção dos *waypoints* num veículo simulado;
2. Validação do tradutor no *Raspberry Pi*, com o objetivo principal de verificar os tempos de execução do programa.

As validações acima mencionadas serão realizadas com recurso a aplicações e programas de simulação. De modo a que os testes sejam o mais próximo possível da realidade, é necessário escolher os *softwares* certos. Desta forma, é necessário possuir os seguintes elementos:

- Estação de controlo com o *standard* STANAG 4586;
- *Software* de simulação de um veículo autónomo;
- Estação de controlo com o protocolo MAVLink.

O objetivo de possuir uma estação de controlo com o *standard* STANAG 4586 é poder enviar mensagens de *waypoints* nesse mesmo formato. A estação de controlo deverá conseguir recolher a informação do operador (que neste caso são os *waypoints*), formatá-la de acordo com as especificações do *standard*, codificar a mensagem em código

binário e enviá-la para o veículo. Esta estação de controlo é conseguida através da biblioteca desenvolvida no capítulo anterior, uma vez que é capaz de cumprir com todos os requisitos referidos.

O *software* de simulação a utilizar será o SITL. Este *software* permite a simulação de veículos, como por exemplo, uma aeronave de asa fixa, asa rotativa, ou ainda veículos terrestes, sem qualquer *hardware*. O SITL é *open-source* e foi criado através da compilação de código utilizado no *ArduPilot*, um piloto automático que utiliza o protocolo MAVLink e que tem a vantagem de poder ser empregue na maioria das plataformas. Assim, o SITL representa um simulador de qualidade, bastante utilizado por vários investigadores na área da robótica móvel.

A obtenção de uma estação de controlo com o protocolo MAVLink trará a vantagem de se poderem confirmar os dados que foram convertidos. Desta forma, a estação de controlo a utilizar será a *APM Planner* 2.0, uma vez que utiliza o MAVLink e é *open-source*.

4.1.1 Validação no Computador

Os primeiros testes serão realizados em computador e serão os mais simples. Os objetivos desta validação inicial estão enumerados de seguida, por ordem crescente de importância:

1. Verificar a receção dos *waypoints* pelo veículo simulado;
2. Verificar os *waypoints* recebidos e a existência de disparidades nos dados;
3. Verificar o tempo de execução do código do tradutor.

O objetivo principal será a determinação da receção dos *waypoints* no veículo simulado. Este objetivo permitirá avaliar o tradutor, uma vez que verificará se este realiza a conversão de forma correta ou não. O segundo objetivo será a comparação de valores dos *waypoints* recebidos com os *waypoints* transmitidos, de modo a perceber se existem disparidades entre estes dados. Por fim, o terceiro objetivo é a verificação do tempo de execução do código do tradutor. Esta recolha de tempos permitirá obter uma primeira avaliação, de modo a entender se o tradutor afetará ou não as comunicações. No entanto, não deverá ser tomada como o tempo de execução padrão, uma vez que esta primeira

avaliação será feita num computador de uso normal e não no *Raspberry Pi, hardware* onde será implementado o tradutor.

Após a enumeração dos objetivos, é necessário definir os requisitos necessários para que a validação do tradutor seja possível. Assim sendo, os requisitos são os seguintes:

- Obter todos os elementos de *software* que foram descritos no subcapítulo 4.1, nomeadamente a estação de controlo com o *standard* STANAG 4586, o simulador SITL e a estação de controlo com o protocolo MAVLink;
- Introduzir o código de programação necessário no tradutor de modo a determinar o seu tempo de execução.

Assim sendo, o primeiro cenário de validação será realizado em computador. Este cenário pode ser dividido em quatro passos. O passo inicial ocorre na estação de controlo que opera com o *standard* STANAG 4586 e consiste na introdução do *waypoint* por parte do utilizador (a partir da GUI que foi desenvolvida). Na estação de controlo, o *waypoint* é transformado numa mensagem formatada, que é codificada em código binário e enviada para o tradutor (ligado ao veículo simulado pelo SITL). O segundo passo é realizado no tradutor, onde a mensagem, que está formatada em STANAG 4586, é recebida e os seus parâmetros são convertidos, originando uma mensagem MAVLink que é enviada para o veículo. O terceiro passo ocorre no veículo simulado, que recebe a mensagem MAVLink contendo o *waypoint*. Finalmente, o quarto passo consiste na verificação do *waypoint* recebido. Esta verificação é realizada no *APM Planner* 2.0. A Figura 25 representa o cenário desenvolvido, assim como os passos descritos.

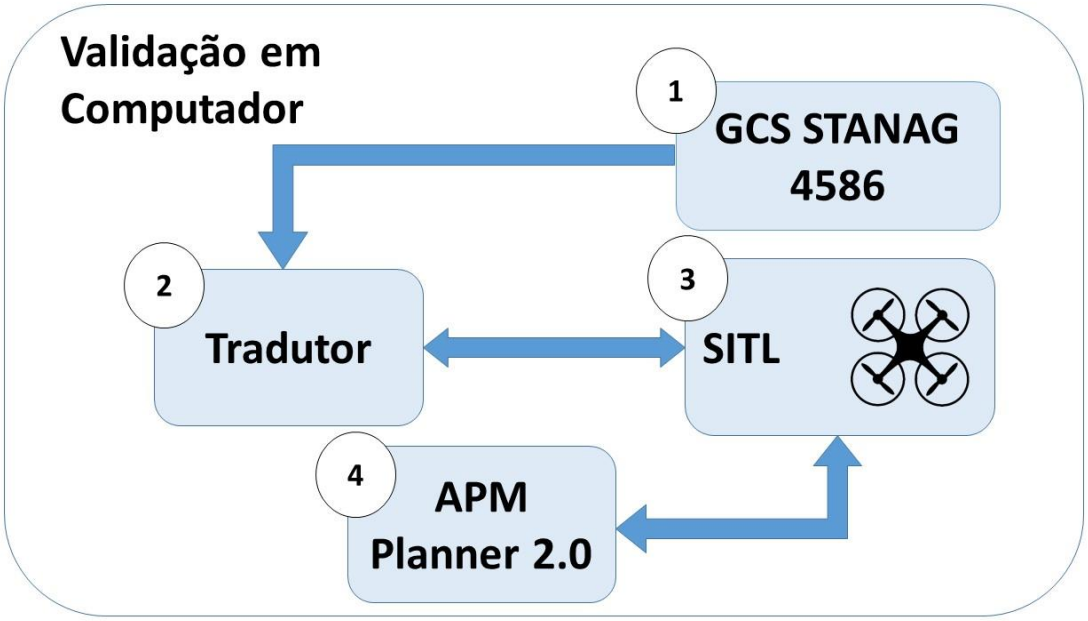

Figura 25 - Cenário de validação em computador

4.1.2 Validação no *Raspberry Pi*

O *hardware* escolhido para ser utilizado foi o *Raspberry Pi*. Assim, é imprescindível a realização de testes com este componente. Os objetivos a atingir com este cenário serão os seguintes, por ordem crescente de importância:

1. Verificar o tempo de execução do código do tradutor;
2. Verificar os *waypoints* recebidos e a existência de disparidades nos dados;

O objetivo principal do segundo cenário de validação consiste na verificação do tempo de execução do código do tradutor. Este é o principal objetivo, uma vez que é no *Raspberry Pi* que o tradutor será executado e os tempos de execução obtidos serão os mais próximos da realidade. Deste modo, é importante verificar se o tempo de processamento do código não afeta as comunicações entre a estação de controlo e o UAV. O segundo objetivo, assim como no primeiro cenário de validação, é a comparação dos *waypoints* enviados com os recebidos, de modo a identificar disparidades nos dados.

Após a identificação dos objetivos, é possível definir alguns requisitos para a validação do segundo cenário:

- É necessário preparar o *Raspberry Pi*, configurando-o e instalando todo o *software* necessário, incluindo o próprio tradutor;
- Introduzir o código de programação necessário no tradutor de modo a determinar o seu tempo de execução.

Os passos descritos para a realização do cenário são idênticos aos da Figura 25, mas desta vez no *Raspberry Pi*. É expectável que os resultados do tempo de execução neste *hardware* sejam superiores aos obtidos no cenário de validação em computador, uma vez que a capacidade de processamento do *Raspberry Pi* é normalmente inferior à capacidade de um computador doméstico.

4.2 Análise de Resultados

Após a criação dos cenários de validação, é importante analisar os resultados obtidos. Esta análise é fundamental para avaliar o desempenho do tradutor. Assim sendo, vai permitir perceber se o modelo desenvolvido consegue fazer a tradução entre os dois protocolos, de modo a que não afete as comunicações entre o veículo e a estação de controlo. De modo a comparar o tradutor desenvolvido com um sistema semelhante, são ainda realizadas comparações dos resultados obtidos nesta investigação com os resultados de um *software* MAVROS.

Este subcapítulo está dividido em: análise de resultados na validação em computador e análise de resultados na validação com o *Raspberry Pi*.

4.2.1 Análise de resultados na validação em computador

Conforme descrito anteriormente, o principal objetivo do primeiro cenário de validação é a verificação da receção dos *waypoints* no veículo. Este objetivo é verificado através da utilização do *APM Planner 2.0*. Este *software* está ligado ao SITL e permite obter os *waypoints* que são enviados ao veículo.

Desta forma, foi efetuada a tradução de 10 *waypoints*. Na Figura 26 está representado o *APM Planner* 2.0, que utiliza MAVLink, acusando a receção de todos os *waypoints* enviados.

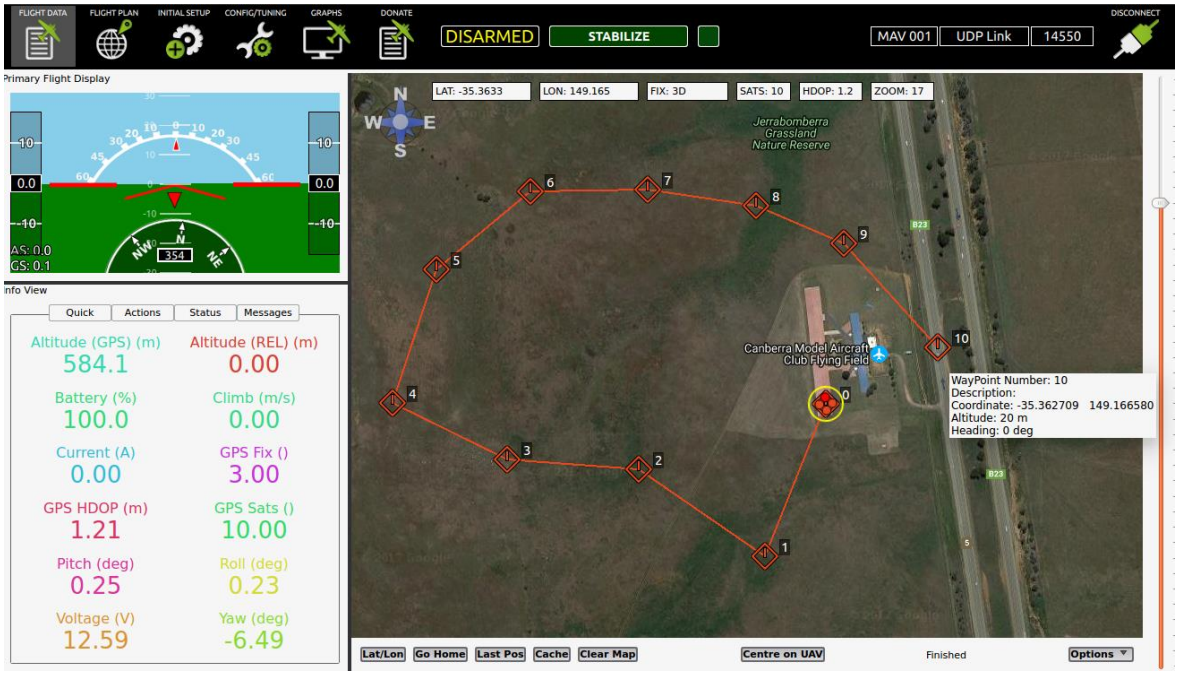

Figura 26 - APM Planner acusando receção dos waypoints

O segundo objetivo da validação em computador é a identificação de disparidades nos dados dos *waypoints*. Na Tabela 8 estão representados os *waypoints* que foram inseridos pela estação de controlo STANAG 4586, de modo a serem traduzidos para MAVLink.

Tabela 8 - Waypoints enviados pela GCS STANAG 4586

Waypoints	Latitude (º)	Longitude (º)	Altitude (m)
1	-35,36475	149,16450	20,0
2	-35,36390	149,16300	15,0
3	-35,36380	149,16143	20,0
4	-35,36325	149,16005	20,0
5	-35,36195	149,16057	25,0
6	-35,36119	149,16170	20,0
7	-35,36117	149,16310	20,0
8	-35,36133	149,16440	15,0
9	-35,36170	149,16545	20,0
10	-35,36271	149,16658	20,0

A verificação dos dados referentes aos *waypoints* recebidos pode também ser obtida no *APM Planner* 2.0, conforme representado na Figura 27.

Figura 27- Valores dos waypoints recebidos no APM Planner

Os *waypoints* referidos na Figura anterior estão também apresentados na Tabela 9. Adicionalmente, são apresentadas as diferenças entre os *waypoints* enviados e os recebidos, indicando a média do erro.

Tabela 9 - Disparidades entre waypoints enviados e recebidos no cenário em computador

Latitude (º)	Longitude (º)	Altitude (m)	Diferença Lat (º)	Diferença Lon (º)	Diferença Alt (m)
-35,364749	149,164505	20,0	0,000001	0,000005	0,0
-35,363903	149,162994	15,0	0,000003	0,000006	0,0
-35,363804	149,161423	20,0	0,000004	0,000007	0,0
-35,363251	149,160049	20,0	0,000001	0,000001	0,0
-35,36195	149,160568	25,0	0,000000	0,000001	0,0
-35,361191	149,161697	20,0	0,000001	0,000003	0,0
-35,361172	149,163101	20,0	0,000002	0,000001	0,0
-35,361328	149,164398	15,0	0,000002	0,000002	0,0
-35,361698	149,165451	20,0	0,000002	0,000001	0,0
-35,362709	149,16658	20,0	0,000001	0,000000	0,0
		Média =	0,0000017	0,0000027	0,0

As diferenças entre os valores enviados e recebidos foram de $1{,}7\text{x}10^{-6}$ (º) na Latitude e $2{,}7\text{x}10^{-6}$ (º) na Longitude. De modo a tornar esta variação entre posição enviada e recebida mais percetível, é possível transformar os valores obtidos em graus para metros, sabendo que 1 minuto corresponde a 1 milha náutica. Assim, a diferença de latitude origina um erro de posição de aproximadamente 19 cm e. Os resultados obtidos

provam que a tradução realizada não afeta os valores de posição de forma significativa. Desta forma, o segundo objetivo de validação é cumprido de forma viável.

O terceiro objetivo da validação em computador é a obtenção dos tempos de execução do código obtido. Estes tempos de execução são obtidos em dois momentos. O primeiro momento é a tradução do *waypoint* com o formato STANAG 4586 para o formato MAVLink. O segundo momento ocorre após a tradução e consiste no envio do *waypoint* para o veículo. A Tabela 10 apresenta os resultados obtidos em ambos os tempos de execução.

Tabela 10 - Resultados dos tempos de execução em computador

Waypoints	Tempo de tradução (ms)	Tempo de envio (ms)
1	0,1100	200,9150
2	0,1300	201,0830
3	0,1500	200,9430
4	0,2100	200,8450
5	0,1600	200,8710
6	0,1370	200,6390
7	0,1260	200,8130
8	0,1270	201,1220
9	0,1000	200,9290
10	0,0800	200,3290
Média =	0,1330	200,8489

Em termos de requisitos operacionais, o *standard* STANAG 4586 indica um valor máximo de latência de 500 milissegundos para a mensagem de *waypoints*. Esse valor não se aplica diretamente neste caso, porque após a tradução, o envio da mensagem é realizado com o protocolo MAVLink. Contudo, o valor de 500 milissegundos pode ser levado em conta como um valor de referência para este processo.

Os resultados do tempo de tradução apresentam uma média de 0,133 milissegundos por *waypoint*. Estes valores são bastante aceitáveis, uma vez que não são significativos comparativamente com os valores de envio. Relativamente aos valores do tempo de envio, a média perfaz um valor de aproximadamente 200,85 milissegundos. Este valor é viável, uma vez que é menor do que o valor de referência estabelecido pelo STANAG 4586.

Desta forma, é possível concluir que o tradutor não afeta as comunicações. Contudo, estes resultados foram obtidos no cenário de validação em computador, sendo que os tempos de execução padrão devem ser os encontrados a partir do *Raspberry Pi*.

4.2.2 Análise de resultados na validação com o *Raspberry Pi*

Os resultados no cenário com o *Raspberry Pi* são fundamentais, uma vez que permitirão analisar a viabilidade do tradutor realizado no *hardware* escolhido. Assim, o principal objetivo desta validação é a recolha dos tempos de execução do código do tradutor. Tal como no cenário de validação anterior, os tempos de execução são recolhidos em dois momentos diferentes: a tradução do *waypoint* com o formato STANAG 4586 para MAVLink; o envio do *waypoint* em MAVLink para o veículo. Os resultados obtidos estão apresentados na Tabela 11.

Tabela 11 - Tempos de execução obtidos no Raspberry Pi

Waypoints	Tempo de tradução (ms)	Tempo de envio (ms)	Tempo de execução Pi (ms)
1	1,2100	206,7190	207,9290
2	0,9100	203,8230	204,7330
3	1,5700	205,4140	206,9840
4	2,3100	202,3410	204,6510
5	1,2500	201,3710	202,6210
6	1,3500	207,3590	208,7090
7	1,2900	201,1310	202,4210
8	1,3400	206,9190	208,2590
9	2,1500	202,2860	204,4360
10	2,1900	209,2910	211,4810
Média =	1,5570	204,6654	206,2224

Era esperado que existisse um aumento nos tempos de execução do *Raspberry Pi* em comparação com os obtidos em computador. Assim, conforme expresso na Tabela 11, a média dos tempos de tradução obtidos foi de 1,557 milissegundos, representando um aumento de 1,424 milissegundos comparativamente com o cenário em computador. Relativamente aos tempos de envio, a média obtida foi de 204,6654 milissegundos, representando um aumento de 3,8165 milissegundos. Assim, o tempo de execução total no *Raspberry Pi* pode ser visto como a soma dos valores de tradução e envio, resultando numa média de 206,2224 milissegundos.

Os valores obtidos em *Raspberry Pi* estão também abaixo do limite máximo de latência expresso no *standard* STANAG 4586. Desta forma, é possível afirmar o sistema desenvolvido nesta investigação não afeta as comunicações entre um UAV e a sua estação de controlo.

Assim como no primeiro cenário de validação, o segundo objetivo é a verificação dos *waypoints* recebidos e respetiva verificação de disparidades nos dados. Os *waypoints* enviados com o formato STANAG 4586 foram iguais aos enviados no cenário anterior (Tabela 8). A Tabela 12 expressa os valores recebidos, indicando a respetiva diferença com os enviados.

Tabela 12 - Disparidades entre waypoints enviados e recebidos no cenário em Raspberry Pi

Latitude (º)	Longitude (º)	Altitude (m)	Diferença Lat (º)	Diferença Lon (º)	Diferença Alt (m)
-35,364748	149,164504	20,0	0,000002	0,000004	0,0
-35,363902	149,163001	15,0	0,000002	0,000001	0,0
-35,363799	149,161427	20,0	0,000001	0,000003	0,0
-35,363253	149,160049	20,0	0,000003	0,000001	0,0
-35,361952	149,160568	25,0	0,000002	0,000001	0,0
-35,361194	149,161699	20,0	0,000004	0,000001	0,0
-35,361173	149,163103	20,0	0,000003	0,000003	0,0
-35,361327	149,164398	15,0	0,000002	0,000002	0,0
-35,361699	149,165452	20,0	0,000001	0,000002	0,0
-35,362711	149,166581	20,0	0,000001	0,000001	0,0
		Média =	0,0000021	0,0000019	0,0

Após a análise dos dados obtidos na Tabela 12, é possível verificar que a diferença de posição quando à latitude foi de $2{,}1\text{x}10^{-6}$ (º), enquanto que a diferença de longitude foi de $1{,}9\text{x}10^{-6}$ (º). Estes valores dão um erro de posição de 23cm na latitude. Assim sendo, tal como no cenário anterior, os resultados obtidos provam que a disparidade entre os valores enviados e recebidos não é significativa.

4.2.3 Comparação com o *software* MAVROS

O *software* MAVROS foi apresentado no Capítulo 1, e assim como o tradutor desenvolvido ao longo desta investigação, pode ser visto com um meio de converter dados entre dois protocolos diferentes. Mais concretamente, o MAVROS faz a tradução entre o

framework ROS e o protocolo MAVLink. Assim sendo, este *software* pode ser utilizado como termo de comparação com o tradutor desenvolvido.

Desta forma, o objetivo é obter os tempos de execução que o MAVROS tem para realizar a tradução e envio dos *waypoints*. Foi utilizado o *package mavwp*, que está integrado no MAVROS, onde se introduzem os *waypoints* a partir de um ficheiro txt. Para que se obtenham os resultados desta tradução, foi necessário inserir algumas linhas de código no package (foi utilizado o mesmo método que efetuou a medição de tempo dos cenários anteriores), de modo a que este indique o tempo de execução.

Assim foram realizados 10 testes, sendo que foram utilizados os *waypoints* das traduções anteriores. Os resultados do MAVROS estão apresentados na Tabela 13.

Tabela 13 - Tempos de execução obtidos no MAVROS

Waypoints	Tempo de execução MAVROS (ms)
1	193,0550
2	204,7310
3	224,7090
4	214,1080
5	234,4530
6	237,0070
7	191,8210
8	231,1860
9	150,0790
10	194,2870
Média =	207,5436

Os resultados obtidos com o MAVROS obtiveram uma média de 207,5436 milissegundos. No entanto, estes valores incluem o tempo de tradução e de envio. Através da soma dos valores do tempo de introdução e de envio do tradutor realizada anteriormente, obteve-se o valor médio de 206,2224 milissegundos. É então possível verificar que os valores do tempo de execução são similares aos obtidos com o tradutor desenvolvido nesta investigação.

Na Figura 28 está apresentado o gráfico com a comparação entre os tempos de execução do MAVROS e do tradutor no *Raspberry Pi*. Como referido anteriormente, os tempos de execução são parecidos, sendo que os tempos do tradutor desenvolvido mantiveram um idêntico ao longo dos 10 *waypoints*.

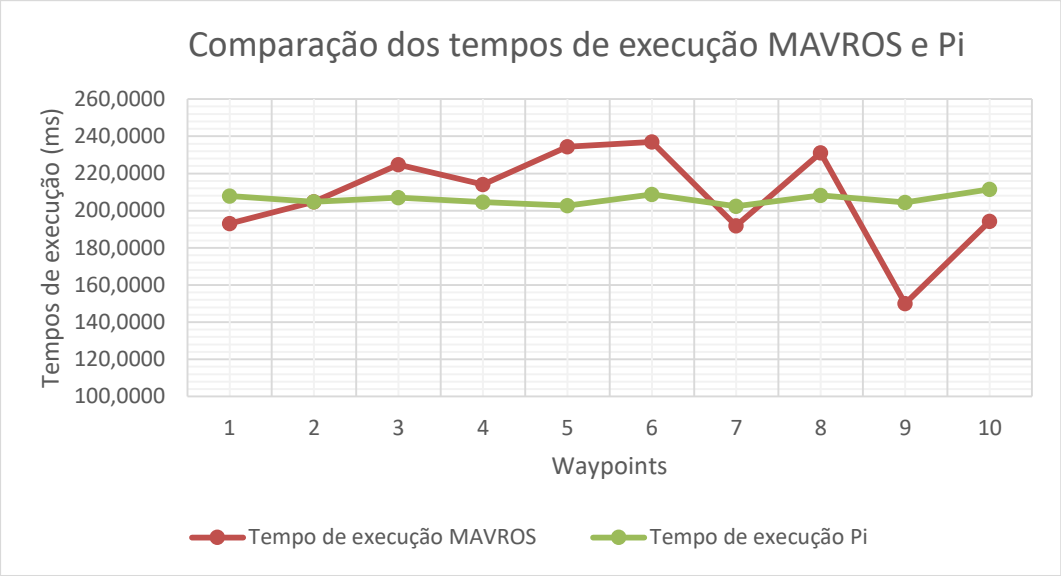

Figura 28 - Comparação de tempos de execução entre o MAVROS e o tradutor no Raspberry Pi

Em suma, é possível afirmar que o tradutor desenvolvido é viável, uma vez que apresenta tempos de execução inferiores ao limite do *standard* STANAG 4586. Adicionalmente, foram ainda comparados os valores do tradutor com os de outro *software* de tradução entre protocolos, neste caso, entre MAVLink e ROS. O MAVROS é um pacote do ROS e é bastante utilizado na comunidade da robótica móvel. A comparação dos tempos de execução entre o tradutor e o MAVROS demonstram que os valores são bastante próximos, comprovando assim a viabilidade do sistema desenvolvido.

Conclusões

Síntese do trabalho efetuado

Relevância dos Resultados

Trabalhos Futuros

Síntese do trabalho efetuado

A presente dissertação descreve o desenvolvimento de um sistema que visa realizar a tradução entre dois protocolos de comunicação bastante utilizados: STANAG 4586 e MAVLink. O principal objetivo do sistema criado foi responder à falta de padronização existente nesta área. Para tal, foi proposto um cenário que consistiu no envio de mensagens de *waypoints* a partir de uma estação de controlo STANAG 4586 para um UAV MAVLink. Os resultados obtidos provaram que foi possível atingir interoperabilidade com a utilização do tradutor desenvolvido.

Esta dissertação contou com uma revisão da literatura que permitiu obter os mais recentes avanços sobre esta área científica. Desta forma, foram realizados estudos sobre a arquitetura de um SANT, assim como sobre os vários protocolos que estabelecem a comunicação entre um UAV e a sua estação de controlo. Adicionalmente, foram ainda realizados estudos sobre outros tradutores já concebidos anteriormente.

O segundo capítulo definiu as metodologias de investigação utilizadas. A realização de uma investigação deve ser apoiada através de métodos, que permitem a sua estruturação de forma organizada e lógica. Assim, foram estabelecidas as várias etapas da dissertação.

A proposta de arquitetura do modelo foi apresentada no terceiro capítulo. Foram apresentadas duas hipóteses para a formulação do tradutor, assim como as suas respetivas vantagens e desvantagens. Através de uma análise cuidada das mesmas, foi escolhida a opção considerada como mais viável. A opção escolhida consiste na introdução do tradutor desenvolvido no veículo, que permite simultaneamente a existência de comunicação entre a estação de controlo e outros veículos que operem com o STANAG 4586, possibilitando a solução do problema. Este capítulo contou ainda com a escolha do *hardware* a utilizar e com o desenvolvimento do software, incluindo um protótipo de biblioteca de mensagens STANAG 4586, o próprio tradutor e a GUI para a realização de testes.

No quarto capítulo foi realizada a validação e discussão de resultados. Deste modo, foram estabelecidos inicialmente os cenários de validação propostos, tendo sido

definidos os objetivos e requisitos para cada um. Posto isto, foi realizada a obtenção dos vários resultados, incluindo tempos de execução do código e comparações entre os *waypoints* enviados e recebidos. Finalmente, foram comparados os valores obtidos no tradutor com os obtidos num outro sistema de tradução já existente, o MAVROS.

Relevância dos Resultados

O problema abordado no segundo capítulo, Metodologias de Investigação, consistia na falta de interoperabilidade existente entre uma estação de controlo a utilizar STANAG 4586 e um veículo a utilizar MAVLink. Para tal, foram estudados vários cenários possíveis para solucionar o problema, tendo sido escolhido o que foi considerado mais viável. Desta forma, a escolha passou pela introdução do tradutor no UAV, de modo a permitir a tradução com o protocolo MAVLink, permitindo simultaneamente a comunicação com outros veículos STANAG 4586.

Apesar de não terem sido realizados testes num veículo real, os resultados obtidos em simulador demonstram que a utilização do tradutor é viável, uma vez que não afeta as comunicações existentes entre um UAV e a sua estação de controlo. Os tempos de execução foram bastante próximos aos obtidos com um *software* similar ao desenvolvido, o MAVROS, que efetua a tradução entre o ROS e o MAVLink.

Assim, a investigação desenvolvida prova que é possível solucionar o problema encontrado através da criação de um tradutor. A utilização de um tradutor deste tipo permitirá a compatibilidade de UAVs que operem com MAVLink em cenários de operações NATO, ou seja, com UAVs e estações de controlo que utilizem STANAG 4586.

A grande vantagem de tornar um veículo MAVLink compatível com o *standard* STANAG 4586 é a diminuição de custos. Caso este não existisse, seria necessário reestruturar ou adquirir um veículo que utilizasse STANAG 4586, o que é bastante dispendioso. Deste modo, é possível adquirir facilmente veículos que utilizem MAVLink (a um custo mais reduzido quando em comparação com os STANAG 4586), uma vez que existem muitos pilotos automáticos a utilizar este protocolo e inserir um tradutor.

Trabalhos Futuros

Relativamente a trabalhos futuros, seria interessante completar o tradutor e a biblioteca de mensagens STANAG 4586, uma vez que para esta investigação foram apenas abordadas mensagens de *waypoints* e de comandos de voo. Nesta investigação não foi realizada a tradução completa porque este *standard* é bastante vasto e completo.

São ainda propostos tradutores entre outros protocolos de comunicação. Por exemplo, entre JAUS e MAVLink ou entre JAUS e STANAG 4586. O JAUS é também um protocolo de comunicação bastante completo e pode ser utilizado com todos os tipos de veículos autónomos (aéreos, marítimos e terrestres). Assim, a conceção de um tradutor deste tipo seria também um processo interessante.

Referências Bibliográficas

ALKIRE, B., et al. (2010), *Applications for Navy Unmanned Aircraft Systems*, Pittsburgh, RAND Corporation.

AUSTIN, R. (2010), *Unmanned Aircraft Systems*, 1ª ed., Chichester, Wiley.

BALAJI, S. e MURUGAIYAN, M. S. (2012), "Waterfall vs v-model vs agile : A comparative study on SDLC", in *International Journal of Information Technology and Business Management*, vol. 2, pp. 1–5.

COFFEY, T. e MONTGOMERY, J. A. (2002), "The emergence of mini UAVs for military applications", in *Defense Horizons*, pp. 1–8.

COOK, K. (2007), "The silent force multiplier: The History and Role of UAVs in Warfare" in *IEEE Aerospace Conference Proceedings*, Big Sky, pp. 1–7.

COOMBES, M., et al. (2012), "Development of an autopilot system for rapid prototyping of high level control algorithms", in *Proceedings of 2012 UKACC International Conference on Control*, pp. 1–5.

CRAWFORD, S. E STUCKI, L. (1990), "Peer Review and the Changing Research Record", in *Journal of the American Society for Information and Science*, pp. 1–6.

DAVID, G. (2000), "Unmanned Aerial Vehicles: Implications for Military Operations", *Center for Strategy and Technology Air War College*, Alabama, pp. 1–27.

ERIKSSON, M., e RINGMAN, P. (2013), *Launch and recovery systems for unmanned vehicles onboard ships. A study and initial concepts*, Tese de Mestrado apresentada no Centre for Naval Architecture, Estocolmo.

EUROPEAN AVIATION SAFETY AGENCY (2009), *Airworthiness Certification of Unmanned Aircraft Systems Policy Statement* (UAS).

FAHLSTROM, P. G. e GLEASON, T. J. (2012), *Introduction to UAV Systems*, 4ª ed., Wiley, Chichester.

GUPTA, S., GHONGE, M. e JAWANDHIYA, P. (2013), "Review of Unmanned Aircraft System", in *International Journal of Advanced Research in Computer Engineering &*

Technology, vol. 2, pp. 1–13.

IEEE. (1990), *IEEE Standard Glossary of Software Engineering Terminology*. *Office*, Standards Coordinating Comitee of the Computer Society of IEEE, Nova Iorque.

JOVANOVIC, M. e STARCEVIC, D. (2008), "Software architecture for ground control station for unmanned aerial vehicle", *Tenth International Conference on Computer Modeling and Simulation*, Cambridge, pp. 1–5.

MARQUES, M. M., et al. (2015), "Unmanned aircraft systems in maritime operations: Challenges addressed in the scope of the SEAGULL project", in *MTS/IEEE OCEANS 2015 - Genova*, Genova, pp. 1-6.

MARQUES, M. M., et al. (2016), "Use of multi-domain robots in search and rescue operations - Contributions of the ICARUS team to the euRathlon 2015 challenge", in *MTS/IEEE OCEANS 2016 - Shanghai*, Xangai, pp.1-7.

MARTY, J. A. (2013), *Vulnerability Analysis of the MAVLink Protocol for Command and Control of Unmanned Aircraft*, Tese de Mestrado apresentada no Air Force Institute of Technology, Ohio.

MARZOCCHI, O. (2015), *Privacy and Data Protection Implications of the Civil Use of Drones*.

MEIER, L., et al. (2011), "PIXHAWK: A system for autonomous flight using onboard computer vision", in *Proceedings - IEEE International Conference on Robotics and Automation*, Zurique, pp. 1-6.

NATO (2006), *Interoperability for joint operations*, NATO Public Diplomacy Division, Bruxelas.

NATO STANDARDIZATION AGENCY (2012), *STANAG 4586 - Standard Interfaces of UAV Control System for NATO UAV Interoperability*, 3ª ed., Bruxelas.

NEHMZOW, U. (2009), *Robot Behaviour: Design, Description, Analysis and Modelling*, 1ª ed., Londres, Springer.

NONAMI, K., et al. (2010), *Autonomous Flying Robots*, 1ª ed., Londres, Springer.

PEREIRA, S. (2014), *Propagação e Radiação de Ondas Eletromagnéticas em Ambientes Urbanos*, Tese de Mestrado apresentada no Instituto Superior Técnico, Lisboa.

POOLE, M. (2015), *Building a home security system with OpenKinect*, 1ª ed., Birmingham, Packt Publishing.

PSIROFONIA, P., et al. (2017), "Use of Unmanned Aerial Vehicles for Agricultural Applications with Emphasis on Crop Protection: Three Novel Case - studies", in *International Journal of Agricultural Science and Technology*, vol. 5, pp.1-10.

QGROUNDCONTROL [s.d.], *MAVLink Protocol*, http://www.qgroundcontrol.org/mavlink/start, consultado em dezembro de 2016.

QUIGLEY, M., et al. (2009), "ROS: an open-source Robot Operating System", in *IEEE International Conference on Robotics and Automation Workshop on Open Source Software*, Kobe, pp. 1-6.

QUIGLEY, M., GERKEY, B., e SMART, W. D. (2015), *Programming Robots with ROS*, 1ª ed., Sebastopol, O'Reilly Media.

RAI, N., e RAI, B. (2013), "Neural Network based Closed loop Speed Control of DC Motor using Arduino Uno", in *International Journal of Engineering Trends and Technology*, vol. 4, pp. 1–4.

RICKLIN, J. C., e DAVIDSON, F. M. (2002), "Atmospheric turbulence effects on a partially coherent Gaussian beam: implications for free-space laser communication", in *Journal of the Optical Society of America*, vol. 19, pp. 1–9.

SEGOR, F., et al. (2010), "Mobile ground control station for local surveillance", *The Fifth International Conference on Systems and Networks Communications*, Nice, pp. 1–6.

SESAR (2016), *European Drones Outlook Study*, SESAR Joint Undertaking, pp. 1-93.

SILVEIRA, R. M. C. e LEITE, S. DE L. (2016), "Sistema de Controle de Acesso Baseado na Plataforma NodeMCU", in *Jornada de Informática Do Maranhão*, São Luís, pp. 1–6.

SKRZYPIETZ, B. T. (2012), "Unmanned Aircraft Systems for Civilian Missions", in *Brandenburg Institute for Society and Security Policy Paper*, vol. 1, pp. 1–28.

SMITH, D. T., et al. (2012), *Unmanned Vehicle Message Conversion System*, patente Estados Unidos da América, Washington D.C.

STANOVOV, V. e SEMENKIN, E. (2016), "Streaming Pulse data to the Cloud with Bluetooth LE or NODEMCU ESP8266", in *5th Mediterranean Conference on Embedded Computing*, Bar, pp. 1–4.

STANSBURY, R. S., VYAS, M. A. e WILSON, T. A. (2009), "A survey of UAS technologies for command, control, and communication (C3)", in *Journal of Intelligent and Robotic Systems: Theory and Applications*, pp. 1–18.

VALAVANIS, K. P., VACHTSEVANOS, G. J. e ANTSAKLIS, P. J. (2007), "Technology and Autonomous Mechanisms in the Mediterranean: From Ancient Greece to Byzantium", in *Proceedings of the European Control Conference*, Kos, pp. 1–9.

VALAVANIS, K. e VACHTSEVANOS, G. (2015), *Handbook of Unmanned Aerial Vehicles*, 1ª ed., Londres, Springer.

Apêndice A – *STANAG_Library*

```
'''''ALEXANDRE RODRIGUES'''

from __future__ import print_function
import struct


class STANAG_header(object):
    '''''Cria o header da mensagem'''
    def __init__(self, msg_type, seq=0, mlen=0, source_id=0, dest_id=0, msg_prop=0):
        self.seq = seq
        self.mlen = mlen
        self.source_id = source_id
        self.dest_id = dest_id
        self.msg_type = msg_type
        self.msg_prop = msg_prop

    def pack(self):
        '''''faz o pack do header'''
        return struct.pack('<BBBBLB', self.seq, self.mlen, self.source_id, self.dest_id, self.msg_type, self.msg_prop)


class STANAG_message(object):
    '''''Cria a base de uma mensagem'''
    def __init__(self, msg_type, name):
        self._header = STANAG_header(msg_type)
        self._payload = None
        self._msgbuf = None
        self._fieldnames = []
        self._msg_type = msg_type  # representa o id da mensagem
        self._name = name

    def pack(self, payload):
        '''''faz o pack da mensagem'''
        self._payload = payload
        self._header = STANAG_header(self._msg_type, mlen=len(self._payload))

        self._msgbuf = self._header.pack() + self._payload
        return self._msgbuf

    # def __str__(self):
    #     ret = '%s {' % self._name
    #     for a in self._fieldnames:
    #         v = getattr(self, a)
    #         ret += '%s : %s, ' % (a, v)
    #     ret = ret[0:-2] + '}'
    #     return ret


# CLASS MENSAGENS STANAG - Cada mensagem e definida e formatada

class STANAG_cucs_authorisation_request(STANAG_message):
    name = "cucs_authorisation_request"
    msg_type = 1
    fieldnames = ["presence_vector", "time_stamp", "vms_id", "data_link_id", "vehicle_type", "vehicle_subtype", "requested_loi", "requested_access", "requested_flight_mode", "controlled_station_1_16", "component_number", "sub_componen
```

```
    t_number", "payload_type", "asset_mode", "wait_coord_message", "cucs_type", "c
    ucs_subtype", "presence_vector_support", "controlled_station_17_32"]
54.     format = ''
55.     native_format = bytearray('<', 'ascii')
56.
57.     def __init__(self, presence_vector, time_stamp, vsm_id, data_link_id, vehi
    cle_type, vehicle_subtype, requested_loi, requested_access, requested_flight_m
    ode, controlled_station_1_16, component_number, sub_component_number, payload_
    type, asset_mode, wait_coord_message, cucs_type, cucs_subtype, presence_vector
    _support, controlled_station_17_32):
58.         STANAG_message.__init__(self, STANAG_heartbeat.msg_type, STANAG_heartb
    eat.name)
59.         self._fieldnames = STANAG_cucs_authorisation_request.fieldnames
60.         self.presence_vector = presence_vector
61.         self.time_stamp = time_stamp
62.         self.vsm_id = vsm_id
63.         self.data_link_id = data_link_id
64.         self.vehicle_type = vehicle_type
65.         self.vehicle_subtype = vehicle_subtype
66.         self.requested_loi = requested_loi
67.         self.requested_access = requested_access
68.         self.requested_flight_mode = requested_flight_mode
69.         self.controlled_station_1_16 = controlled_station_1_16
70.         self.component_number = component_number
71.         self.sub_component_number = sub_component_number
72.         self.payload_type = payload_type
73.         self.asset_mode = asset_mode
74.         self.wait_coord_message = wait_coord_message
75.         self.cucs_type = cucs_type
76.         self.cucs_subtype = cucs_subtype
77.         self.presence_vector_support = presence_vector_support
78.         self.controlled_station_17_32 = controlled_station_17_32
79.
80.     def pack(self):
81.         return STANAG_message.pack(self, struct.pack('<', self.presence_vector
    , self.time_stamp, self.vsm_id, self.data_link_id, self.vehicle_type, self.veh
    icle_subtype, self.requested_loi, self.requested_access, self.requested_flight
    _mode, self.controlled_station_1_16, self.component_number, self.sub_component
    _number, self.payload_type, self.asset_mode, self.wait_coord_message, self.cuc
    s_type, self.cucs_subtype, self.presence_vector_support, self.controlled_stati
    on_17_32))
82.
83.
84. class STANAG_vsm_authorization_response(STANAG_message):
85.     name = "vsm_authorization_response"
86.     msg_type = 2
87.
88.
89. class STANAG_vehicle_operating_mode_command(STANAG_message):
90.     name = "vehicle_operating_mode_command"
91.     msg_type = 2001
92.     fieldnames = ["presence_vector", "time_stamp", "flight_control_mode", "act
    ivity_id"]
93.     format ="<IIBB"
94.     native_format = bytearray ('<IIBB', 'ascii')
95.     # When flight control mode is 21 = Launch
96.
97.     def __init__(self, presence_vector, time_stamp, flight_control_mode, activ
    ity_id):
98.         STANAG_message.__init__(self, STANAG_heartbeat.msg_type, STANAG_heartb
    eat.name)
99.         self._fieldnames = STANAG_vehicle_operating_mode_command.fieldnames
100.            self.presence_vector = presence_vector
```

```
101.                self.time_stamp = time_stamp
102.                self.flight_control_mode = flight_control_mode
103.                self.activity_id = activity_id
104.
105.            def pack(self):
106.                return STANAG_message.pack(self, struct.pack('<IIBB', self.pres
    ence_vector, self.time_stamp, self.flight_control_mode, self.activity_id))
107.
108.
109.        class STANAG_mission_transfer_cmd(STANAG_message):
110.            name = "mission transfer command"
111.            msg_type = 13000
112.            fieldnames = ["presence_vector", "time_stamp", "mission_id", "missi
    on_plan_mode", "wp_number", "route_id", "activity_id"]
113.            format = "<BI20sBH33sI"
114.            native_format = bytearray('<BIcBHcI', 'ascii')
115.            lengths = [1, 1, 20, 1, 1, 33, 1]
116.            array_lengths = [0, 0, 20, 0, 0, 33, 0]
117.
118.            def __init__(self, presence_vector, time_stamp, mission_id, mission
    _plan_mode, wp_number, route_id, activity_id):
119.                STANAG_message.__init__(self, STANAG_mission_transfer_cmd.msg_t
    ype, STANAG_mission_transfer_cmd.name)
120.                self._fieldnames = STANAG_mission_transfer_cmd.fieldnames
121.                self.presence_vector = presence_vector
122.                self.time_stamp = time_stamp
123.                self.mission_id = mission_id.encode('ascii')
124.                self.mission_plan_mode = mission_plan_mode
125.                self.wp_number = wp_number
126.                self.route_id = route_id
127.                self.activity_id = activity_id
128.
129.            def pack(self):
130.                return STANAG_message.pack(self, struct.pack("<BI20sBH33sI", se
    lf.presence_vector, self.time_stamp, self.mission_id, self.mission_plan_mode,
    self.wp_number, self.route_id, self.activity_id))
131.
132.
133.        class STANAG_av_position_wp(STANAG_message):
134.            name = "av position waypoint"
135.            msg_type = 13002
136.            fieldnames = ["presence_vector", "time_stamp", "wp_number", "wp_to_
    lat_or_rel_y", "wp_to_lat_or_rel_x", "location_type",
137.                          "wp_to_alt", "wp_alt_type", "alt_change_behaviour", "
    wp_to_speed", "wp_speed_type", "next_wp", "turn_type", "optional_msg_wp", "wp_
    type", "limit_type", "loop_limit", "arrival_time", "activity_id"]
138.            format = "<IIHffBiBBHBHBBBBHII"
139.            native_format = bytearray('<IIHffBiBBHBHBBBBHII', 'ascii')
140.            lengths = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
141.
142.            def __init__(self, presence_vector, time_stamp, wp_number, wp_to_la
    t_or_rel_y, wp_to_lat_or_rel_x, location_type, wp_to_alt, wp_alt_type, alt_cha
    nge_behaviour, wp_to_speed, wp_speed_type, next_wp, turn_type, optional_msg_wp
    , wp_type, limit_type, loop_limit, arrival_time, activity_id):
143.                STANAG_message.__init__(self, STANAG_av_position_wp.msg_type, S
    TANAG_av_position_wp.name)
144.                self._fieldnames = STANAG_av_position_wp.fieldnames
145.                self.presence_vector = presence_vector
146.                self.time_stamp = time_stamp
147.                self.wp_number = wp_number
148.                self.wp_to_lat_or_rel_y = wp_to_lat_or_rel_y
149.                self.wp_to_lat_or_rel_x = wp_to_lat_or_rel_x
```

```
150.                self.location_type = location_type
151.                self.wp_to_alt = wp_to_alt
152.                self.wp_alt_type = wp_alt_type
153.                self.alt_change_behaviour = alt_change_behaviour
154.                self.wp_to_speed = wp_to_speed
155.                self.wp_speed_type = wp_speed_type
156.                self.next_wp = next_wp
157.                self.turn_type = turn_type
158.                self.optional_msg_wp = optional_msg_wp
159.                self.wp_type = wp_type
160.                self.limit_type = limit_type
161.                self.loop_limit = loop_limit
162.                self.arrival_time = arrival_time
163.                self.activity_id = activity_id
164.
165.            def pack(self):
166.                return STANAG_message.pack(self, struct.pack('<IIHffBiBBHBHBBBB
    HII', self.presence_vector, self.time_stamp, self.wp_number, self.wp_to_lat_or
    _rel_y, self.wp_to_lat_or_rel_x, self.location_type, self.wp_to_alt, self.wp_a
    lt_type, self.alt_change_behaviour, self.wp_to_speed, self.wp_speed_type, self
    .next_wp, self.turn_type, self.optional_msg_wp, self.wp_type, self.limit_type,
     self.loop_limit, self.arrival_time, self.activity_id))
167.
168.
169.
170.        class STANAG_heartbeat(STANAG_message):
171.            name = "heartbeat"
172.            msg_type = 16002
173.            fieldnames = ["presence_vector", "time_stamp"]
174.            format = '<BBBBBB'
175.            native_format = bytearray('<BBBBBB', 'ascii')
176.
177.            def __init__(self, presence_vector, time_stamp):
178.                STANAG_message.__init__(self, STANAG_heartbeat.msg_type, STANAG
    _heartbeat.name)
179.                self._fieldnames = STANAG_heartbeat.fieldnames
180.                self.presence_vector = presence_vector
181.                self.time_stamp = time_stamp
182.
183.            def pack(self):
184.                return STANAG_message.pack(self, struct.pack('<BBBBBB', self.pr
    esence_vector, self.time_stamp))
185.
186.
187.        stanag_messages = {
188.                1: STANAG_cucs_authorisation_request,
189.                2: STANAG_vsm_authorization_response,
190.                2001: STANAG_vehicle_operating_mode_command,
191.                13000: STANAG_mission_transfer_cmd,
192.                13002: STANAG_av_position_wp,
193.                16002: STANAG_heartbeat,
194.            }
195.
196.
197.        class STANAGString(str):
198.            '''''NUL terminated string'''
199.
200.            def __init__(self, s):
201.                str.__init__(self)
202.
203.            def __str__(self):
204.                i = self.find(chr(0))
205.                if i == -1:
```

```
206.                     return self[:]
207.                 return self[0:i]
208.
209.
210.         ''''' CLASS STANAG - Para enviar mensagens e preencher o buffer '''
211.
212.
213.         class STANAG(object):
214.             def __init__(self):
215.                 self.buf = bytearray()
216.                 self.single_buf_size = []  # size of each buf
217.                 self.total_packets_sent = 0
218.                 self.total_bytes_sent = 0
219.                 self._type = []
220.
221.             def send(self, stanag_msg):
222.                 temp_len = len(self.buf)
223.                 self.buf += stanag_msg.pack()
224.                 self.total_packets_sent += 1
225.                 self.total_bytes_sent += len(self.buf)
226.                 self.single_buf_size.append(len(self.buf) - temp_len)
227.
228.             def get_msg_type(self, msgbuf):
229.                 headerlen = 9
230.                 seq, mlen, source_id, dest_id, msg_type, msg_prop = struct.unpa
    ck('<BBBBLB', msgbuf[:headerlen])
231.                 return msg_type
232.
233.             def get_total_packets_sent(self):
234.                 return self.total_packets_sent
235.
236.             def decode(self, msgbuf):
237.                 headerlen = 9
238.                 seq, mlen, source_id, dest_id, msg_type, msg_prop = struct.unpa
    ck('<BBBBLB', msgbuf[:headerlen])
239.                 type = msg_type
240.                 if mlen != len(msgbuf) - headerlen:
241.                     raise Exception('invalid MAVLink message length')
242.
243.                 if not type in stanag_messages:
244.                     raise Exception('unknown STANAG message type %s' % type)
245.
246.                 # Decode
247.                 msg = stanag_messages[type]
248.                 fmt = msg.format
249.                 csize = struct.calcsize(fmt)
250.                 mbuf = msgbuf[headerlen:]
251.                 #print(csize)
252.                 #print(len(mbuf))
253.                 if len(mbuf) != csize:
254.                     raise Exception('STANAG messages length is incorrect')
255.                 t = struct.unpack(fmt, mbuf)
256.                 tlist = list(t)
257.                 #print(t)
258.
259.                 for i in range(0, len(tlist)):
260.                     if isinstance(tlist[i], str):
261.                         tlist[i] = str(STANAGString(tlist[i]))
262.                 t = tuple(tlist)
263.                 #print(t)
264.
265.                 try:
266.                     m = msg(*t)
```

```
267.                except Exception as emsg:
268.                    raise Exception('Unable to instantiate STANAG message of ty
    pe %s : %s' % (type, emsg))
269.                m._msgbuf = msgbuf
270.                m._payload = msgbuf[9:]
271.                m._header = STANAG_header(seq, mlen, source_id, dest_id, msg_ty
    pe, msg_prop)
272.                return m
273.
274.            def buf_len(self):
275.                return len(self.buf)
276.
277.            def get_buf(self):
278.                #index = self.single_buf_size[-1]
279.                return self.buf[-self.single_buf_size[-1]:]
280.                # return self.buf
281.
282.            def mission_transfer_cmd_encode(self, presence_vector, time_stamp,
    mission_id, mission_plan_mode, wp_number, route_id, activity_id):
283.                return STANAG_mission_transfer_cmd(presence_vector, time_stamp,
     mission_id, mission_plan_mode, wp_number, route_id, activity_id)
284.
285.            def mission_transfer_cmd_send(self, presence_vector, time_stamp, mi
    ssion_id, mission_plan_mode, wp_number, route_id, activity_id):
286.                return self.send(self.mission_transfer_cmd_encode(presence_vect
    or, time_stamp, mission_id, mission_plan_mode, wp_number, route_id, activity_i
    d))
287.
288.            def av_position_wp_encode(self,presence_vector, time_stamp, wp_numb
    er, wp_to_lat_or_rel_y, wp_to_lat_or_rel_x, location_type, wp_to_alt, wp_alt_t
    ype, alt_change_behaviour, wp_to_speed, wp_speed_type, next_wp, turn_type, opt
    ional_msg_wp, wp_type, limit_type, loop_limit, arrival_time, activity_id):
289.                return STANAG_av_position_wp(presence_vector, time_stamp, wp_nu
    mber, wp_to_lat_or_rel_y, wp_to_lat_or_rel_x, location_type, wp_to_alt, wp_alt
    _type, alt_change_behaviour, wp_to_speed, wp_speed_type, next_wp, turn_type, o
    ptional_msg_wp, wp_type, limit_type, loop_limit, arrival_time, activity_id)
290.
291.            def av_position_wp_send(self,presence_vector, time_stamp, wp_number
    , wp_to_lat_or_rel_y, wp_to_lat_or_rel_x, location_type, wp_to_alt, wp_alt_typ
    e, alt_change_behaviour, wp_to_speed, wp_speed_type, next_wp, turn_type, optio
    nal_msg_wp, wp_type, limit_type, loop_limit, arrival_time, activity_id):
292.                return self.send(self.av_position_wp_encode(presence_vector, ti
    me_stamp, wp_number, wp_to_lat_or_rel_y, wp_to_lat_or_rel_x, location_type, wp
    _to_alt, wp_alt_type, alt_change_behaviour, wp_to_speed, wp_speed_type, next_w
    p, turn_type, optional_msg_wp, wp_type, limit_type, loop_limit, arrival_time,
    activity_id))
293.
294.            def vehicle_operating_mode_command_enconde(self, presence_vector, t
    ime_stamp, flight_control_mode, activity_id):
295.                return STANAG_vehicle_operating_mode_command(presence_vector, t
    ime_stamp, flight_control_mode, activity_id)
296.
297.            def vehicle_operating_mode_command_send(self, presence_vector, time
    _stamp, flight_control_mode, activity_id):
298.                return self.send(self.vehicle_operating_mode_command_enconde(pr
    esence_vector, time_stamp, flight_control_mode, activity_id))
299.
300.        s_msg = STANAG()
301.
302.        #TESTES
303.
304.        #mensagem_base = STANAG_message(16002, 'heartbeat')
305.        #mensagem_hb = STANAG_heartbeat(10, 20)
```

```
306.       #mensagem_wp = STANAG_av_position_wp(1, 2, 10, 20, 30, 40, 50, 1, 2, 5,
     10, 23, 12, 2, .0, 1, 2, 3, 1420)
307.
308.       # MSG STANAG
309.       #msg_mission = STANAG_mission_transfer_cmd(2, .3, 3, 3, 4, 2, 2)
310.
311.
312.       # PACK da Mensagem
313.       #msg_mission_packed = msg_mission.pack()
314.
315.
316.       # Fazer o envio da MSG (Faz o pack e poe no buffer)
317.       #stanag.mission_transfer_cmd_send(2, 3, 'julieaaaaa', 3, 40000, 'lisboa
    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\x00', 2)
318.       #s_msg.av_position_wp_send(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1
    4, 15, 16, 17, 18, 19)
319.       #buf = s_msg.get_buf()
320.       #decoded_message = s_msg.decode(buf)
321.
322.       #print ('decoded message:', decoded_message)
323.       #print(buf)
324.
325.
326.
327.       #print(msg_mission)
```

Apêndice B – *SM_Bridge*

```
# -*- coding: utf-8 -*-
# Alexandre Rodrigues - STANAG to MAVLink

import STANAG_library as Stanag
from dronekit import connect, VehicleMode, Command
from pymavlink import mavutil
import time
from PyQt4 import QtCore, QtGui
import sys

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
        return s

try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig, _encoding
)
except AttributeError:
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig)
# Connect to the Vehicle and reset commands


vehicle = connect('127.0.0.1:14551', wait_ready=True)
print("Connected to 127.0.0.1:14551")
cmds = vehicle.commands
cmds.clear()
cmds.upload()

# BRIDGE


class Reception(object):
    '''''Receives every message and identifies them'''
    def __init__(self):
        self.buf = bytearray()
        self.type = None
        self.length = 0
        self.total_packets = 0
        self.msg_type = []

    def identification(self, msg_buf):
        self.buf = msg_buf
        self.msg_type.append(Stanag.s_msg.get_msg_type(self.buf))
        #self.total_packets = Stanag.s_msg.get_total_packets_sent()
        #while self.total_packets != 0:
        if self.msg_type[-1] == 13002:
            wp.wp_trans(msg_buf)
            # print(self.total_packets)
            # self.total_packets -= 1
            self.buf = self.buf[53:]
            #print(len(self.buf))
            # self.msg_type.pop(0)

    def close_vehicle(self):
```

```
59.         vehicle.close()
60.         print('Program finished and vehicle closed')
61.         return
62.
63.
64. class SM_WP(object):
65.     '''''WP Transformation - STANAG to MAVLink'''
66.     def __init__(self):
67.         self.wp_number = 0
68.         self.len_buf = 0
69.         self.wp_total = 0
70.         self.cmd = []
71.
72.     def wp_trans(self, wp_buf):
73.         decoded_wp = Stanag.s_msg.decode(wp_buf)
74.         lat = decoded_wp.wp_to_lat_or_rel_y
75.         lon = decoded_wp.wp_to_lat_or_rel_x
76.         alt = decoded_wp.wp_to_alt
77.         #print(lat, lon, alt)
78.         if not self.cmd:
79.             takeoff_cmd = Command(0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RE
    LATIVE_ALT, mavutil.mavlink.MAV_CMD_NAV_TAKEOFF, 0, 0, 0, 0, 0, 0, 0, 10)
80.             self.cmd.append(takeoff_cmd)
81.             print('Takeoff Added')
82.
83.         wp_cmd = Command(0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_AL
    T, mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0, lat, lon, alt)
84.         self.cmd.append(wp_cmd)
85.
86.         #cmd2 = Command(0, 0, 0, mavutil.mavlink.MAV_FRAME_GLOBAL_RELATIVE_ALT
    , mavutil.mavlink.MAV_CMD_NAV_WAYPOINT, 0, 0, 0, 0, 0, 0, lat, lon, alt)
87.         #cmds.add(cmd1)
88.         #cmds.add(cmd2)
89.         #cmds.upload()  # Send commands
90.         #time.sleep(1)
91.         return
92.
93.     def send_all(self):
94.         start_time2 = time.time()
95.         for i in range(0, len(self.cmd)):
96.             print (i)
97.             cmds.add(self.cmd[i])
98.
99.         cmds.upload()
100.                finish_time2 = time.time()
101.                running_time2 = finish_time2 - start_time2
102.                print('All WPs Sent, time:', running_time2)
103.                time.sleep(2)
104.
105.            def wp_clear(self):
106.                for i in range(0, len(self.cmd)):
107.                    del(self.cmd[0])
108.                cmds.clear()
109.                cmds.upload()
110.
111.
112.        # Graphical User Interface
113.
114.
115.        class Ui_Form(QtGui.QWidget):
116.            def __init__(self):
117.                QtGui.QWidget.__init__(self)
118.                self.setupUi(self)
```

```
119.
120.             def setupUi(self, Form):
121.                 Form.setObjectName(_fromUtf8("Form"))
122.                 Form.resize(517, 517)
123.                 self.horizontalLayoutWidget = QtGui.QWidget(Form)
124.                 self.horizontalLayoutWidget.setGeometry(QtCore.QRect(20, 60, 24
    1, 51))
125.                 self.horizontalLayoutWidget.setObjectName(_fromUtf8("horizontal
    LayoutWidget"))
126.                 self.horizontalLayout = QtGui.QHBoxLayout(self.horizontalLayout
    Widget)
127.                 self.horizontalLayout.setObjectName(_fromUtf8("horizontalLayout
    "))
128.                 self.label = QtGui.QLabel(self.horizontalLayoutWidget)
129.                 self.label.setObjectName(_fromUtf8("label"))
130.                 self.horizontalLayout.addWidget(self.label)
131.                 self.lineEdit = QtGui.QLineEdit(self.horizontalLayoutWidget)
132.                 self.lineEdit.setObjectName(_fromUtf8("lineEdit"))
133.                 self.horizontalLayout.addWidget(self.lineEdit)
134.                 self.horizontalLayoutWidget_2 = QtGui.QWidget(Form)
135.                 self.horizontalLayoutWidget_2.setGeometry(QtCore.QRect(20, 110,
     241, 51))
136.                 self.horizontalLayoutWidget_2.setObjectName(_fromUtf8("horizont
    alLayoutWidget_2"))
137.                 self.horizontalLayout_2 = QtGui.QHBoxLayout(self.horizontalLayo
    utWidget_2)
138.                 self.horizontalLayout_2.setObjectName(_fromUtf8("horizontalLayo
    ut_2"))
139.                 self.label_2 = QtGui.QLabel(self.horizontalLayoutWidget_2)
140.                 self.label_2.setObjectName(_fromUtf8("label_2"))
141.                 self.horizontalLayout_2.addWidget(self.label_2)
142.                 self.lineEdit_2 = QtGui.QLineEdit(self.horizontalLayoutWidget_2
    )
143.                 self.lineEdit_2.setObjectName(_fromUtf8("lineEdit_2"))
144.                 self.horizontalLayout_2.addWidget(self.lineEdit_2)
145.                 self.horizontalLayoutWidget_3 = QtGui.QWidget(Form)
146.                 self.horizontalLayoutWidget_3.setGeometry(QtCore.QRect(20, 160,
     241, 51))
147.                 self.horizontalLayoutWidget_3.setObjectName(_fromUtf8("horizont
    alLayoutWidget_3"))
148.                 self.horizontalLayout_3 = QtGui.QHBoxLayout(self.horizontalLayo
    utWidget_3)
149.                 self.horizontalLayout_3.setObjectName(_fromUtf8("horizontalLayo
    ut_3"))
150.                 self.label_3 = QtGui.QLabel(self.horizontalLayoutWidget_3)
151.                 self.label_3.setObjectName(_fromUtf8("label_3"))
152.                 self.horizontalLayout_3.addWidget(self.label_3)
153.                 self.lineEdit_3 = QtGui.QLineEdit(self.horizontalLayoutWidget_3
    )
154.                 self.lineEdit_3.setObjectName(_fromUtf8("lineEdit_3"))
155.                 self.horizontalLayout_3.addWidget(self.lineEdit_3)
156.                 self.textBrowser = QtGui.QTextBrowser(Form)
157.                 self.textBrowser.setGeometry(QtCore.QRect(130, 10, 221, 31))
158.                 self.textBrowser.setObjectName(_fromUtf8("textBrowser"))
159.                 self.line = QtGui.QFrame(Form)
160.                 self.line.setGeometry(QtCore.QRect(20, 290, 481, 20))
161.                 self.line.setFrameShape(QtGui.QFrame.HLine)
162.                 self.line.setFrameShadow(QtGui.QFrame.Sunken)
163.                 self.line.setObjectName(_fromUtf8("line"))
164.                 self.pushButton_4 = QtGui.QPushButton(Form)
165.                 self.pushButton_4.setGeometry(QtCore.QRect(70, 320, 151, 31))
166.                 self.pushButton_4.setObjectName(_fromUtf8("pushButton_4"))
167.                 self.pushButton_5 = QtGui.QPushButton(Form)
```

```
168.                 self.pushButton_5.setGeometry(QtCore.QRect(280, 320, 151, 31))
169.                 self.pushButton_5.setObjectName(_fromUtf8("pushButton_5"))
170.                 self.pushButton = QtGui.QPushButton(Form)
171.                 self.pushButton.setGeometry(QtCore.QRect(280, 120, 151, 31))
172.                 self.pushButton.setObjectName(_fromUtf8("pushButton"))
173.                 self.pushButton_2 = QtGui.QPushButton(Form)
174.                 self.pushButton_2.setGeometry(QtCore.QRect(70, 240, 151, 31))
175.                 self.pushButton_2.setObjectName(_fromUtf8("pushButton_2"))
176.                 self.pushButton_3 = QtGui.QPushButton(Form)
177.                 self.pushButton_3.setGeometry(QtCore.QRect(280, 240, 151, 31))
178.                 self.pushButton_3.setObjectName(_fromUtf8("pushButton_3"))
179.                 self.listWidget = QtGui.QListWidget(Form)
180.                 self.listWidget.setGeometry(QtCore.QRect(70, 370, 361, 121))
181.                 self.listWidget.setObjectName(_fromUtf8("listWidget"))
182. 
183.                 self.retranslateUi(Form)
184.                 QtCore.QMetaObject.connectSlotsByName(Form)
185. 
186.             def retranslateUi(self, Form):
187.                 Form.setWindowTitle(_translate("Form", "SM", None))
188.                 self.label.setText(_translate("Form", "Latitude (ยน):    ", Non
    e))
189.                 self.label_2.setText(_translate("Form", "Longitude (ยน): ", Non
    e))
190.                 self.label_3.setText(_translate("Form", "Altitude (m):   ", Non
    e))
191.                 self.textBrowser.setHtml(_translate("Form", "<!DOCTYPE HTML PUB
    LIC \"-//W3C//DTD HTML 4.0//EN\" \"http://www.w3.org/TR/REC-
    html40/strict.dtd\">\n"
192.         "<html><head><meta name=\"qrichtext\" content=\"1\" /><style type=\"tex
    t/css\">\n"
193.         "p, li { white-space: pre-wrap; }\n"
194.         "</style></head><body style=\" font-family:\'Ubuntu\'; font-
    size:11pt; font-weight:400; font-style:normal;\">\n"
195.         "<p align=\"center\" style=\" margin-top:0px; margin-
    bottom:0px; margin-left:0px; margin-right:0px; -qt-block-indent:0; text-
    indent:0px;\"><span style=\" font-size:12pt; font-
    weight:600;\">STANAG 4586 to MAVLink</span></p></body></html>", None))
196.                 self.pushButton_4.setText(_translate("Form", "Start Mission", N
    one))
197.                 self.pushButton_5.setText(_translate("Form", "Load WPs from UAV
    ", None))
198.                 self.pushButton.setText(_translate("Form", "Introduce WP", None
    ))
199.                 self.pushButton_2.setText(_translate("Form", "Send All to UAV",
     None))
200.                 self.pushButton_3.setText(_translate("Form", "Clear All", None)
    )
201.                 self.pushButton.clicked.connect(self.introduce)
202.                 self.pushButton_2.clicked.connect(self.send_all)
203.                 self.pushButton_3.clicked.connect(self.clear_all)
204.                 self.pushButton_5.clicked.connect(self.load)
205.                 self.pushButton_4.clicked.connect(self.startmission)
206. 
207.             def closeEvent(self, event):
208.                 buf_list.close_vehicle()
209.                 event.accept()
210. 
211.             def introduce(self):
212.                 lat = float(self.lineEdit.text())
213.                 lon = float(self.lineEdit_2.text())
```

```
214.                alt = float(self.lineEdit_3.text())
215.                # LIB
216.                Stanag.s_msg.av_position_wp_send(0, 0, 1, lat, lon, 0, alt, 3,
    4, 5, 1, 0, 0, 0, 1, 1, 0, 0, 0)
217.                start_time = time.time()
218.                wp_buf = Stanag.s_msg.get_buf()
219.                # Bridge
220.                buf_list.identification(wp_buf)
221.                finish_time = time.time()
222.                running_time = finish_time - start_time
223.                wp_text = 'WP Added, time:' + str(running_time)
224.                self.listWidget.addItem(wp_text)
225.
226.            def send_all(self):
227.                wp.send_all()
228.                self.listWidget.addItem('Mission sent to the UAV')
229.
230.            def clear_all(self):
231.                self.listWidget.clear()
232.                wp.wp_clear()
233.                self.listWidget.addItem('Waypoints deleted')
234.
235.            def startmission(self):
236.                # LIB
237.                Stanag.s_msg.vehicle_operating_mode_command_send(0, 0, 21, 0)
238.                wp_buf = Stanag.s_msg.get_buf()
239.                # Bridge
240.                buf_list.identification(wp_buf)
241.
242.                # LAUNCH THE VEHICLE
243.                vehicle.mode = VehicleMode("GUIDED")
244.                vehicle.armed = True
245.                time.sleep(1)
246.                aTargetAltitude = 10
247.                vehicle.simple_takeoff(aTargetAltitude)
248.                while True:
249.                    print " Altitude: ", vehicle.location.global_relative_frame
    .alt
250.                    if vehicle.location.global_relative_frame.alt >= aTargetAlt
    itude * 0.95:  # Trigger just below target alt.
251.                        print "Reached target altitude"
252.                        break
253.                    time.sleep(1)
254.
255.                # Reset mission set to first (0) waypoint
256.                vehicle.commands.next = 0
257.
258.                # Set mode to AUTO to start mission
259.                vehicle.mode = VehicleMode("AUTO")
260.
261.            def load(self):
262.                # Download the vehicle waypoints
263.                self.listWidget.clear()
264.                missionlist = []
265.                cmds = vehicle.commands
266.                cmds.download()
267.                cmds.wait_ready()
268.                for cmd in cmds:
269.                    missionlist.append(cmd)
270.                    # Add file-format information
271.                output = 'QGC WPL 110\n'
272.                # Add home location as 0th waypoint
273.                home = vehicle.home_location
```

```
                output += "%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\n" %
(
                0, 1, 0, 16, 0, 0, 0, 0, home.lat, home.lon, home.alt, 1)
                # Add commands
                for cmd in missionlist:
                    commandline = "%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t
%s\n" % (
                    cmd.seq, cmd.current, cmd.frame, cmd.command, cmd.param1, c
md.param2, cmd.param3, cmd.param4, cmd.x, cmd.y,
                    cmd.z, cmd.autocontinue)
                    Stanag.s_msg.av_position_wp_send(0, 0, 1, cmd.x, cmd.y, 0,
cmd.z, 3, 4, 5, 1, 0, 0, 0, 1, 1, 0, 0, 0)
                    wp_text = 'Lat:' + str(cmd.x) + ' Lon' + str(cmd.y) + ' Alt
' + str(cmd.z)
                    self.listWidget.addItem(str(wp_text))
                    output += commandline
                with open('mission_load', 'w') as file_:
                    self.listWidget.addItem('Mission loaded and written in miss
ion_load')
                    file_.write(output)


        buf_list = Reception()
        wp = SM_WP()


        if __name__ == '__main__':
            app = QtGui.QApplication(sys.argv)
            ex = Ui_Form()
            ex.show()
            sys.exit(app.exec_())
```