INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Implementação do LoRaWAN 1.0.4 e 1.1 em Micropython

Miguel Mário Inácio Guerreiro

Licenciado

Dissertação para obtenção do Grau de Mestre
em Engenharia Eletrónica e Telecomunicações

Orientador : Professor Doutor Nuno Cruz

Júri:

Presidente: Professor Doutor Rui Duarte

Vogal: Professor Doutor João Casaleiro

Setembro, 2023

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Implementação do LoRaWAN 1.0.4 e 1.1 em Micropython

Miguel Mário Inácio Guerreiro

Licenciado

Dissertação para obtenção do Grau de Mestre
em Engenharia Eletrónica e Telecomunicações

Orientador : Professor Doutor Nuno Cruz

Júri:

Presidente: Professor Doutor Rui Duarte

Vogal: Professor Doutor João Casaleiro

Setembro, 2023

À minha avó, Maria Ângela de Almeida

Agradecimentos

Primeiramente, gostaria de agradecer ao meu Orientador Professor Doutor Nuno Cruz, que brevemente introduziu-me a este projeto e acompanhou o meu percurso orientando-me com os passos certos para completar este desafio. Muito obrigado pela sua confiança e disponibilidade durante esta fase da minha vida.

Agradeço a todos os colegas que se cruzaram comigo durante este período. Obrigado por toda a aprendizagem, apoio e motivação que me foram passando ao longos dos anos.

Quero agradecer aos meus pais e irmã por todo o apoio e amor que me foram dando ao longo da minha vida e que de tudo fizeram para que eu conseguisse chegar aqui e concluir esta etapa. Agradeço a todos os restantes elementos da minha família que sempre me motivaram e apoiaram.

Por fim, agradeço também a todos os meus amigos que nos momentos tensos desta jornada ajudaram-me a distrair e me apoiaram para finalizar esta aventura.

Resumo

A tecnologia tem vindo a evoluir e a tecnologia LoRa e o protocolo LoRaWAN não são exceção. Com esta evolução foram publicadas novas versões do protocolo LoRaWAN, porém nem todos os dispositivos que utilizam LoRaWAN evoluem ao mesmo ritmo quer seja por falta de necessidade ou por investimento noutros pontos relevantes do microcontrolador ou do *kit* de desenvolvimento em questão.

Com a publicação das versões 1.1 e 1.0.4 do protocolo foram introduzidas novas funcionalidades e uma nova arquitetura. Ao atualizar o Stack LoRaWAN obtém-se uma ligação mais segura e mais funcionalidades introduzidas pelos novos comandos MAC.

Tendo isto em conta, pretende-se atualizar o *firmware* dos dispositivos que utilizem o ESP32, no caso utilizar-se-ão dispositivos da Pycom. Os dispositivos escolhidos foram desenvolvidos pela Pycom e foram os selecionados para este projeto por utilizarem um microcontrolador ESP32, ter o *firmware* aberto e já utilizarem a modulação LoRa e que implementam o protocolo LoRaWAN.

Inicialmente, utilizou-se uma versão beta do *firmware* da Pycom que implementa a versão 1.0.3 do protocolo LoRaWAN, porém as funcionalidades não estavam implementadas para o utilizador final. Para conseguir implementar estas funcionalidades foi necessário uma análise dos códigos das diferentes versões do *firmware* para perceber a interligação da *Stack* com o resto do código. Este processo consistiu observação detalhada do *firmware* utilizado e entender o contributo de cada "setor".

Ao passar à atualização da *Stack* no *firmware* da Pycom, utilizou-se o conhecimento adquirido na análise do *firmware* de modo a efetuar as alterações necessárias no âmbito do projeto. Nesta fase apareceram as maiores dificuldades devido às diferenças da nova versão da Stack e como esta seria implementada no *firmware*. Após a resolução de todos os problemas, verificou-se o funcionamento das funções básicas referentes ao LoRa,

sendo estas as funções de *join* e de envio de pacotes. Em seguida, implementaram-se as funções que permitiam a utilização dos comandos MAC e testaram-se as funções criadas.

Por fim, utilizou-se um documento disponibilizado pela LoRa Alliance como guia para testes de conformidade dos comandos MAC. Este documentos especifica os testes que devem ser efetuados para averiguar o funcionamento das diferentes funcionalidades da versão 1.0.4 do protocolo LoRaWAN para certificar um dispositivo.

Palavras-chave: LoRa, LoRaWAN, ESP32. *firmware*, Pycom, MicroPython

Abstract

Technology has been evolving, and LoRa technology and the LoRaWAN protocol are no exception. With this evolution, new versions of the protocol have been published, but not all devices that use LoRaWAN evolve at the same pace, either due to a lack of necessity or investment in other relevant aspects of the microcontroller or development kit in question.

With the release of version 1.0.4 of the protocol, new features and a new architecture were introduced. Updating the LoRaWAN Stack provides a more secure connection and additional functionality due to the new MAC commands.

In the release of versions 1.1 and 1.0.4 of the LoRaWAN protocol, new functionalities and a new architecture were introduced. Updating the LoRaWAN Stack ensures a more secure connection and additional features facilitated by the new MAC commands.

Given this progression, the goal is to update the firmware of devices utilizing the ESP32, in this case, Pycom devices will be used. These Pycom devices were chosen for the project because they implent the ESP32 microcontroller, have an open-source firmware, and already feature devices using LoRa that implement the LoRaWAN protocol.

Initially, a beta version of Pycom's firmware, implementing LoRaWAN protocol version 1.0.3, was used. The funcionalities implemented by this new version wasn't implementend in this firmware for the end-users. To implement these functionalities successfully, a thorough analysis of the firmware codes from different versions was conducted to understand the connection of the Stack with the rest of the code.

Moving on to the Stack update in the Pycom firmware, the knowledge gained from the firmware analysis was applied to make the necessary changes within the project

scope. This phase presented the greatest challenges due to the differences in the new Stack version and how it would be implemented in the firmware. After resolving all the problemas, the basic LoRa functions, such as join and packet sending functions, were confirmed to be operational. Subsequently, functions enabling the use of MAC commands were implemented and tested.

Finally, a document provided by the LoRa Alliance was used as a guide for MAC command conformity tests. This document specifies the tests that should be performed to verify the functionality of different features of version 1.0.4 of the LoRaWAN protocol to certify a device.

Keywords: LoRa, LoRaWAN, ESP32. *firmware*, Pycom, MicroPython

Índice

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Lista de Abreviaturas e Siglas

ABP	*Activation By Personalisation*.
ADR	*Adaptive Data Rate*.
AppKey	*Application Key*.
AppSKey	*Application Session Key*.
BW	*Bandwidth*.
CR	*Coding Rate*.
CSS	*Chirp Spread Spectrum*.
DR	*Data Rate*.
ETSI	*European Telecommunications Standards Institute*.
FEC	*Forward Error Correction*.
fNS	*Forwarding Network Server*.
FNwkSIntKey	Forwarding Network Session Integrity Key.
hNS	*Home Network Server*.
IBM	*International Business Machines Corporation*.
IoT	*Internet Of Things*.
JS	*Join Server*.

LoRa	*Long Range*.
LoRaWAN	*Long Range Wide Area Network*.
LPWAN	*Low Power Wide Area Network*.
MAC	*Medium Access Control*.
MIC	*Message Integrity Check*.
NS	*Network Server*.
NVRAM	*Non-Volatile Random Access Memory*.
NwkKey	*Network Key*.
NwkSEncKey	*Network Session Encryption Key*.
NwkSKey	*Network Session Key*.
OTAA	*Over-The-Air-Activation*.
SF	*Spreading Factor*.
SNR	*Signal-to-Noise Ratio*.
sNS	*Serving Network Server*.
SNwkSIntKey	*Serving Network Session Integrity Key*.
ToA	*Time On Air*.
TTN	*The Things Network*.

1

Introdução

Uma das áreas tecnológicas que cresceu bastante nos últimos anos foi a área do *Internet Of Things* (IoT) ou, em português, Internet das Coisas, sendo esta uma tecnologia mais próxima da sociedade e em que, por vezes, uma pessoa comum consegue facilmente configurar os dispositivos e aproveitar o que têm para oferecer.

Tendo em conta as diferentes necessidades foram desenvolvidas diversas tecnologias de comunicação e os respetivos protocolos de comunicação. Em algumas situações é necessário mais segurança, mais alcance, mais ritmo de transmissão e para cada opção há tecnologias que se podem usar que melhor se adequam à necessidade, isto acontece porque não existe uma tecnologia perfeita então tem-se em conta as características e estabelece-se um compromisso.

Neste projeto será atualizado o *firmware* do microcontrolador ESP32, um dispositivo programado em *MicroPython*, para a utilizar a versão da *Stack Long Range Wide Area Network* (LoRaWAN) 1.0.4 de modo a que seja possível usar as funcionalidades que foram implementadas na atualização realizada pela **LoRa Alliance**.

No inicio deste documento apresenta-se o estado da arte do projeto, ou seja, apresenta-se o que existe e já foi feito com o microcontrolador para o qual se pretende atualizar o *firmware*, e ainda se apresentam bibliotecas que já implementem a versão 1.0.4 do protocolo LoRaWAN mas que não podem ser aplicadas para o ESP32.

Em seguida, é descrito o trabalho realizado. Todo o processo é descrito desde os primeiros passos de ambientalização com o *firmware* até à explicação e resultados dos testes de conformidade da *Stack* LoRaWAN para a versão pretendida.

1.1 Motivação

Existem cerca 6,5 milhões de *gateways*, cerca 280 milhões de dispositivos LoRa [1] sendo que uma boa parte destes dispositivos são baseados em Python. Os dispositivos baseados em Python só têm acesso à versão 1.0.2 do protocolo LoRaWAN, o que deixa de fora as novas funcionalidades provenientes das versões seguintes.

Tendo em conta as bases da linguagem de programação C juntou-se a necessidade às competências obtidas durante o percurso académico de modo a possibilitar todos os utilizadores da comunidade a usarem as novas funcionalidades para os seus projetos. O código desenvolvido será disponibilizado livremente para uso de qualquer um.

Esta opção veio fazer mais sentido quando se soube do processo de insolvência da empresa Pycom e mais tarde a aquisição da mesma pelo grupo *Season Group*. Ao disponibilizar o código dar-se-ia mais liberdade nos projetos produzidos mundialmente a mais de oitocentos mil utilizadores, duzentos mil clientes comerciais e empresariais em mais de cento e vinte países [2]. Num caso mais concreto, o *firmware* foi adaptado para ser utilizado no dispositivo **TTGO T-BEAM** que serviu para, de uma maneira acessível, introduzir novos utilizadores ao setor da Internet das Coisas.

1.2 Objetivos

O principal objetivo deste projeto é atualizar este *firmware* para os dispositivos que utilizem ESP32 de modo a que todos os seus utilizadores tenham acesso às funcionalidades introduzidas nas novas versões do LoRaWAN, podendo que estas sejam aplicadas às mais diferentes aplicações no âmbito do IoT.

- Implementar funções em linguagem C de modo a utilizar os novos comandos MAC
- Teste destas funções
- Implementação em Python
- Teste final do *firmware*.

2

Estado da Arte

Neste capitulo procura-se demosntrar as tecnologias essencias ao projeto e ainda o trabalho que já foi realizado no mesmo contexto. Deste modo, este capitulo divide-se num enquadramento da tecnologia a ser utilizada e no trabalho realizado.

2.1 Enquadramento tecnológico

Fazendo uma análise à evolução da tecnologia *Long Range* (LoRa) e do respetivo protocolo LoRaWAN é necessário fazer uma distinção entre ambos os temas e verificar as evoluções mais relevantes para o projeto proposto, sendo que, este terá mais ênfase nas evoluções do LoRaWAN devido ao que é pretendido nesta dissertação.

2.1.1 LoRa

LoRa, é uma tecnologia de modulação rádio usada pelo protocolo LoRaWAN [3] na categoria de tecnologias de rede *Low Power Wide Area Network* (LPWAN) [4]. Esta patente proprietária da **Semtech** implementa, então, a camada física, baseando-se em uma rede com tipologia em estrela [5] e fundamentada numa técnica de modulação denominada *Chirp Spread Spectrum* (CSS) [5].

O CSS é uma técnica de espalhamento espetral que usa impulsos *chirp* modulados em frequência linear para codificar informações [6]. Um *Chirp* é um sinal sinusoidal cuja

frequência aumenta ou diminui com o tempo, dando origem aos denominados *up-chirp* e *down-chirp*. Estas duas designações correspondem ao que se pode observar na Figura 2.1.

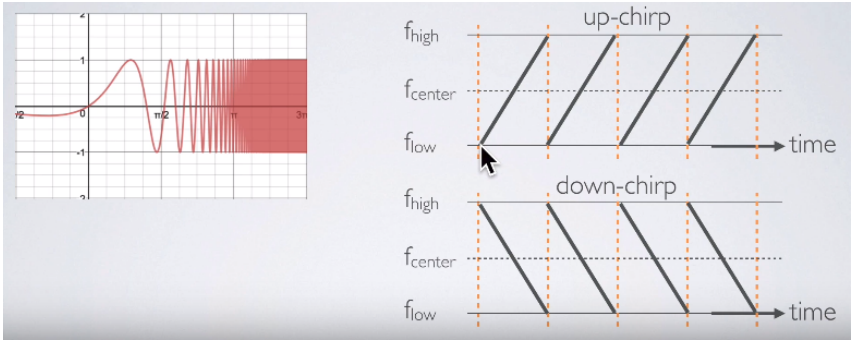


Figura 2.1: Modulação CSS (Extraída de [7])

Esta técnica é usada nas redes LoRa de modo a oferecer um equilíbrio entre sensibilidade e o ritmo de transmissão, operando numa largura de banda (*Bandwidth* (BW)) fixa de 125 kHz, 250 kHz ou 500 kHz para *uplink* e 500 kHz para *downlink*, estas diferenças devem-se à região onde é utilizado. O mais comum em Portugal é a banda de 125 kHz.

Para além da largura de banda, as próprias frequências são diferentes dependendo da região, dado que são frequências não licenciadas, sendo estas: 433MHz e 868 MHz para a Europa, 915 MHz para a Austrália e para a América do Norte, 923 MHz para a Ásia, e de 865 a 867 MHz para a Índia.

LoRa usa fatores de espalhamento, ou *Spreading Factor* (SF) ortogonais, permitindo preservar a bateria dos dispositivos. Estes fatores correspondem ao número de bits de informação por símbolo, sendo o número bits dado por 2^{SF} *chirps*. Quanto maior for o fator de espalhamento, mais *chirps* vão ser usados para representar um símbolo. A rede pode ter valores de SF entre 7 e 12, tendo em conta as condições ambientais em que esta está inserida, quanto menor for o SF maior é o número de *chirps* enviados por segundo, isto devido à maior capacidade de codificar nos SF mais baixos [8]. Ter um fator de espalhamento alto faz com que o tempo de transmissão seja mais prolongado resultando numa maior sensibilidade, porque o recetor tem um tempo de integração maior da potência do sinal. Assim, quanto maior este fator for, mais longe poderá o sinal ser recebido, ou seja, maior será a distância de transmissão.

O **Duty Cycle** representa a fração de tempo em que um recurso está ocupado. No caso da modulação LoRa, num único canal, com um *duty cycle* de 20% ocorreria transmissão de dados em duas partes de tempo em cada dez. Para calcular o valor de *duty cycle* de

um sinal, parte-se da equação 2.1 (*Time On Air* (ToA), que corresponde ao tempo no ar, enquanto $T_{interval}$ corresponde ao tempo de espera entre transmissões):

$$DutyCycle = \frac{ToA}{T_{interval} + ToA} \tag{2.1}$$

A partir desta equação obtemos a fração de tempo em que o pacote é enviado. O valor de $T_{interval}$ é definido pelo utilizador do módulo consoante necessário desde que o *duty cycle* não ultrapasse o limite máximo. Já o ToA é definido à custa do número de bytes a transportar no *payload*, pelo fator de espalhamento, pela frequência definida para cada região e pela largura de banda. Considerando a largura de banda e frequência fixas, ao variar o fator de espalhamento, quanto maior este for, menor será o ritmo de transmissão e maior será o tempo no ar (ToA) (Figura 2.2). Desta forma com o aumento do fator de espalhamento, maior será o respetivo *duty cycle* do sinal.

Tabela 2.1: Regulamentação do *duty cycle* para as respetivas sub-bandas na Europa.

g(863.0-868.0MHz)	1%
g1(868.0-868.6MHz)	1%
g2(868.7-869.2MHz)	0.1%
g3(869.4-869.69MHz)	10%
g4(869.7-870.0MHz)	1%

O *duty cycle* máximo dos dispositivos rádio é, geralmente, regulamentado por autoridades governamentais, sendo que na Europa quem define estes parâmetros é a *European Telecommunications Standards Institute* (ETSI), porém quem controla são as entidades locais, que no caso de Portugal é a ANACOM. No caso da Europa, este valor varia consoante as frequências divididas em 5 grupos (Tabela 2.1).

A modulação LoRa adiciona a técnica *Forward Error Correction* (FEC) na transmissão de informação para estar menos vulnerável ao efeito do ruído no canal e facilitar a descodificação do sinal [7], bem como possibilitar a correção de erros no recetor através da inserção de bits de paridade redundantes a cada bit útil. Esta proporção de bits úteis e de bits de redundância denomina-se *Coding Rate* (CR).

Tipicamente, esta taxa costuma ser de 4/6 ou 4/8. Isto significa que a cada quatro bits úteis, existem dois ou quatro bits de paridade, respetivamente.

A taxa de codificação é ajustada consoante as condições do canal de transmissão. Caso haja uma maior interferência neste, então aumenta-se esta taxa de modo a reduzir o

efeito dos erros no canal. Porém, quanto maior for o valor de CR, maior será a duração de transmissão [9].

Assim, o ritmo binário (Rb) poderá ser descrito pela equação 2.2, concluindo que, quanto maior for a taxa de codificação, menor será o ritmo binário.

$$R_b = SF * \frac{\frac{4}{4+CR}}{\frac{2^{SF}}{BW}} \tag{2.2}$$

Para implementar uma rede é necessário obter um compromisso entre vários fatores, entre eles o consumo energético, o ritmo binário e a distância de transmissão. Neste caso, o protocolo **LoRaWAN** é conhecido pelo alcance de transmissão mais elevado que outras soluções da mesma tipologia devido ao uso do *Adaptive Data Rate* (ADR). Esta técnica consiste na adaptação do ritmo binário consoante o SF, o que tem efeitos no tempo entre a transmissão e receção. Na Figura 2.2 pode-se observar estes efeitos.

Esta técnica é realizada devido à recolha da informação das últimas transmissões e verifica os dados da rede e a qualidade através da relação sinal-ruído e, com isso, realiza um cálculo (Equação 2.3) que obtém a margem do SNR e a distância da margem para o SNR medido. Se a margem for muito superior a 0 indica que está a consumir mais bateria do que a necessária. O SNR_{limite} depende do SF que está a ser usado. Após efetuar este cálculo a rede envia esta informação para o dispositivo e este ajusta os parâmetros.

$$margem = SNR_{medido} - SNR_{limite} - margem_{padrao} \tag{2.3}$$

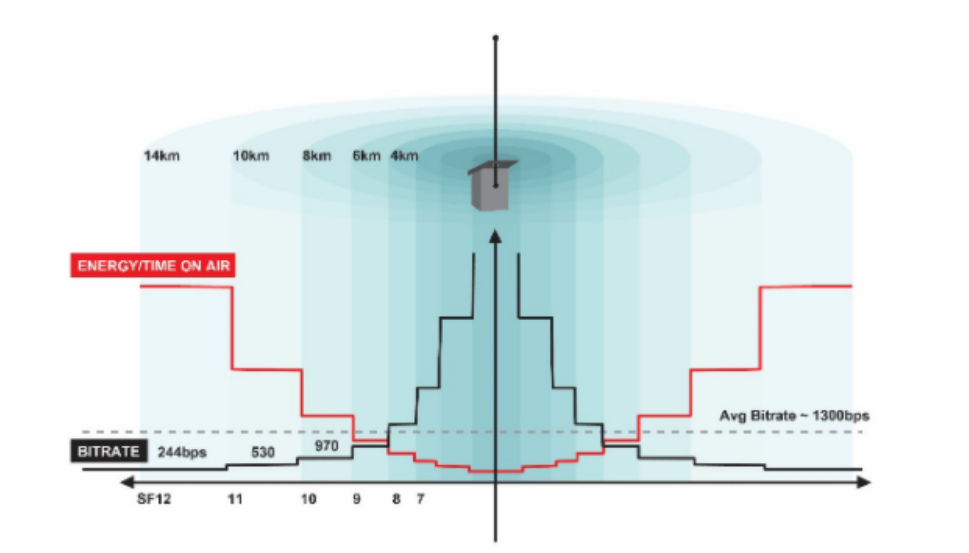


Figura 2.2: Ritmo binário em relação ao fator espalhamento (Extraída de [10])

2.1.2 LoRaWAN

A tipologia LoRaWAN, englobada na categoria LPWAN, veio solucionar a transmissão para longas distâncias para dispositivos com baixo consumo, reduzindo o respetivo ritmo binário. Através da Figura 2.3 é visível o ritmo binário de algumas tecnologias em função do respetivo alcance.

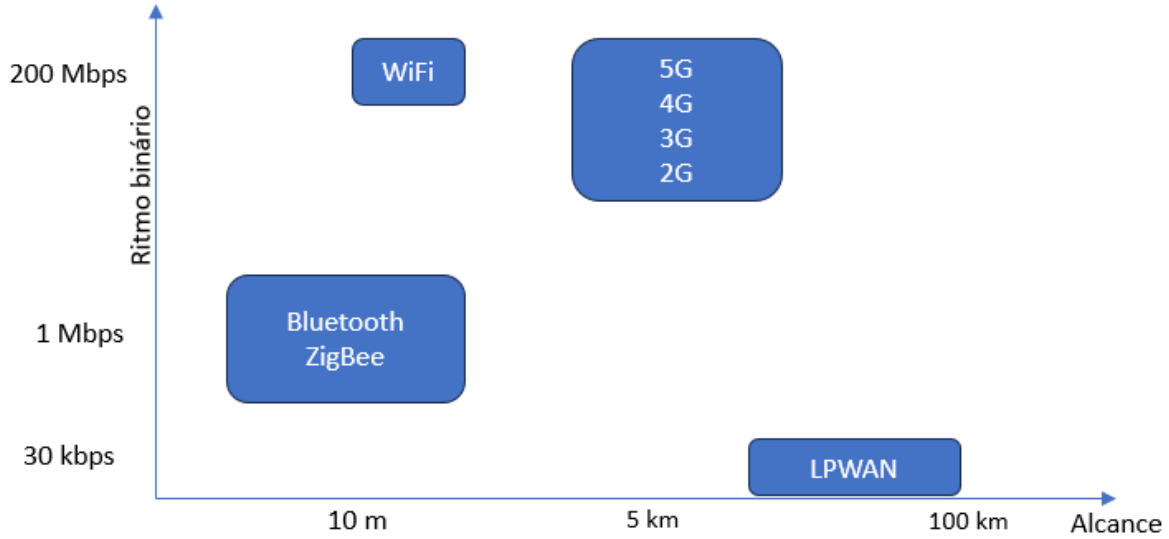


Figura 2.3: Ritmo binário de algumas tecnologias em função do alcance (Adaptado de [3])

Esta tecnologia utiliza bandas de frequência não licenciadas, sendo que a respetiva banda varia da zona geográfica, sendo estas: 433MHz e 868MHz para a Europa, 915MHz para a Austrália e para a América do Norte, 923MHz para a Ásia, de 865 a 867MHz para a Índia, entre outras como se pode observar na Tabela 2.2 [11]. Esta tecnologia é, também, conhecida por ser *open source*.

Tabela 2.2: Frequências consoante a Região

Nome da Região	Banda	Nome	ID
Europa	EU863-870	EU868	1
Estados Unidos da América	US902-928	US915	2
China	CN779-787	CN779	3
Europa	EU433	EU433	4
Austrália	AU915-928	AU915	5
China	CN470-510	CN470	6
Ásia	AS923-1	AS923	7
Ásia	AS923-2	AS923-2	8
Ásia	AS923-3	AS923-3	9
Coreia	KR920-923	KR920	10
India	IN865-867	IN865	11
Russia	RU864-870	RU864	12

A rede tem uma tipologia em estrela, os dispositivos (com o respetivo módulo LoRa) estão diretamente ligados à *gateway* LoRa. Não são usados repetidores devido ao longo alcance que o LoRa permite.

De forma resumida o protocolo LoRaWAN na sua implementação, é usado de modo a transmitir informações de sensores ou de outro tipo de dispositivos através do módulo LoRa para a *gateway* e o contrário também acontece. O *gateway* está ligado à *internet* e quando recebe os dados dos dispositivos envia para um servidor ou para um computador onde serão tratados para a respetiva finalidade.

A trama transmitida tem em si 32 bits correspondentes à identificação do dispositivo, 128 bits para a informação cifrada, 32 bits para a verificação (*Message Integrity Check* (MIC)) e por fim ainda tem o *frame counter*.

Para a segurança da transmissão, o protocolo inclui duas chaves para controlo, *Application Session Key* (AppSKey) e *Network Session Key* (NwkSKey), e ainda o *frame counter*[12].

Começando pela chave AppSKey, esta serve para proteger a informação entre o dispositivo até ao *The Things Network* (TTN) sendo que esta cifra e decifra a informação através do algoritmos **AES128**[12].

Em seguida tem-se a chave NwkSKey, sendo esta necessária entre o módulo e o servidor de rede. Esta chave é a parte da trama MIC mas cifrada através do algoritmo, também, usado para a chave AppSKey. MIC funciona como um *checksum*, ou seja, um somatório das informações do resto da trama (identificação do dispositivo, *frame counter* e a informação útil) de modo a verificar se houve ou não alterações no pacote enviado [12].

Já o *frame counter* tem como objetivo bloquear e detetar ataques de retransmissão de pacotes em excesso (em que o objetivo é sobrecarregar a rede) [12].

2.1.2.1 LoRaWAN 1.0.2

A versão 1.0.2 do protocolo LoRaWAN foi a terceira versão lançada pelo grupo LoRa Alliance em Julho de 2016 [13]. Uma das principais alterações foi a separação dos parâmetros regionais [14], isto é, para cada região do mundo definiu-se a banda de frequências a utilizar, o *duty cycle*, o canal para fazer o JOIN REQUEST e ainda as potências máximas de radiação [11]. As diferenças da banda de frequências tem a ver com as diferentes bandas não licenciadas de cada região. A separação dos parâmetros regionais evita mudanças na base do protocolo [15]. Ainda nesta versão, foram definidos *data rates* de modo a conjugar o fator de espelhamento e a largura de banda e estabelecer um tamanho máximo de *payload* [16].

Outra questão introduzida nesta versão, é a obrigatoriedade de que todos os dispositivos LoRaWAN têm de implementar pelo menos a classe A, sendo esta classe a mais utilizada pois é a que consome menos e a que tem menos interações com a rede. Ainda assim, existe a opção de implementar a classe B e C desde que continue compatível com a classe A [15].

A classe A [17] tem de ser implementada em todos os dispositivos. Os dispositivos de classe A apenas conseguem receber mensagens quando enviam alguma mensagem, quando se faz o *uplink* abrem-se duas janelas temporais para receber as mensagens que vêm em *downlink* (estas com um determinado intervalo entre elas). Deste modo o dispositivo não está sempre à espera de mensagens de *downlink*, apenas as recebe quando decide enviar um pacote.

Já na classe B não só é possível receber pacotes em *downlink* quando se envia uma mensagem em *uplink* como é possível receber sem que haja o envio de um pacote. A *gateway* transmite *beacons* com um determinado intervalo entre eles, o que se chama período de *beacon*. O dispositivo torna-se recetivo a mensagens vindas do servidor a uma determinada altura, esta janela temporal é denominada como *Ping Slot*.

Por fim, na classe C é sempre possível receber mensagens em *downlink* desde que o dispositivo não esteja a transmitir, isto é importante para situações em que seja necessária comunicação com baixa latência.

2.1.2.2 LoRaWAN 1.1

Em Outubro de 2017 [13] a ***LoRa Alliance*** lançou a versão 1.1 do protocolo LoRaWAN, versão que marca uma grande diferença no protocolo, já na versão sugere isso, tendo em conta que as versões anteriores e posteriores a esta versão são 1.0.x. A diferença está na arquitetura, esta versão tem uma arquitetura diferente relativamente à arquitetura das restantes versões.

Na arquitetura das versões 1.0.x, os dispositivos conectavam-se a uma ou mais *gateways* para comunicar e estas estariam ligadas a um servidor de rede, como por exemplo o TTN, e este, por sua vez, passaria os dados para a aplicação, sendo assim, uma arquitetura com topologia de rede em estrela, como é possível ver na Figura 2.4. Apenas as ligações entre dispositivos em *gateways* são suportados por LoRa, sendo que o restante das ligações são por IP.

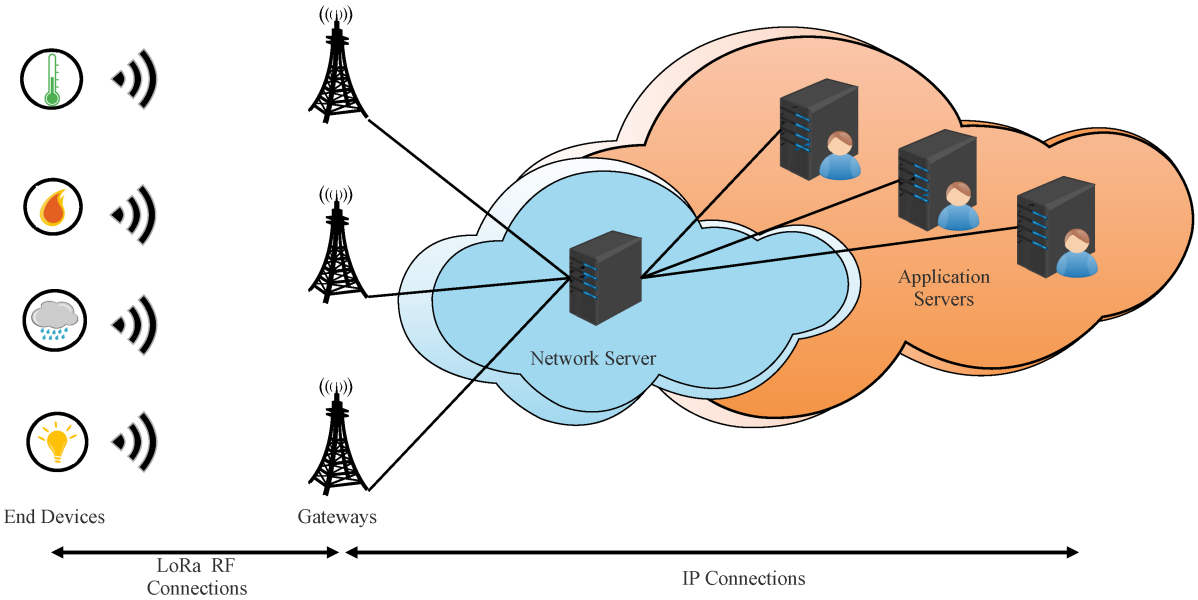


Figura 2.4: Arquitetura LoRaWAN 1.0.x (Extraído de [18])

Passando à arquitetura da versão 1.1, pode-se observar na Figura 2.5 que existem novos elementos na estrutura da rede. Na parte da rede observa-se que o servidor de rede foi dividido em três servidores, estes são o *Home Network Server* (hNS), o *Serving Network Server* (sNS) e o *Forwarding Network Server* (fNS). No que respeita à outra diferença face à arquitetura vista anteriormente tem-se o *Join Server* (JS).

O hNS controla os pacotes de *uplink* e de *downlink* e ainda controla a camada MAC. Este servidor também guarda os perfis de encaminhamento, de serviço, de dispositivo e ainda guarda o identificador único do dispositivo (DevEUI). Os outros dois servidores (sNS e fNS) servem para permitir o *roaming* nesta nova versão. Há dois tipos de *roaming*, daí estes dois servidores, que são o *handover roaming* e o *passive roaming*. No caso do *handover roaming* o sNS assume os controlos que do hNS, já no caso do *passive roaming* o servidor fNS apenas recebe pacotes entre o dispositivo e o sNS. O sNS mantém o controlo da camada MAC. Para isto ocorrer é necessário o fNS estar ligado ao sNS e o sNS ligado ao hNS, tal como se pode observar na Figura 2.5.

Por fim, o *Join Server* é o responsável por todas as chaves de segurança quer sejam para a camada de aplicação quer sejam chaves para a camada de rede. Este novo elemento traz mais segurança ao protocolo. Este servidor assume a responsabilidade, que anteriormente era do *Network Server* (LoRaWAN 1.0.x), de tratar do processo de ativação *Over-The-Air-Activation* (OTAA).

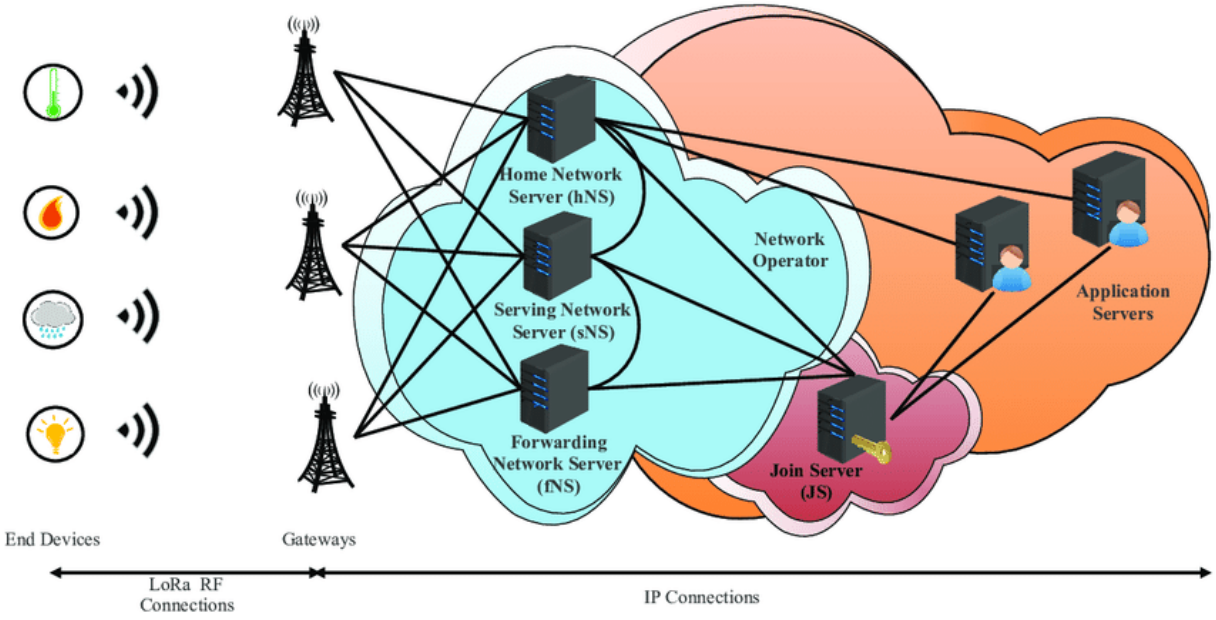


Figura 2.5: Arquitetura LoRaWAN 1.1 (Extraído de [18])

Esta nova versão é marcada pelas melhorias de segurança e pelas novas funcionalidades.

Começando pelas melhorias de segurança, com a introdução do *Join Server* na arquitetura são realizadas algumas mudanças no processo de OTAA de modo a torná-lo mais seguro. Aparece uma nova chave *Network Key* (NwkKey) que, tal como a *Application Key* (AppKey), é cifrada com AES-128 [19]. É "substituído"o AppEUI pelo JoinEUI.

Já no modo de ativação *Activation By Personalisation* (ABP) passa de duas chaves para quatro. Nas versões 1.0.x existem apenas as chaves NwkSKey e o AppSKey, passando para a versão 1.1 mantém-se o AppSKey, desaparece o NwkSKey e aparece a Forwarding Network Session Integrity Key (FNwkSIntKey), a *Serving Network Session Integrity Key* (SNwkSIntKey) e *Network Session Encryption Key* (NwkSEncKey).

Estas novas chaves têm funções diferentes [20][21][22]:

- SNwkSIntKey - É usada para verificar o MIC de todas as mensagens de *downlink*. Também usada para calcular metade do MIC das mensagens de *uplink*.
- NwkSEncKey - É usada para cifrar e decifrar os comandos MAC de *uplink* e *downlink*. Os comandos MAC são transmitidos como se fosse um *payload*.
- FNwkSIntKey - É usada para calcular o MIC de todas as mensagens de *uplink*. Por vezes pode calcular apenas metade do MIC.

Para questões de compatibilidade, quando se liga a um servidor de rede com a versão 1.0 do LoRaWAN a chave SNwkSIntKey fica com o mesmo valor da chave FNwkSIntKey, tal como a NwkSEncKey.

Uma das grandes novidades desta nova versão do LoRaWAN está nos novos comandos MAC. As diferenças destes comandos começam na formatação das mensagens MAC.

Na versão 1.1 o PHYPayload passa a fazer *Join-Request* ou *Rejoin-Request* e a única resposta será o *Join-Accept* e o MIC é cifrado juntamente com o *payload*, enquanto na versão 1.0.2 era enviado um *Join-Response* com o MIC num campo separado.

Outra das diferenças é que o FOpts (que transporta comandos MAC até 15 octetos) tem de ser cifrado depois do cálculo do MIC e será cifrado com AES128 em que a chave será a NwkSEncKey.

Também se faz distinção no cálculo do MIC para *downlink frames* e *uplink frames*. No *downlink* é usada apenas a SNwkSIntKey enquanto para o *uplink* é usada a SNwkSIntKey e a FNwkSIntKey

Passando aos novos comandos MAC [21]:

- ResetInd - Comando enviado pelo dispositivo, em que este usa ABP, para indicar um *reset* à rede e à negociação da versão do LoRaWAN.
- ResetConf - Comando enviado pelo *gateway* para confirmar o ResetInd.

- RekeyInd - Comando enviado pelo dispositivo, em que este usa OTAA, para confirmar a atualização da chave de segurança.
- RekeyConf - Comando enviado pelo *gateway* para confirmar o RekeyInd.
- ADRParamSetupReq - Comando enviado pelo *gateway* onde são definidos parâmetros o ADR_ACK_LIMIT e o ADR_ACK_DELAY.
- ADRParamSetupAns - Comando enviado pelo dispositivo para confirmar o ADRParamSetupReq.
- DeviceTimeReq - Comando enviado pelo dispositivo para pedir a data e a hora.
- DeviceTimeAns - Comando enviado pelo *gateway* com a data e a hora.
- ForceRejoinReq - Comando enviado pelo *gateway* a pedir ao dispositivo que envie imediatamente um *Rejoin-Request* com um número de tentativas, periodicidade e *data rate* programáveis.
- RejoinParamSetupReq - Comando enviado pelo *gateway* para definir a periodicidade das mensagens de *Rejoin* do dispostivo.
- RejoinParamSetupAns - Comando enviado pelo dispositivo para confirmar o RejoinParamSetupReq.

2.1.2.3 LoRaWAN 1.0.4

Passando à última versão publicada, até então, do protocolo LoRaWAN mencionar-se-à algumas atualizações que fazem parte da publicação da versão 1.0.3 porém também estão incluídas na versão 1.0.4.

Começando pelas melhorias de segurança é implementada uma medida que obriga *frame counters* de 32 bits e estes têm de ser guardados numa parte da memória que se possa aceder após o processo estar concluído. Para guardar estes *frame counters* utiliza-se uma parte da memória conhecida como *Non-Volatile Random Access Memory* (NVRAM). Esta medida faz com que os contadores não se percam em caso de *reboot*.

Nesta versão foram implementados alguns pormenores semelhantes à versão 1.1, como por exemplo, o AppEui que passa a chamar-se JoinEui e passa a usar o JS.

Como melhorias para os dispositivos que operem no modo classe B, é introduzido e explicado como se usa o *Frame Pending bit* sendo que este bit no *downlink* para dispositivos classe B é ativado de modo a ter prioridade no caso de colisão entre *pings*.

Para além disso, os *Beacon Frame Formats* passam a ser especificados do SF8 ao SF12. Tendo em conta a necessidade de sincronização entre *beacons* de classe B melhorou-se a precisão da sincronização com o relógio do GPS de modo a que a precisão seja de 1 μs. Se esta condição for respeitada a *gateway* transmite o *beacon* de 128 em 128 segundos. Também é implementada uma medida que previne o acontecimento de confirmações de *downlink* de dispositivos classe B e C após o *timeout*.

Para finalizar, houve alguns pontos que foram retificados nesta nova versão. Começando pela co-existência das diferentes classes, a gestão do ADR, tratar os comandos MAC como obrigatórios, as mensagens de *uplink* são retransmitidas caso não seja recebida nenhuma confirmação do *Network Server* ou do *Application Server*.

2.2 Trabalho Relacionado

Nesta secção pretende-se mostrar trabalho relacionado com o projeto. No caso, são implementações de *firmwares* usados em diferentes microcontroladores e que tenham implementadas diferentes versões do LoRaWAN de modo a averiguar se o objetivo desta dissertação foi conseguido em ambientes semelhantes.

2.2.1 Mbed OS 5.8

Mbed é uma plataforma e sistema operativo para dispositivos baseados no microcontrolador ARM Cortex-M de 32 bits, sendo estes dispositivos usados para desenvolver projetos no ramo do IoT. O Mbed OS é sistema operativo de uso livre que oferece conectividade, *drivers* para *hardware* e um sistema operativo em tempo real.

A 28 de Março de 2018 foi lançada a versão 5.8 onde foi feito um investimento nas redes LPWAN ao trazer uma atualização no âmbito destas redes de baixo consumo e longa alcance. Nesta atualização foi introduzido o *stack* LoRaWAN ao sistema operativo na versão 1.0.2. Ainda nesta atualização foi introduzido o NB-IoT e o CAT-M1 o que demonstra o ênfase desta plataforma nas aplicações da Internet das Coisas [23].

No dia 3 de Abril de 2018, poucos dias depois da primeira versão da atualização 5.8, foi lançada uma especificação em relação ao *stack*LoRaWAN [24].

Esta necessidade de introduzir o *stack* de LoRaWAN deve-se ao investimento feito pela Arm nesta área e à empresa que criou a modulação LoRa, a Semtech, que publica o *stack* na plataforma Mbed. Até então, os utilizadores (comerciais ou não comerciais) tinham

de desenvolver o seu próprio *stack* LoRaWAN. Esta versão do Mbed OS 5.8 facilita a o desenvolvimento das aplicações do consumidor final por encurtar o tempo que este perderia a criar o seu próprio *stack* e ainda seria pior tendo em conta a complexidade acrescida da versão 1.1 do protocolo LoRaWAN.

Passado um ano da especificação referida no parágrafo anterior, no dia 8 de Abril de 2019 foi lançada uma especificação com a introdução à versão 1.1 do protocolo LoRaWAN onde é descrito como se implementa esta versão do protocolo na versão 5.8 do Mbed OS.

Devido à popularidade do LoRaWAN na comunidade de utilizadores de Mbed foi lançada esta atualização para fortalecer a preferência e melhorar o ecossistema de utilização das tecnologias LoRa.

Para além de adicionar o LoRaWAN 1.1, foram também adicionadas funcionalidades para suportar a versão LoRaWAN 1.0.3.

Ainda que a Mbed faça para da **LoRa Alliance**, foram necessários dois anos para que os dispositivos que usem Mbed OS possam utilizar a versão 1.1 do LoRaWAN.

2.2.2 STM32CubeWL

A STMicroeletronics é uma empresa criada em 1987 no setor da eletrónica, como o próprio nome revela. Dedicava-se à produção de semicondutores, porém, hoje em dia, o seu foco é o IoT e a conectividade criando diversos produtos e soluções para as aplicações IoT [25][26]. Esta empresa é patrocinadora e faz parte do grupo LoRa Alliance [27].

O STM32CubeWL foi criado pela STMicroelectronics. É um *firmware* criado para melhorar a produtividade dos utilizadores dos dispositivos STM32, como por exemplo o STM32WL que é um módulo que está preparado para implementar LoRa. O STM32CubeWL está implementa LoRaWAN e SigFox nas tecnologias de rede LPWAN. Este *firmware* é de uso gratuito e tem condições de utilização acessíveis ao utilizador.

O pacote MCU do STM32CubeWL é composto por uma camada de abstração de *hardware* (HAL) e uma camada das APIs (LL). O STM32CubeWL HAL é uma camada de *software* embebida que assegura as funcionalidade do *firmware* independentemente do dispositivo STM32 que esteja a usar o STM32CubeWL. Já o STM32CubeWL LL é uma camada mais próxima do *hardware* resultando numa camada rápida, leve e orientada

a especialistas. O HAL e o LL podem ser usados em simultâneo caso se cumpram determinadas restrições.

Normalmente, o LL é usado para soluções que necessitem de uma grande desempenho e que ainda assim seja leve. O HAL peca na desempenho para ser simples.

Passando para a parte do LoRa, com este *firmware* é permitido usar diferentes modulações com o LoRa, sendo estas o GFSK, GMSK e o BPSK.

Neste *firmware* é possível usar as versões mais recentes do protocolo LoRaWAN.

2.2.3 LMIC

LMIC é uma implementação do protocolo LoRaWAN realizado pela IBM. LMIC significa LoRaMAC-In-C, ou seja, uma implementação dos comandos MAC do LoRaWAN em linguagem de programação C. Esta biblioteca é bastante usada em Arduinos.

A *International Business Machines Corporation* (IBM) é um empresa dos Estados Unidos que está presente no setor da tecnologia fundada em 1911.

Têm o foco na produção e venda de *hardware* e *software* para qualquer tipo de tecnologia, vai desde tecnologia industrial à nanotecnologia.

Ainda assim, a versão original da LMIC foi descontinuada o que fez com que duas entidades pegassem na versão original adaptando-a e realizando atualizações. As duas entidades são a MCCI Catena e a LacunaSpace, ambas utilizaram o código original da IBM e foram implementadas de modo a ser usado em dispositivos Arduino.

No caso da MCCI Catena [28], implementa-se apenas a classe A e B, tal como no código original, porém a classe B não foi testada em funcionamento. Nesta versão, só está implementado o protocolo LoRaWAN 1.0.2 e 1.0.3. Esta biblioteca ainda auxilia novos módulos que a versão original não conseguia ao ser implementada.

Passando à versão desenvolvida pela LacunaSpace [29], esta para além das classes implementadas na versão da IBM implementa a classe que resta, a classe C. Devido ao tamanho desta versão há placas que deixam de poder utilizar esta biblioteca.

Esta biblioteca é focada em trazer atualizações das funcionalidades LoRaWAN porém ainda não implementa a versão 1.1 do protocolo em questão.

Tabela 2.3: Comparação das duas versões LMIC

	Classes	**Tamanho (face à biblioteca original)**	**Última Publicação**
MCCI Catena	A e B	Aproximadamente igual	27 de Dezembro 2021
Lacuna Space	A, B e C	Superior	16 de Abril 2020

2.2.4 Pycom

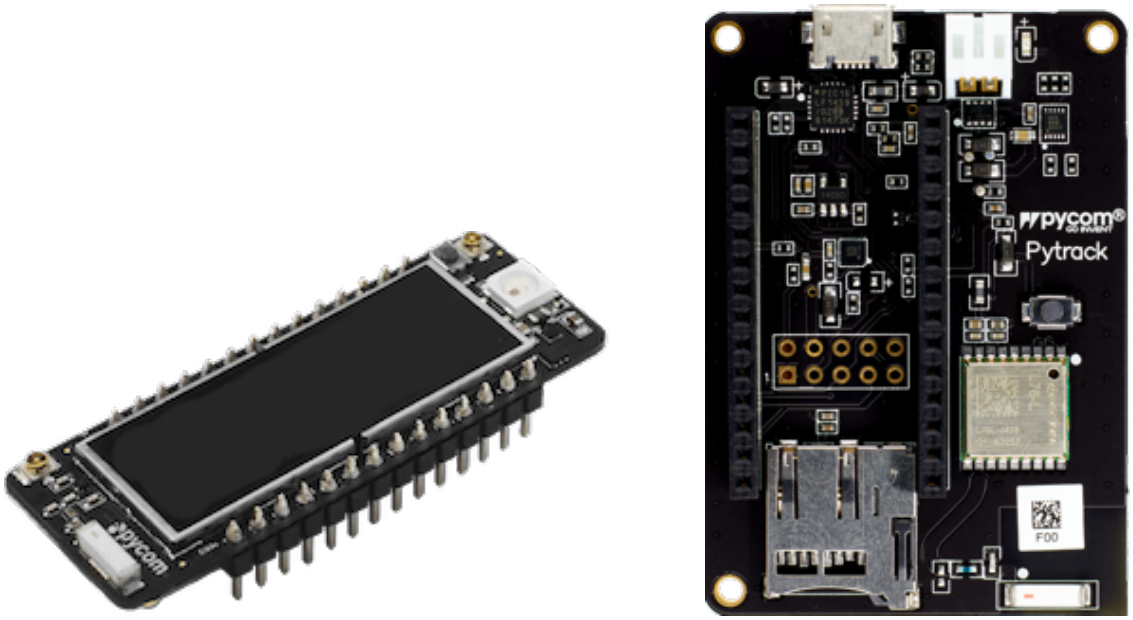
A Pycom é uma empresa internacional focada nas tecnologias relacionadas com o IoT. Esta empresa teve o seu primeiro produto disponibilzado em Abril de 2015 através do apoio financeiro da comunidade, o seu primeiro produto foi o WiPy. O seu foco inicial foi desenvolver um ambiente de fácil acesso e que fosse facilmente programável com intuito dos seus desenvolvedores serem qualquer pessoa que tenha interesse na área de IoT e que acabe por criar o seu próprio projeto.

Já com vários produtos a serem comercializados, a Pycom decidiu continuar com a sua visão e dar sempre as ferramentas para que o utilizador possa escolher as versões e opções que mais se adequam ao projeto que este pretende realizar. Estes projetos podem ser tantos projetos comerciais de variadas dimensões ou apenas um projeto pessoal para entretenimento.

Como exemplo, empresas de grande dimensão como a Vodafone, Cisco, Google, Amazon e diversas instituições de ensino utilizam os dispositivos da Pycom para desenvolver os seus protótipos.

Uma das vantagens dos dispositivos Pycom é a versatilidade que introduzem no mercado. Esta empresa tem diversos módulo de desenvolvimento que implementam diversas tecnologias de comunicação, quer seja LoRa, BLE, redes móveis, Wi-Fi, entre outras. A versatilidade vem da possibilidade de utilizar qualquer um destes módulos de desenvolvimento com um placa que já vem com sensores e outros módulos mais vocacionados com o propósito do projeto, por exemplo, módulo GPS, acelerómetro, sensor de temperatura e humidade, entre outros. Com este junção de equipamentos pode-se utilizar qualquer tecnologia de comunicação com os sensores que necessários consoante os requisitos do projeto que se pretende criar.

Como se observa na Figura 2.6, o objetivo é encaixar o módulo de desenvolvimento no *shield*. Todos os módulos têm o mesmo *pinout* para poder ser ligado a qualquer placa desenvolvida pela Pycom.



(a) Módulo de Desenvolvimento FiPy (b) *Shield* Pytrack

Figura 2.6: Dispositivos da Pycom

Outra das vantagens do ambiente proporcionado pela Pycom é o facto do *firmware* dos módulos de desenvolvimento serem *open-source*. Qualquer pessoa pode acompanhar a evolução do *firmware* através do Github da Pycom e até fazer as alterações que se ache necessário para qualquer que seja aplicação.

O *firmware* é programado em C e é bastante complexo portanto não é aconselhável que sejam efetuadas alterações sem um estudo prévio e uma análise cuidada. Ainda assim, o utilizador comum apenas irá programar em MicroPython, sendo esta é a linguagem de programação utilizada para as aplicações que são executadas nestes dispositivos da Pycom.

Esta conversão de C para MicroPython é responsável por alguma da complexidade do *firmware*.

Para além do que já foi mencionado, o sucesso da Pycom vem da preocupação com o utilizador. Através do site da Pycom, pode-se fazer *download* das bibliotecas necessárias para a placa que se pretende utilizar e ainda um exemplo de código em MicroPython para começar a utilizar os sensores ou os meios de comunicação.

3

Implementação da nova versão do protocolo LoRaWAN

Neste capítulo pretende-se mostrar o trabalho realizado ao longo do projeto. Para começar, foi necessário estudar qual era a composição do *firmware* nas versões anteriores com o objetivo de perceber como está organizado e quais são as partes que envolvem o protocolo LoRaWAN. Deste modo, as alterações realizadas no *firmware* são mais focadas e mais eficientes.

Ao comparar o *firmware* com a versão 1.0.2 do protocolo com a versão 1.0.3 para compreender as alterações realizadas observou-se que na versão 1.0.3 faltava implementar as novas funcionalidades na zona de código que gere o ESP32. Tendo em conta a relevância destas funcionalidades e para adaptação ao ambiente do projeto, optou-se por implementar estas funcionalidades na versão 1.0.3. Sendo que este código seria aproveitado para a atualização para a versão 1.0.4.

Para atualizar da versão 1.0.3 para a versão 1.0.4, foi necessário alterar tanto o protocolo LoRaWAN como a transição de C para MicroPython (fichero modlora.c) e ainda algumas alterações na parte rádio.

3.1 Arquitetura do firmware e ambiente de desenvolvimento

O *firmware* que está a ser utilizado como base é o pycom-micropython-sigfox da Pycom [30]. No Github da Pycom encontram-se versões do *firmware* tanto com o protocolo LoRaWAN 1.0.2 como com o protocolo LoRaWAN 1.0.3, sendo que esta segunda versão ainda está em beta.

A composição geral da biblioteca passa por quatro principais secções do código. Utilizando o nome das pastas, são estas a pasta dos **`drivers`**, a pasta do controlador ESP32, a pasta **`lib`** referente às bibliotecas a serem utilizadas e ainda a pasta **`mods`** (integrada na pasta **`esp32`**) com funções em C para utilização do MicroPython.

A pasta dos **`drivers`** engloba o código desenvolvido para utilização das placas que transmitirão e receberão as mensagens LoRa, ou seja, estes *drivers*, no ambiente LoRa, gerem o dispositivo do ponto de vista rádio. Neste *firmware* são suportados os *chips* LoRa SX1276 e SX1272.

As bibliotecas presentes na pasta **`lib`** são usadas como base de funcionamento para o ESP32. No caso do protocolo LoRaWAN, é nesta pasta que se encontra a *Stack* do protocolo em que está dividida entre a parte *Medium Access Control* (MAC) e a interligação da *Stack* com o ESP32. A *Stack* LoRaWAN vai aumentado a complexidade ao longo das versões que foram sendo publicadas. Uma maneira fácil de observar esta evolução é analisar a quantidade de ficheiros que vão sendo adicionados e as ligações que vão sendo feitas entre estes ficheiros. Ao analisar a versão 1.0.2 para a versão 1.0.3 observa-se claramente este acréscimo de complexidade. Na versão 1.0.2, existem apenas cinco ficheiros sendo que três deles são *headers* (.h). Já na versão 1.0.3 existe um total de vinte e três ficheiros. Este aumento de ficheiros deve-se essencialmente aos novos MAC *Commands* e às novas funcionalidades com as diferentes classes de dispositivos LoRa.

Finalizando com a pasta que utiliza as bibliotecas anteriormente mencionadas para integrar com o microcontrolador ESP32. Esta parte do *firmware* trata de todos os pontos fulcrais para o funcionamento do microcontrolador com novos *chips* para as mais variadas tecnologias. Esta parte do código está preparada para ser versátil e conseguir utilizar o mesmo código para as diferentes placas que utilizem um ESP32. A parte mais relevante para o contexto deste projeto é a passagem das funções escritas em C para MicroPython. Esta passagem para o LoRa encontra-se nos ficheiros modlora.c e modlora.h. Criam-se funções que serão chamadas em MicroPython que utilizam funções da Stack LoRaWAN, isto exige processamento e variações a nível de *software* para as diferentes versões do protocolo LoRaWAN.

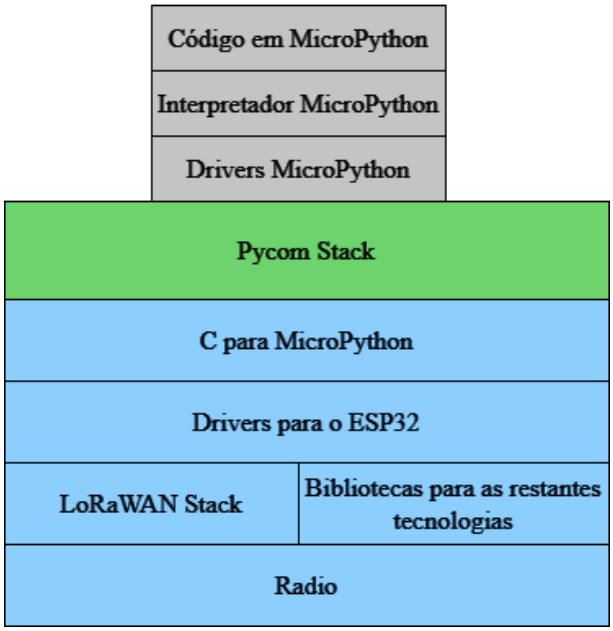


Figura 3.1: Arquitetura do *firmware*

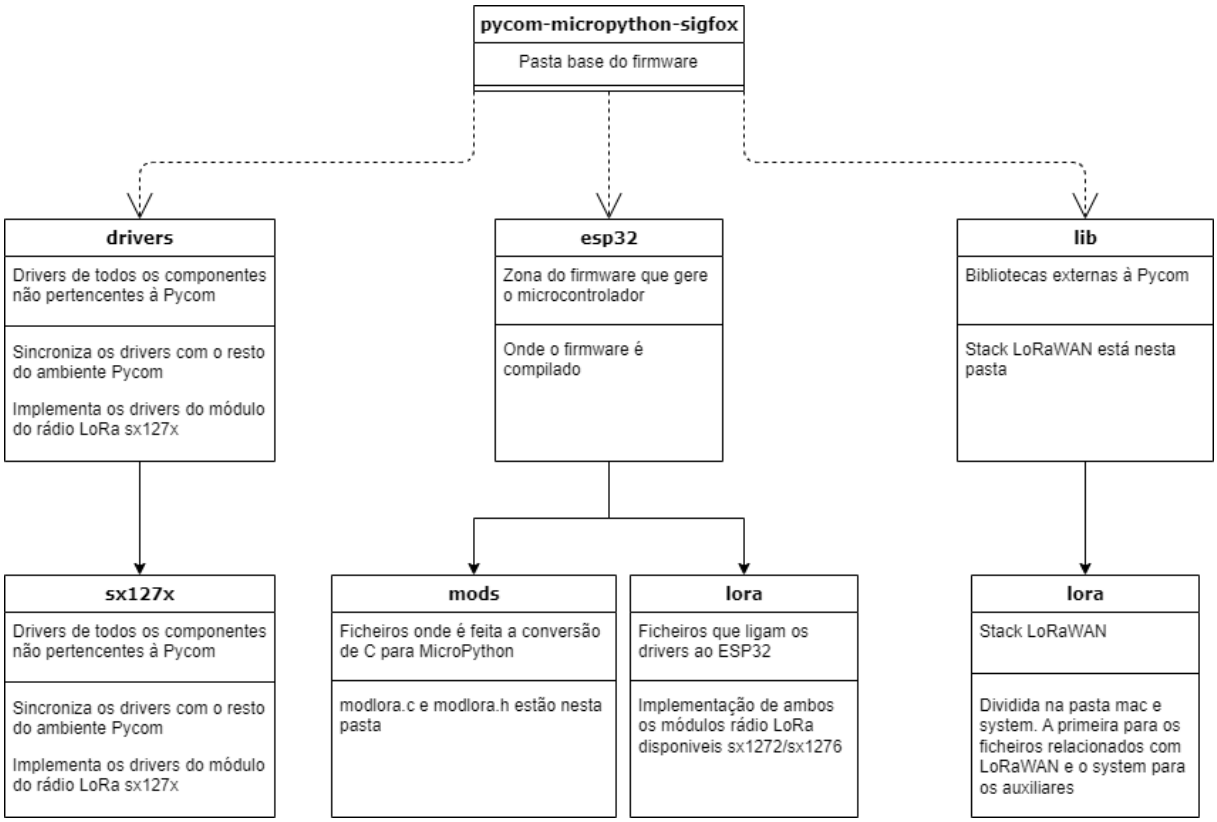


Figura 3.2: Diagrama de pastas do *firmware*

Após a análise do *firmware*, preparou-se o ambiente utilizado durante este processo. O código foi desenvolvido e compilado numa máquina virtual com o sistema operativo

Linux com a versão Ubuntu 22.0.4.1 LTS. Foi utilizado este ambiente pois já era um ambiente familiar e mais confortável para enfrentar o desafio.

Inicialmente, utilizou-se a última versão disponibilizada pela Pycom da biblioteca pycom-micropython-sigfox, a versão 1.20.2.r6 que implementa a versão 1.0.2 do protocolo LoRaWAN. Esta versão do *firmware* é usada pela maioria dos utilizadores do dispositivo e já foi testada e finalizada. Para compilar o *firmware* é necessário utilizar a biblioteca pycom-esp-idf na versão 3.3.1. Esta biblioteca serve para o funcionamento básico do ESP32.

Após preparar o ambiente testou-se compilar o *firmware* e instalar o mesmo no dispositivo. Para este efeito seguiu-se os passos indicados no Github referente a esta biblioteca. O teste foi dado como concluído ao verificar que a versão do *firmware* do dispositivo correspondia à versão que tinha sido instalada. Este teste tem como objetivo comprovar o funcionamento do ambiente para prosseguir com o projeto.

3.2 Análise das versões existentes do firmware

No repositório da Pycom no Github encontra-se uma versão beta do *firmware* que implementa a versão 1.0.3 do *Stack* do protocolo LoRaWAN, esta versão é fundamental para o objetivo final do projeto. Ao ter versões diferentes do *firmware* com diferentes versões do protocolo LoRaWAN é possível comparar entre as mesmas e verificar o que foi alterado e utilizar este conhecido no decorrer do projeto.

3.2.1 Evolução do protocolo LoRaWAN no *firmware*

A versão 1.0.3 do protocolo LoRaWAN apresenta muitas mudanças a nível de *Stack*. Na versão 1.20.3.b3 da biblioteca pycom-micropython-sigfox da Pycom está implementada essa versão do LoRaWAN, sendo que a versão do *firmware* ainda é uma versão beta. Em comparação com a versão 1.20.2.r6 da biblioteca pycom-micropython-sigfox, a versão do ESP-IDF passa da verão 3.3.1 para a versão 4.1.

Nesta fase foi onde se observou a composição do *firmware* e onde se realçou as partes do *firmware* mais relevantes para o contexto do projeto, no caso as partes intervenientes para a utilização de LoRa/LoRaWAN. Com base nesta análise obteve-se a Figura 3.2.

Sendo uma versão beta, a tecnologia LoRa foi testada com a versão 1.0.3 do protocolo LoRaWAN. Certas funcionalidades não estavam implementadas, porém, para enviar e receber mensagens normais este *firmware* está funcional.

3.2.2 Alterações efetuadas ao firmware

Sendo uma versão beta ainda não se espera que esteja completa, porém, esta versão é funcional, permitindo fazer tudo o que era possível na versão anterior mesmo com a *Stack* LoRaWAN atualizada. Já as novas funcionalidades da versão 1.0.3 não foram introduzidas no ficheiro modlora o que impossibilita a utilização das mesmas. Devido à utilidade de uma das funcionalidades e à mais valia em termos de aprendizagem para o trabalho futuro optou-se por criar as funções que permitem usufruir desta novidade. Esta funcionalidade permite receber a data e as horas através de LoRa.

Esta informação é crucial para diferentes projetos na área do IoT. Esta informação, geralmente, implica ou um módulo GPS adicional ou um módulo de WiFi (que poderá ser desligado logo após receber a informação para reduzir o consumo) para atualizar a data. Contudo, com esta função do LoRaWAN, consegue-se obter essa informação com o módulo de comunicação que será usado, ou seja, o módulo LoRa.

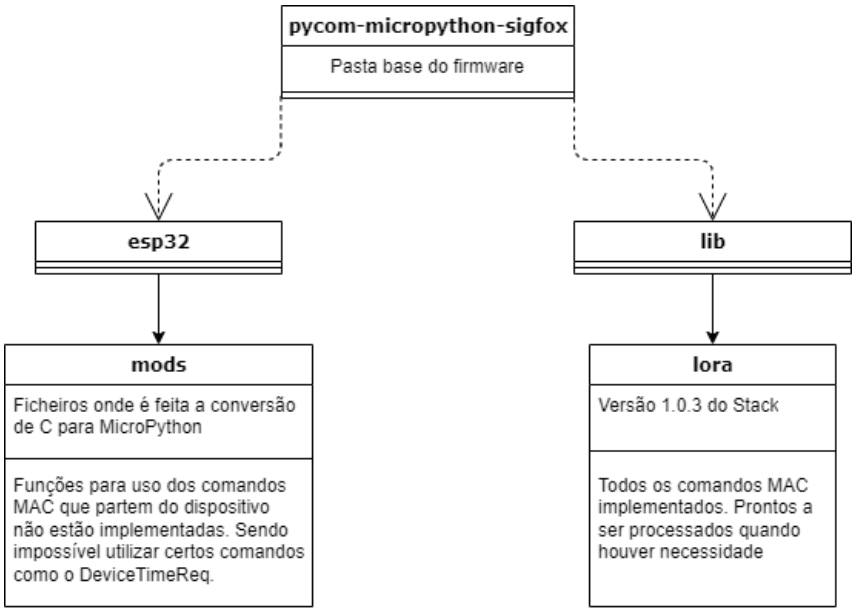


Figura 3.3: Estrutura de pastas/ficheiros

Como é possível ver nas listagens 3.1 e 3.2 [31], na *Stack* já estava implementada esta possibilidade através de um pedido enviado para a *gateway*. Posteriormente é recebida a resposta com data e hora atualizada. No caso do dispositivo ser de classe A, só quando houver um *uplink* da parte do dispositivo é que o mesmo recebe um *downlink* com a resposta.

```
4546 case MLME_DEVICE_TIME:
4547 {
4548     // LoRaMac will send this command piggy-pack
4549     status = LORAMAC_STATUS_OK;
4550 
4551     if( LoRaMacCommandsAddCmd( MOTE_MAC_DEVICE_TIME_REQ, macdPayload, 0 ) !
     = LORAMAC_COMMANDS_SUCCESS )
4552     {
4553         status = LORAMAC_STATUS_MAC_COMMAD_ERROR;
4554     }
4555     break;
4556 }
```

Listagem 3.1: Pedido de atualização de data e hora (Extraído do ficheiro LoRaMac.c da versão 4.7 do repositório da LoRaMacNode)

```
2146 case SRV_MAC_DEVICE_TIME_ANS:
2147 {
2148     SysTime_t gpsEpochTime = { 0 };
2149     SysTime_t sysTime = { 0 };
2150     SysTime_t sysTimecurrent = { 0 };
```

```
2151
2152        gpsEpochTime.Seconds = (uint32_t )payload[macIndex++];
2153        gpsEpochTime.Seconds |= (uint32_t )payload[macIndex++] << 8;
2154        gpsEpochTime.Seconds |= (uint32_t )payload[macIndex++] << 16;
2155        gpsEpochTime.Seconds |= (uint32_t )payload[macIndex++] << 24;
2156        gpsEpochTime.SubSeconds = payload[macIndex++];
2157
2158        // Convert the fractional second received in ms
2159        // round( pow( 0.5, 8.0 ) * 1000) = 3.90625
2160        gpsEpochTime.SubSeconds = (int16_t )( ( (int32_t )gpsEpochTime.
           Subseconds * 1000) >> 8);
2161        // Copy received GPS Epoch time into system time
2162        sysTime = gpsEpochTime;
2163        // Add Unix to Gps epcoh offset. The system time is based on Unix time.
2164        sysTime.Seconds += UNIX_GPS_EPOCH_OFFSET ;
2165        // Compensate time difference between Tx Done time and now
2166        sysTimeCurrent = SysTimeGet( );
2167        sysTime = SysTimeAdd( sysTimeCurrent, SysTimesub( sysTime, Macctx.
           LastTxSysTime ) );
2168        // Apply the new system time.
2169        SysTimeset( sysTime );
2170        LoRaMacClassBDeviceTimeAns( );
2171        MacCtx.McpsIndication.DeviceTimeAnsReceived = true;
2172        break;
2173        }
```

Listagem 3.2: Resposta ao pedido de atualização de data e hora (Extraído do ficheiro LoRaMac.c da versão 4.7 do repositório da LoRaMacNode)

Para utilizar estas implementações da *Stack* foi necessário criar umas outras funções para poderem ser chamadas, em MicroPython, pelo utilizador final. Ao perceber como funcionava esta ligação entre C e MicroPython desenvolveram-se as seguintes funções (Listagem 3.3).

```
1  STATIC mp_obj_t lora_reqdevtime (mp_obj_t self_in) {
2      MlmeReq_t mlmeReq;
3      mlmeReq.Type = MLME_DEVICE_TIME;
4      if((LoRaMacMlmeRequest (&mlmeReq)) == LORAMAC_STATUS_OK){
5          return mp_const_true;
6      }
7
8      return mp_const_false;
9  }
10 STATIC MP_DEFINE_CONST_FUN_OBJ_2(lora_reqdevtime_obj, lora_reqdevtime);
11
```

```
mp_obj_t lora_getdevtime (mp_obj_t self_in) {
    static const qstr lora_compliance_devTime_fields[] = {
        MP_QSTR_enabled, MP._QSTR_running
    };

    SysTime_t time = SysTimeGet();

    mp_obj_t compliance_tuple[2];
    compliance_tuple[0] = mp_obj_new_float (time.Seconds);
    compliance_tuple[1] = mp_obj_new_float(time.SubSeconds);
    return mp_obj_new_attrtuple(lora_compliance_devTime_fields, 2,
    compliance_tuple);
STATIC MP_DEFINE_CONST_FUN_OBJ_2(lora_getdevtime_obj, lora_getdevtime);
```

Listagem 3.3: Funções criadas para implementação de uma nova funcionalidade

Por fim, testou-se estas mesmas funções na ótica de utilizador comum dos dispositivos. Na listagem 3.4 observa-se o código desenvolvido em MicroPython para chamar as funções enquanto na figura 3.4 é apresentado o resultado de todo este processo.

```
s = socket.socket (socket.AF_LORA, socket.SOCK_RAW)
s. setsockopt (socket.SOL_LORA, socket.SO_DR, 5)
ret = lora.reqdevtime (10)
s. setblocking (True)|
s. send (bytes([0x01, 0x02, 0x03]))
s.setblocking (False)
data = s. recv (64)
epoch = lora.getdevtime (10)
tempo = utime. localtime(int(epoch[0]))
print (tempo)
```

Listagem 3.4: Pedido de atualização de data e hora

```
Waiting to join
Network joined!
GOT HERE!
(2023, 9, 19, 7, 15, 12, 1, 262)
b''
```

Figura 3.4: Resultado do teste

3.3 Implementação da versão 1.0.4

Ao passar pelo processo de aprendizagem com a versão 1.0.3 seguiu-se para a versão seguinte. A versão pretendida é a versão 1.0.4 do protocolo LoRaWAN, que, como mencionado anteriormente, não tem disponível nenhum *firmware* que implemente esta novidade da comunicação LoRa para o ESP32.

Esta nova versão introduz mais complexidade e maneiras diferentes de interligar a *Stack* possibilitando a introdução de novas funcionalidades e uma maior segurança na comunicação.

Para atualizar a *Stack* utilizou-se um mecanismo idêntico ao usado para comparar a versão 1.0.2 com a versão 1.0.3. No entanto, a passagem da versão 1.0.3 para a versão 1.0.4 foi um pouco mais complexa.

O primeiro foco na atualização para a versão 1.0.4 foi integrar a nova *Stack* e comparar as diferenças de modo a entender o que foi alterado entre as duas versões. Com esta análise, notou-se que havia diversas estruturas que tinham sido ligeiramente alteradas. Estas ligeiras alterações eram, normalmente, adições para suporte das novas funcionalidades. Sendo que, na parte da *Stack*, as principais alterações são para melhorar a segurança.

Seguindo para a atualização do *firmware* de modo a implementar a versão 1.0.4 do *Stack* LoRaWAN, o ambiente anteriormente mencionado consta com duas versões do *firmware*, a versão 1.20.2 e a 1.20.3, porém a versão base utilizada, para a que seria a versão cuja versão da *Stack* correspondente à 1.0.4, é a versão 1.20.3 por se estar mais próxima da versão 1.0.4 do protocolo.

Tendo em conta a versão base, a primeira alteração foi na pasta referente à *Stack* (Figura 3.2) onde se substituiu a *Stack* da versão 1.0.3 pela *Stack* que implementa a versão 1.0.4 e a versão 1.1. Esta versão está disponível no repositório Github da LoRaMac-node pertencente à Lora-net, sendo a versão 4.7.0 do repositório LoRaMac-node e que, até à data, é a versão mais recente.

Com a nova *Stack* colocada no conjunto do *firmware*, optou-se por compilar o respetivo *firmware* e obter uma lista vasta de locais do código que apresentam erros derivados desta atualização. É uma técnica simples mas eficaz para um ponto de partida, a quantidade de erros é muito elevada, porém ao resolver os erros pela listagem que é apresentada ao compilar, por vezes, são resolvidos múltiplos erros com apenas uma alteração. Ao compilar, analisar, resolver e compilar novamente foram resolvidos todos os erros até que ao compilar não fosse apresentado nenhum erro.

A maioria dos erros resultantes da compilação são diferenças nas estruturas que ou mudam de nome ou quando a informação que anteriormente correspondia a uma estrutura agora, com a nova versão da *Stack*, passa a utilizar várias estruturas devido a acréscimo de informação face à versão 1.0.3. Um exemplo claro do que se mencionou é a estrutura LoRaMacNvmCtx_t do *firmware* que implementa a versão 1.0.3 e as estruturas LoRaMacNvmDataGroup1_t e LoRaMacNvmDataGroup2_t da *Stack* com a versão 1.0.4 no repositório da LoRaMac-node. Como era expectável, grande parte do erros estão na parte de integração de LoRaWAN com o microcontrolador. Esta fase aparenta ser simples porém foi verificada, por vezes, alguma dificuldade a relacionar as antigas estruturas com as novas de modo a que todas as funções sejam executadas como nas versões anteriores do *firmware*. Um dos erros mais simples, porém, que demorou algum tempo a ser resolvido foi um erro que mencionava algo de errado com as funções de alguns ficheiros, o erro era descrito como "não coincide". Com alguma pesquisa, percebeu-se que para resolver este erro era necessário o Makefile de modo a incluir os novos ficheiros da *Stack*.

Após o *firmware* compilar sem qualquer tipo de erro foram testadas as funções em MicroPython que permitiam a utilização da tecnologia LoRa. Neste teste verificou-se que no modo OTAA o dispositivo ou não era capaz de enviar o *join* ou a TTN não o apresentava por problemas nas chaves. Com um *gateway* pessoal numa localização que não tem cobertura LoRa, efetuaram-se os mesmos testes mas desta vez observou-se a informação recebida na *gateway* de modo a perceber se o sensor estaria a enviar o *join* e este estava a ser ignorado ou se apenas não estava a enviar. Com este teste concluiu-se que, de facto, o sensor não estava a enviar.

Para resolver este problema, analisou-se o código e o todo processo de *join*, isto implica observar as funções cruciais ao processo e ter um análise crítica em relação ao que se esperava dessas funções e se a expectativa foi cumprida, com o objetivo de encontrar as falhas. Efetuaram-se dezenas de testes, cada vez mais minuciosos em que até as funções mais "pequenas"eram alvo deste processo, e todos pareciam estar a correr como esperado. No fim deste teste desconfiou-se que o problema fosse do rádio LoRa e ao comparar o *drivers* que estavam em utilização no *firmware* com o *drivers* sugeridos no repositório LoRaMac-node percebeu-se que os *drivers* do *firmware* necessitavam de sofrer alterações mas que mantivessem a compatibilidade com o dispositivo. Para isso comparou-se os *drivers* do *firmware* nas versões mais antigas do *firmware,* que implementam a *Stack*, com a versão do repositório LoRaMac-node que implementa a *Stack* na versão correspondente. Deste modo percebeu-se o que era essencial manter e acrescentar nos novos *drivers* do módulo rádio LoRa.

Com a alteração dos *drivers* efetuada, repetiu-se o teste ao *join* e verificou-se que este já era enviado, recebido pela *gateway* e ainda apresentado na TTN, porém a resposta ao *join* nunca era positiva, ou seja, o *join* era sempre rejeitado. Foi analisado, novamente, todo o processo de *join* e foi evidente que algumas configurações na *Stack* não vinham pré-definidas para o ambiente que se pretendia. Por omissão, a *Stack* utilizaria a versão 1.1 quando se pretendia testar a versão 1.0.4 e ainda o devNonce era incrementado quando deveria ser e não aleatório.

Ao alterar estas configurações o *join* passou a obter uma resposta de sucesso, Join-Accept, e prosseguiu-se com os testes, no caso, com tentativas de envio de pacotes com mensagens básicas apenas para testar o envio do dispositivo e receção na TTN. Este teste não teve sucesso, pois as mensagens enviadas não apareciam na aplicação do dispositivo na TTN. Usou-se, novamente, a informação da *gateway* para perceber se o dispositivo estava a enviar os pacotes, e, com esta informação, percebeu-se que os pacotes estavam a ser enviados mas que haveria um problema nas chaves que impedia a TTN de apresentar a informação na aplicação do dispositivo.

Para perceber o que teria acontecido para as chaves não estarem corretas analisou-se o código cuidadosamente para verificar se o MIC, chaves e outros parâmetros estariam corretos. Concluiu-se que em alguns destes parâmetros apresentavam valores fora do expectável, e, optou-se por observar o inicio do pedido de *join* em MicroPython, onde é convertido para C e efetua pedidos para a execução do *join*. Ao observar que as chaves que são configuradas no dispositivo e na TTN estavam corretas observou-se a resposta ao *join* e as chaves que ao dispositivo são passadas, aí reparou-se que havia uma distinção entre GEN_APP_KEY e APP_KEY no ficheiro modlora.c, sendo que na *Stack* da versão 1.0.3 não existia esta distinção. Confirmou-se que na versão 1.0.4 da *Stack* esta distinção era feita e que estaria aí o erro, a GEN_APP_KEY deixou de existir e não acusava nenhum erro porque no modlora.c é efetuado um pedido para as funções da *Stack* guardarem aquela chave, como a *Stack* não implementa aquela chave esta parte é ignorada o que resulta numa falha ao cifrar por faltar uma das chaves necessárias. Retirando a GEN_APP_KEY e ao utilizar apenas a APP_KEY como deveria ser utilizada, o dispositivo já enviava os pacotes corretamente de modo a que a TTN conseguisse decifrar e associar à aplicação correta.

Para finalizar esta fase, foi testado o comando realizado na versão anterior, que o código seria aproveitado para esta versão (Listagem 3.3), o comando DevTimeReq. O objetivo era verificar que na versão 1.0.4, após sofrer bastantes alterações, o comando tem o comportamento adequado, ou seja, idêntico ao comportamento observado no *firmware* que implementa o versão 1.0.3 do protocolo LoRaWAN. O teste foi bem sucedido permitindo prosseguir para a implementação dos restantes comandos.

Sendo que a grande maioria dos novos comandos só pedidos efetuados pelo *Network Server* o código de implementação destes comandos já faz parte da *Stack*, o que é necessário é criar funções que permitam ao utilizador efetuar os comandos que são pedidos pelo dispositivo, tal como o DevTime.

Neste caso, foi criado no modlora.c novas funções (Listagem 3.5) que permitissem efetuar o comando de LinkCheck, um comando que permite ao dispositivo saber quantas *gateways* receberam o seu pacote e qual é a margem de sinal. À semelhança do que foi feito para o DevTime, criaram-se as funções que efetuam o pedido LinkCheckReq e a função que permite ler a informação obtida.

```
STATIC mp_obj_t lora_linkcheck (mp_obj_t self_in) {
    MlmeReq_t mlmeReq;

    mlmeReq.Type = MLME_LINK_CHECK;

    if((LoRaMacMLmeRequest (&mLmeReq)) == LORAMAC_STATUS_OK){
        return mp_const_true;
}
    return mp_const_false;
}

STATIC MP_DEFINE_CONST_FUN_OBJ_2(lora_linkcheck_obj, lora_linkcheck);

mp_obj_t lora_getlinkcheck (mp_obj_t self_in){
    static const qstr lora_compliance_linkCheck_fields[] ={
    MP_QSTR_enabled, MP_QSTR_running
    };
    uint8_t link_check[] = {0,0};
    LoRaMacLinkCheckAns (link_check);
    mp_obj_t compliance_tuple[2];
    compliance_tuple[0] = mp_obj_new_int(link_check[0]);
    compliance_tuple[1] = mp_obj_new_int(link_check[1]);
    return mp_obj_new_attrtuple(lora_compliance_linkcheck_fields, 2,
   compliance_tuple);
}
STATIC MP_DEFINE_CONST_FUN_OBJ_2(lora_getlinkcheck_obj, lora_getlinkcheck);
```

Listagem 3.5: Código de implementação do comando LinkCheck

4

Testes de validação

Para validar o *firmware* com a nova versão do protocolo LoRaWAN, é necessário efetuar testes que confirmem o funcionamento do protocolo de acordo com uma norma. Ao pesquisar possíveis testes para certificar o trabalho realizado, encontraram-se os testes da LoRa Alliance, o principal grupo de entidades que tem como objetivo evoluir o protocolo LoRaWAN [32].

Os testes recomendados são bem orientados e têm descrições claras do que se esperar para concluir cada ponto como válido. Após analisar o documento oficial da LoRa Alliance, elaborou-se o seguinte plano de testes:

- OTAA
- ABP
- *Device Funcionality*
- MAC *Commands*

Para verificar se está tudo a comportar-se como esperado verifica-se o valor do **Join-Nonce**, isto passa por analisar as tramas do **Join-Accept**. Ainda no *Join*, é necessário testar se as configurações passadas na mensagem **Join-Accept** para as janelas RX1 e RX2.

No modo ABP não há *Join*, logo estes testes, anteriormente mencionados, não são realizados. Contudo, há outras vertentes a testar. No caso do ABP, é necessário testar se o

dispositivo ao ser reiniciado mantém todas as configurações que são pré estabelecidas nas comunicações entre dispositivo e *gateway*.

Segue-se para os testes que provam que o dispositivo está funcional, estes começam por testar a criptografia das comunicações. É essencial verificar se a cifra AES, o MIC e o *Downlink Sequence Number* estão a funcionar corretamente. Após esta questão estar garantida, são testadas as mensagens de *uplink* e de *downlink*, ou seja, é verificado se as mensagens estão a ser enviadas/recebidas e se os parâmetros variáveis ao longo da comunicação têm o comportamento esperado.

Por fim, o teste aos comandos MAC. Este teste consiste em verificar um a um cada MAC *Command*, sendo que estes são:

4.1 DevStatus

O *Network Server* (NS) pode enviar um DevStatusReq para o dispositivo para pedir informação do estado do dispositivo. O DevStatusReq não tem *payload*. O dispositivo deve responder com o comando DevStatusAns, sendo que este tem *payload* e este deve consistir no nível da bateria e ainda o estado do rádio. O *payload* da respostas está distribuído com valores diferentes, o nível da bateria está descrito com 256 bits enquanto o estado do rádio tem apenas 8 bits, em que 6 destes 8 são para informar qual é a relação sinal ruído ou *Signal-to-Noise Ratio* (SNR).

Tabela 4.1: Formato da trama DevStatusReq

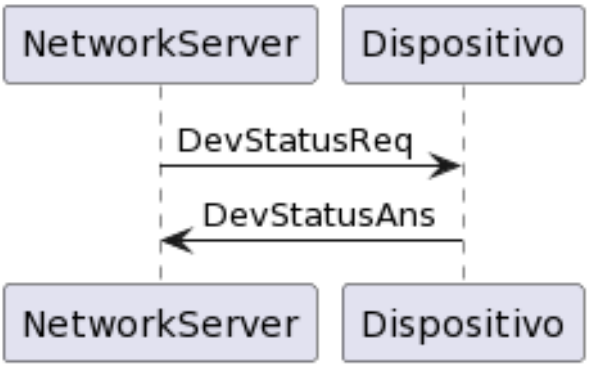
Tamanho [bytes]	1	1
Trama DevStatusReq	Bateria	RadioStatus

Tabela 4.2: Formato do campo RadioStatus

Bits	[7:6]	[5:0]
RadioStatus	RFU	SNR

Para testar esta troca de comandos MAC, o NS envia um DevStatusReq e espera que o dispositivo responda com o DevStatusAns, se isso não acontecer o NS repetirá o comando até cinco vezes, caso o dispositivo não responda entre os cinco *uplinks* seguintes o teste é considerado como falhado.

Tabela 4.3: Exemplo de procedimento dos comandos DevStatusReq e DevStatusAns



4.2 NewChannel

À semelhança do comando anterior, o NS envia o pedido e o dispositivo responde. Neste caso o pedido é o NewChannelReq e pode ter dois efeitos sendo o primeiro a criação de um novo canal bidirecional, e o segundo é a modificação de parâmetros num canal existente. O comando define a frequência central do canal e as opções de *Data Rate* (DR) disponíveis para este canal. O *payload* destes pedido consiste em cinco bytes sendo estes distribuídos por três campos. O primeiro campo é o índice do canal e tem a dimensão de um byte, já o segundo campo corresponde à frequência central e tem a dimensão de três bytes, e por fim, o último campo são as opções de DR que ocupa um byte.

Tabela 4.4: Formato da trama NewChannelReq

Tamanho [bytes]	1	3	1
Trama NewChannelReq	Índice do canal	Frequência	DataRateRange

Tabela 4.5: Formato do campo DataRateRange

Bits	[7:4]	[3:0]
DataRateRange	DataRate Máximo	DataRate Mínimo

O campo correspondente à frequência é um 24-bit *unsigned intenger*, ou seja, um inteiro de vinte e quatro bits. Para a frequência ser representada em Hz é necessário multiplicar o valor do *payload* por cem.

Para definir quais são os DR permitidos, no campo correspondente, é definido qual é o DR máximo em quatro bits e nos outros quatro bits é definido o DR mínimo.

O comando de resposta, tem um *payload* de apenas um byte e usa dois dos oito bits para confirmar que os limites de DR foram estabelecidos e que a frequência do canal também foi definida.

Tabela 4.6: Formato da trama NewChannelAns

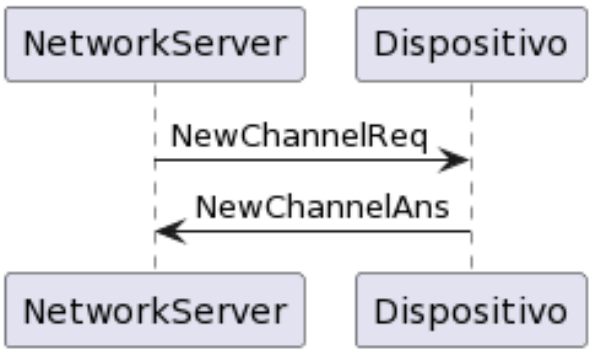
Tamanho [bytes]	1
Trama NewChannelAns	Status

Tabela 4.7: Formato do campo Status da trama NewChannelAns

Bits	[7:2]	1	0
Status	RFU	DataRate ok	Frequência do canal ok

Os testes destes comandos são um pouco mais extensos que o teste para o comando anterior devido às diferentes opções de utilização deste comando. Para testar por completo este comando é necessário testar se é possível saber quais são os canais predefinidos, se é possível adicionar um canal, se é possível remover um canal, se é possível adicionar e remover múltiplos canais e ainda se é resistente erros na frequência ou no DR. Só se é considerado um teste bem sucedido se todos os testes parciais forem validados.

Tabela 4.8: Exemplo de procedimento dos comandos NewChannelReq e NewChannelAns



4.3 DlChannel

O comando DlChannelReq é um comando enviado pelo NS para o dispositivo permitindo associar uma frequência de *downlink* distinta para o RX1. Este comando só

pode ser usado em algumas regiões, caso não seja suportado o dispositivo deve ignorar. O *payload* deste comando tem apenas três bytes sendo um deles para identificação do canal e o outro para indicar a frequência pretendida. Tal como no comando NewChannelReq, o campo da frequência é um inteiro de vinte e quatro bits que para ser representado em Hz é necessário multiplicar o valor passado no campo por cem.

Tabela 4.9: Formato da trama DlChannelReq

Tamanho [bytes]	1	3
Trama DlChannelReq	Índice do canal	Frequência

O dispositivo deve informar sempre a receção do comando DlChannelReq enviando um DlChannelAns quer seja pelo campo FOpts ou pelo FRMPayload dependendo do valor do FPort. Esta medida garante que mesmo que se perca um pacote de *uplink*, o NS sabe sempre a frequência de *downlink* usado pela dispositivo. O *payload* da resposta DlChannelAns tem apenas um byte de dimensão em que dois dos bits servem para confirmar se a frequência de *uplink* existe e se a frequência do canal de *downlink* foi bem definida.

Tabela 4.10: Formato da trama DlChannelAns

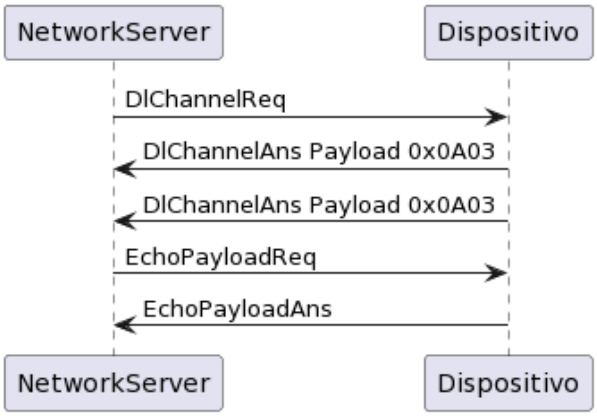
Tamanho [bytes]	1
Trama DlChannelAns	Status

Tabela 4.11: Formato do campo Status da trama DlChannelAns

Bits	[7:2]	1	0
Status	RFU	Frequência de *uplink* existe	Frequência do canal ok

Segundo os testes de conformidade [32], para testar estes comandos é necessário que o NS envie o comando 5 para modificar a frequência de *downlink* na janela RX1 e em seguida o NS testas a janela de *downlink* RX1 com a nova frequência. O NS espera por um *uplink* em que a resposta tem de ter um certo valor, no caso é 0x0A03. É enviado um *downlink* e espera um novo *uplink* cuja resposta tenha um valor diferente do mencionado anteriormente.

Tabela 4.12: Exemplo de procedimento dos comandos DlChannelReq e DlChannelAns



4.4 RXParamSetup

O RXParamSetupReq é um comando MAC enviado pelo NS,este permite alterar a frequência e o DR do RX2 e ainda programar um *offset* entre o *uplink* e o espaço do RX1. O comando tem uma dimensão de quatro bytes sendo que três são para descrever a frequência, tal como acontece nos comandos anteriores que incluem a frequência, e o restante byte serve para as restantes configurações. São usados três bits para definir o *offset* do DR do RX1 e outros três bits para descrever o DR do RX2. O campo RX1DRoffset define o *offset* entre o DR de *uplink* e o DR de *downlink* usado para comunicar com o dispositivo na janela RX1.

Tabela 4.13: Formato da trama RXParamSetupReq

Tamanho [bytes]	1	3
Trama RXParamSetupReq	DLSettings	Frequência

Tabela 4.14: Formato do campo DLSettings

Bits	7	[6:4]	[3:0]
DLSettings	RFU	Rx1DROffset	RX2DataRate

A resposta deve partir do dispositivo através do RXParamSetupAns, este comando deve vir integrado no FOpts ou no FRMPayload consoante o valor de FPort, tal como descrito anteriormente, este método serve para garantir que mesmo que se perca um pacote de *uplink*, o NS sabe sempre a frequência de *downlink* usado pela dispositivo.

Caso o dispositivo seja da classe C, este não deve enviar um *downlink* antes que seja recebido um *uplink* com o comando RXParamSetupAns válido.

Tabela 4.15: Formato da trama RXParamSetupAns

Tamanho [bytes]	1
Trama RXParamSetupAns	Status

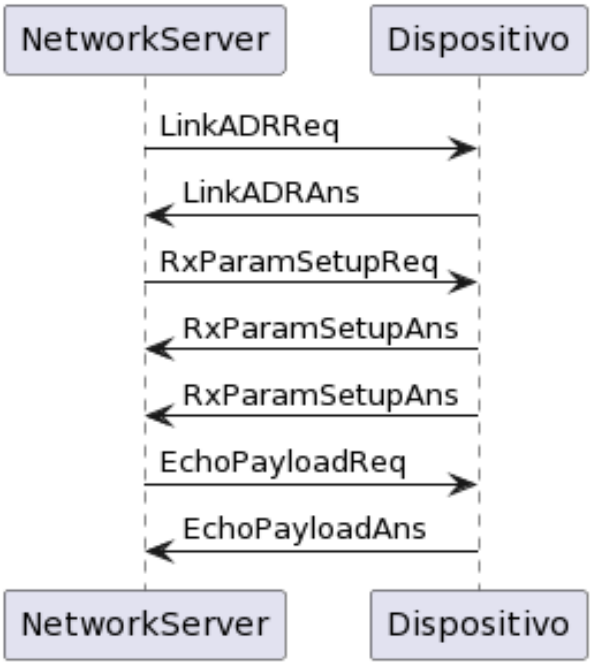
Tabela 4.16: Formato do campo Status

Bits	[7:3]	2	1	0
Status	RFU	RX1DROffsetACK	RX2DataRateACK	ChannelACK

Para efetuar os testes é necessário que o NS envie o comando RXParamSetupReq para configurar o novo RX1DRoffset, o RX2DataRate e ainda a frequência. É exigido ao dispositivo que nos subsequentes *uplinks* envie o comando RXParamSetupAns como respostas até que o NS envie um *downlink*, só a partir do *downlink* é que o dispositivo pode parar de enviar o RXParamSetupAns. O NS esperar receber várias vezes a confirmação que os parâmetros de RX foram bem estabelecidos e só ter a certeza é que este envia um *downlink*. Ainda assim, após o *downlink*, o NS verifica se o próximo *uplink* não contém o comando de resposta RXParamSetupAns. A janela RX1 e a janela RX2 são testados com os novos parâmetros.

O NS envia propositadamente parâmetros inválidos para validar se o dispositivo ignora o comando, como é pretendido.

Tabela 4.17: Exemplo de procedimento dos comandos RXParamSetupReq e RXParam-SetupAns



4.5 RXTimingSetup

Este comando é um comando muito simples e que é enviado pelo NS, o NS envia o RXTimingSetupReq com o finalidade de configurar o atraso entre o fim do processo de transmissão do *uplink* e a abertura da janela de receção RX1. A janela de receção RX2 é aberta ao fim de um segundo da janela RX1, sendo apenas necessário configurar a janela RX1.

O *payload* deste comando tem um byte de dimensão e apenas quatro bits quantificam o atraso necessário. O atraso é descrito em segundos e pode ter valores de um a quinze segundos de atraso.

O dispositivo responde com o comando RXTimingSetupAns, este não tem *payload* e é enviado no campo FOpts ou FRMPayload.

Tabela 4.18: Formato da trama RXTimingSetupReq

Tamanho [bytes]	1
Trama RXTimingSetupReq	RxTimingSettings

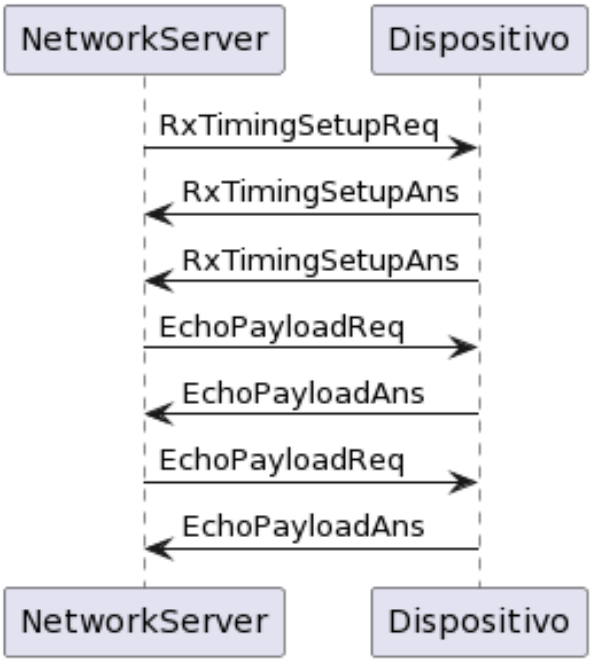
Tabela 4.19: Formato do campo RxTimingSettings

Bits	[7:4]	[3:0]
RxTimingSettings	RFU	Atraso

O teste deste comando é iniciado com o envio do comando RXTimingSetupReq pelo NS. O dispositivo deve responder com o 0x08, porém, sem qualquer *payload*. Após esta confirmação por parte do dispositivo, o NS testa a janela de receção RX1 e RX2 com os novos parâmetros.

Outro teste realizado consiste no envio do RXTimingSetupReq por parte do NS, é recebida a confirmação enviada pelo dispositivo mas o NS não efetua qualquer *downlink* e espera um novo *uplink* para verificar se o dispositivo continua a enviar o comando RXTimingSetupAns. Assim que o NS envie um *downlink* o dispositivo deve deixar de enviar o comando RXTimingSetupAns no *uplink*.

Tabela 4.20: Exemplo de procedimento dos comandos RxTimingSetupReq e RxTimingSetupAns



4.6 TXParamSetup

Este comando é obrigatório ser usado por parte do NS, o NS tem de enviar o comando TXParamSetupReq para notificar o dispositivo qual é o tempo máximo de transmissão

contínua dos pacotes e ainda para informar qual é a potência máxima efetiva que pode ser utilizada. O *payload* tem um byte de dimensão em que um bit descreve o tempo de *downlink*, outro byte descreve o tempo de *uplink* e quatro bits descrevem a potência máxima. Os quatro bits são um código que correspondem a um valor de potência como é descrito na Tabela 4.23. Já o tempo máximo de transmissão é quatrocentos milisegundos se o bit tiver o valor um ou então sem limite de transmissão se o valor do bit for zero.

Tabela 4.21: Formato da trama TXParamSetupReq

Tamanho [bytes]	1
Trama TXParamSetupReq	EIRP_DwellTime

Tabela 4.22: Formato do campo EIRP_DwellTime

Bits	[7:6]	5	4	[3:0]
RxTimingSettings	RFU	DownlinkDwellTime	UplinkDwellTime	Potência Máxima

Tabela 4.23: Codificação da Potência máxima (Adaptado de [32])

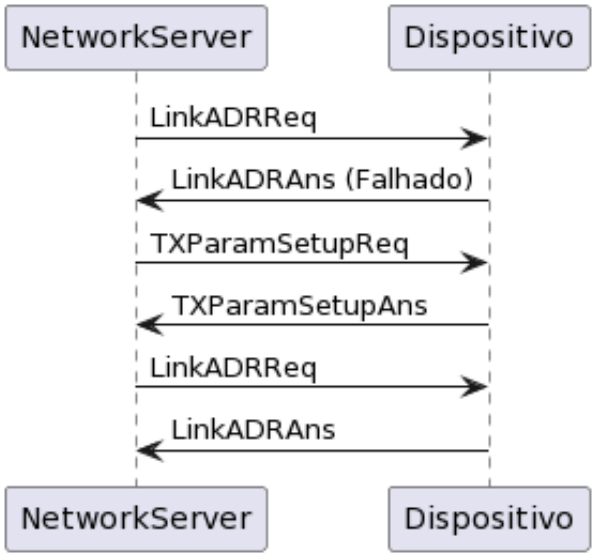
Código	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Potência Máxima (dBm)	8	10	12	13	14	16	18	20	21	24	26	27	29	30	33	36

Como verificado em comandos anteriores, o TXParamSetupAns é enviado no FOpts ou no FRMPayload. Se o comando TXParamSetupReq for utilizado numa região que não sejam necessários estes parâmetros o dispositivo não deve processar o comando MAC e não transmitir nenhuma mensagem de confirmação.

Os testes para este comando têm uma sequência um pouco distinta dos testes que foram descritos até então, a sequência de testes implica outros comandos MAC para concluir o estado desta funcionalidade. O primeiro teste consiste em não efetuar nenhuma alteração nos TXParam e efetuar o comando LinkADRReq do NS para o dispositivo para configurar utilizar os DRs que não estavam definidos no Join-Request, é expectável que o LinkADRAns seja enviado com uma mensagem de não sucesso. Em seguida, definir o tempo de transmissão para ilimitado no comando TXParamSetupReq e espera a resposta TXParamSetupAns para poder repetir o processo do primeiro teste, enviar um comando LinkADRReq e desta vez é expectável uma resposta de sucesso por parte do dispositivo. O NS verifica que o DR do *uplink* corresponde ao que foi pedido.

Para testar a configuração da potência máxima é forçado que este valor tenha o seu valor máximo possível e em seguido o valor mínimo possível e verificar se é registada esta diferença.

Tabela 4.24: Exemplo de procedimento dos comandos TxParamSetupReq e TxParamSetupAns



4.7 LinkCheck

Este comando é um dos poucos que é enviado pelo dispositivo, ainda que possa ser por iniciativa própria ou por exigência do NS. O dispositivo envia o comando LinkCheckReq sem qualquer *payload* para informar o NS que pretende saber quantos *gateways* estão a receber as mensagens e qual é margem do sinal em dB. Se o dispositivo for de classe A a resposta LinkCheckAns apenas chegará quando houver um *uplink* por parte do dispositivo. O comando de resposta LinkCheckAns tem um *payload* de dois bytes em que o primeiro byte é a margem do sinal e o segundo byte indica a quantidade de *gateways* (Tabela 4.25).

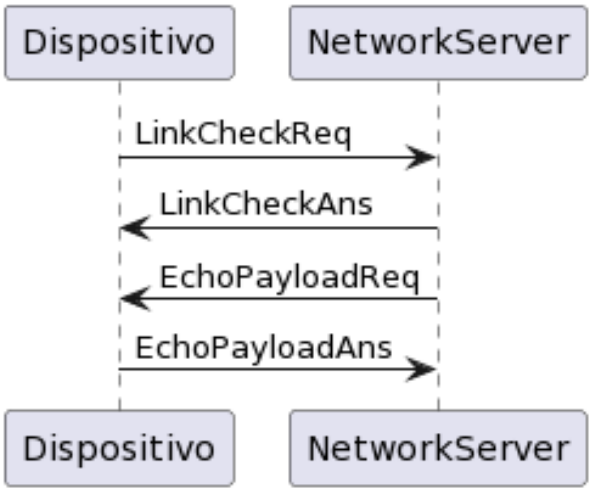
Tabela 4.25: Formato da trama LinkCheckAns

Tamanho [bytes]	1	1
Trama LinkCheckAns	Margem	Contagem de gateways

Para verificar o funcionamento destes comandos é necessário que dispositivo envie o

comando LinkCheckReq e que seja recebido o LinkCheckAns com a informação correta. Ainda que a informação seja correta, é necessário verificar se o dispositivo continua o seu funcionamento normal.

Tabela 4.26: Exemplo de procedimento dos comandos LinkCheckReq e LinkCheckAns



4.8 LinkADR

O comando LinkADRReq é enviado pelo NS para pedir ao dispositivo que faça uma adaptação de DR e de potência de transmissão. O *payload* deste comando MAC tem uma dimensão de quatro bytes, um byte de redundância, dois bytes de máscara e um byte para a restante informação (Tabela 4.27, Tabela 4.28 e Tabela 4.29). A restante informação está dividida entre o DR e a potência de transmissão, tendo quatro bits para cada um destes parâmetros. A potência de transmissão está codificada do mesmo modo da potência do comando TXParamSetup, como se pode consultar na Tabela 4.23. O dispositivo deve responder ao pedido e se a potência pedida for superior ao que o dispositivo consegue transmitir, este deve transmitir na máxima potência que conseguir. Se a potência pedida for inferior ao que o dispositivo consegue, este deve ignorar e continuar a usar a potência que estava a utilizar.

O NS deve incluir múltiplos comandos LinkADRReq num único *downlink* e o dispositivo deve processar todos os comandos LinkADRReq. O LinkADRAns é o comando de resposta e tem apenas um byte de *payload* que consiste na confirmação dos diferentes parâmetros.

Tabela 4.27: Formato da trama LinkADRReq

Tamanho [bytes]	1	2	1
Trama LinkADRReq	DataRate_TxPower	Máscara do canal	Redundância

Tabela 4.28: Formato do campo DataRate_TxPower

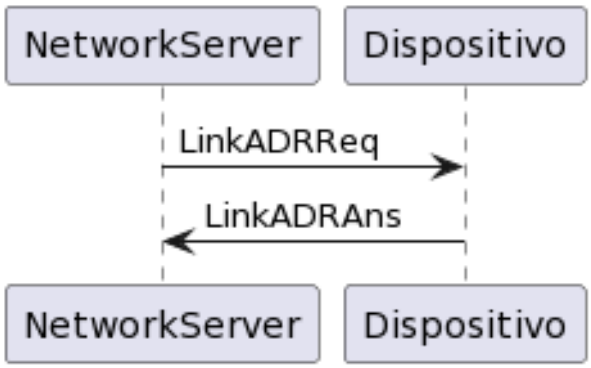
Bits	[7:4]	[3:0]
DataRateTxPower	DataRate	Potência de transmissão

Tabela 4.29: Formato do campo Redundância

Bits	7	[6:4]	[3:0]
Redundância	RFU	Controlo da máscara do canal	Nº Transmissões para cada *uplink*

Para aprovar o funcionamento deste comando, são feitos vários testes parciais para averiguar o funcionamento de diversas aplicações deste comando. Um dos testes serve para validar se o dispositivo se comporta corretamente consoante os diversos valores possíveis para a potência de transmissão. Em seguida, também são testados os canais e os DRs associados aos mesmos. É necessário verificar se varia o DR e utilização deste comando com outros comandos, como mencionado anteriormente. Por fim, é testada a redundância. É verificado se está a ocorrer como é expectável e se não há erros na retransmissão.

Tabela 4.30: Exemplo de procedimento dos comandos LinkADRReq e LinkADRAns



4.9 DutyCycle

O comando DutyCycleReq pode ser usado pelo NS para limitar o *duty cycle* para todos as sub-bandas de transmissão. O *payload* deste comando tem apenas um byte em que

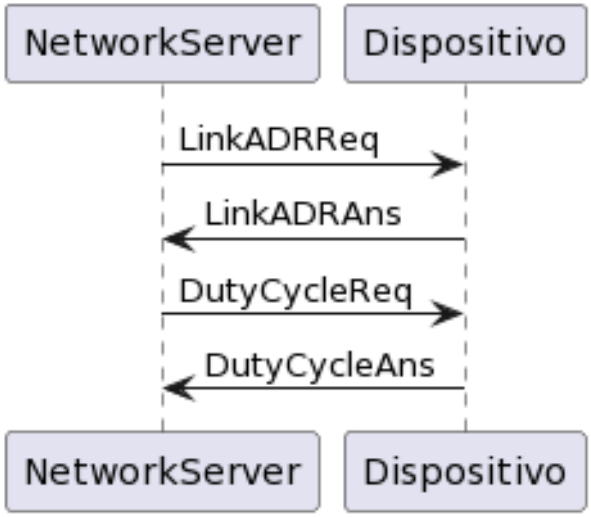
apenas quatro bits são usados para descrever o *duty cycle* máximo (Tabela 4.31). O dispositivo responde com um comando que não tem *payload*, o comando DutyCycleAns, servindo apenas para confirmar.

Tabela 4.31: Formato do campo DutyCyclePL

Bits	[7:4]	[3:0]
DutyCyclePL	RFU	DutyCycle máximo

O teste usado para validar esta funcionalidade começa por validar se o dispositivo atualiza corretamente o *duty cycle*. O NS deve definir o DR máximo para o dispositivo e esperar pelos *uplinks* enviados pelo dispositivo ao que o NS deve guardar o tempo em que recebeu os *uplinks* de dois pacotes consecutivos. Nesta fase o NS envia o comando DutyCycleReq para ajustar o *duty cycle* e volta a esperar pelos *uplinks* do dispositivo e guarda o tempo de dois pacotes consecutivos. Assim, é verificado se o comando foi executado com sucesso.

Tabela 4.32: Exemplo de procedimento dos comandos DutyCycleReq e DutyCycleAns



4.10 DeviceTime

Este comando vem da versão 1.0.3 do protocolo LoRaWAN e é enviado a partir do dispositivo, o dispositivo o comando DeviceTimeReq para pedir à rede a data e hora para poder utilizar o relógio sem necessitar de um módulo extra para este fim. Este pedido que é enviado não tem qualquer *payload*. Já o comando de resposta, o comando DeviceTimeAns, tem um *payload* com cinco bytes sendo que quatro desses bytes é para

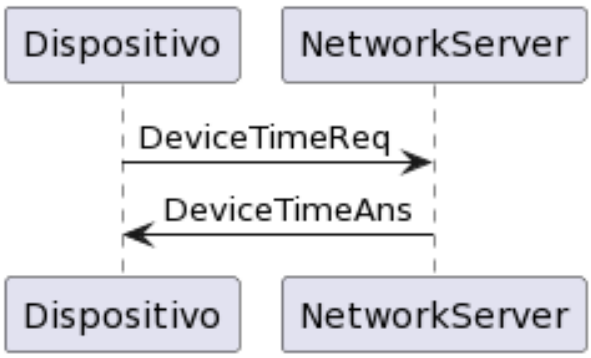
indicar os segundos que passaram desde o *epoch* e o restante byte é a parte fracional. O tempo tem um precisão de cem milisegundos.

Tabela 4.33: Formato da trama DeviceTimeAns

Tamanho [bytes]	4	1
Trama DeviceTimeAns	Segundos desde o epoch	Fração de segundos

O teste é muito semelhante ao teste efetuado para o LinkCheck, o dispositivo envia o comando DeviceTimeReq e tem de enviar um *uplink* para poder receber o DeviceTimeAns. Quando recebe o DeviceTimeAns é necessário verificar se a informação está correta e se o dispositivo continua o seu funcionamento normal.

Tabela 4.34: Exemplo de procedimento dos comandos DevTimeReq e DevTimeAns



4.11 Resumo dos Resultados

Por fim, foram realizados os testes de conformidade para a versão 1.0.4 do protocolo LoRaWAN e obteve-se resultados positivos de modo a comprovar o funcionamento do novo *firmware* com a *Stack* atualizada (Figura 4.35).

Tabela 4.35: Resultado dos testes

Teste	**Estado**
Activation by Personalization	✓
Over the Air Activation	✓
Link Check Commands	✓
Link ADR Commands	✓
End-Device Transmit Duty Cycle	✓
Receive Windows Parameters	✓
End-Device Status	✓
Creation / Modification of a Channel	✓
Setting Delay between TX and RX	✓
End-Device Transmit Parameters	✓
End-Device Time Commands	✓

5

Conclusões e trabalho futuro

O principal objetivo desta dissertação é a implementação das novas versões do *Stack* LoRaWAN para dispositivos cujo microcontrolador é um ESP32. Estas versões mencionadas são a versão 1.0.4 e a versão 1.1.

Para iniciar o projeto, procurou-se compreender o funcionamento do *firmware* já existente para dispositivos da Pycom (cujo microcontrolador é um ESP32 e implementa o protocolo LoRaWAN). O método seguido envolveu a comparação de diferentes versões do *firmware* de modo a perceber as alterações necessárias, tanto a nível de *Stack* quanto na interligação entre a *Stack* e o microcontrolador. Para compreender esta interligação, foram introduzidas funções que não estavam implementadas nas versões disponibilizadas pela Pycom.

Com este conhecimento adquirido, iniciou-se a fase de atualização efetiva da *Stack*. Foi introduzida uma versão da *Stack* que implementa tanto a versão 1.0.4 como a versão 1.1. Devido à complexidade da versão 1.1, optou-se por começar por implementar as funções da versão 1.0.4. Após uma análise aprofundada do *firmware* e efetuadas as alterações necessárias, tornou-se possível utilizar as novas funcionalidades da versão 1.0.4 do protocolo.

Quando a implementação do *Stack* foi considerada como concluída, prosseguiu-se para os testes que iriam confirmar o funcionamento do dispositivos com as novas funções. Os testes foram realizados com base num documento oficial da LoRa Alliance cujo o objetivo do documento é descrever os testes mínimos para que um dispositivo possa ser certificado. De todos os testes sugeridos apenas efetuaram-se os testes aos métodos

de ativação e aos comandos MAC.

Os testes da versão 1.0.4 decorreram como esperado, ou seja, os resultados positivos.

Com o objetivo de respeitar o prazo de entrega da dissertação, optou-se por não implementar a versão 1.1 do protocolo para este microcontrolador, ainda que a *Stack* presente no *firmware* final assim o permita. Tendo isto em conta, para trabalho futuro, fica a implementação das funções da versão 1.1 para este microcontrolador e os respetivos testes.

Referências

[1] Semtech, *Semtech LoRa technology overview*. URL: https://www.semtech.com/lora (acedido em 21/03/2023).

[2] Season Group, *Season group's acquisition of pycom ltd-a global IoT technology startup with a vision to give all connected ideas an opportunity to succeed*. URL: https://www.prnewswire.com/news-releases/season-groups-acquisition-of-pycom-ltd-a-global-iot-technology-startup-with-a-vision-to-give-all-connected-ideas-an-opportunity-to-succeed-301701254.html.

[3] André Filipe Braz Rabaça, "Aplicação de Tecnologia LoRaWAN à Monitorização de Redes de Distribuição de Energia", jun. de 2019.

[4] *Rede de Sensores Sem Fio*. URL: https://www.gta.ufrj.br/seminarios/semin2002_1/flavio/#4.%20Classifica%C3%A7%C3%A3o%20e%20Arquiteturas (acedido em 13/03/2023).

[5] *LoRa and LoRaWAN: The technologies, ecosystems, use cases and market*, i-SCOOP. URL: https://www.i-scoop.eu/internet-of-things-iot/lpwan/iot-network-lora-lorawan/.

[6] *Chirp spread spectrum*, em *Wikipedia*, 19 de set. de 2022. URL: https://en.wikipedia.org/w/index.php?title=Chirp_spread_spectrum&oldid=1111168230.

[7] *LoRa - Redes de Computadores I*. URL: https://www.gta.ufrj.br/ensino/eel878/redes1-2019-1/vf/lora/modulacao.html.

[8] Qoitech, *How spreading factor affects LoRaWAN® device battery life*, The Things Network. URL: https://www.thethingsnetwork.org/article/how-spreading-factor-affects-lorawan-device-battery-life.

[9] "Spreading factor, bandwidth, coding rate and bit rate in LoRa (english)", Josef Matondang. (14 de ago. de 2018), URL: https://josefmtd.com/2018/08/14/spreading-factor-bandwidth-coding-rate-and-bit-rate-in-lora-english/.

[10] Semtech, *Understanding LoRa Adaptive Data Rate*, dez. de 2019. URL: https://lora-developers.semtech.com/uploads/documents/files/Understanding_LoRa_Adaptive_Data_Rate_Downloadable.pdf.

[11] LoRa Alliance. "Regional Parameters RP002-1.0.2". (8 de out. de 2020), URL: https://lora-alliance.org/wp-content/uploads/2020/11/RP_2-1.0.2.pdf.

[12] LoRa Alliance, *Security*, The Things Network. URL: https://www.thethingsnetwork.org/docs/lorawan/security/.

[13] *What are LoRa and LoRaWAN?*, The Things Network, Section: lorawan. URL: https://www.thethingsnetwork.org/docs/lorawan/what-is-lorawan/.

[14] *AWS IoT Core for LoRaWAN workshop*. URL: https://catalog.us-east-1.prod.workshops.aws/workshops/b95a6659-bd4f-4567-8307-bddb43a608c4/en-US/100-intro/lorawanversions.

[15] N. Sornin, M. Luis, T. Eirich, T. Kramp & O. Hersent, *LoRaWAN® Specification v1.0.2*, jul. de 2016. URL: https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-0-2.

[16] The Things Network, *Regional parameters*. URL: https://www.thethingsnetwork.org/docs/lorawan/regional-parameters/.

[17] *Device classes*, The Things Network. URL: https://www.thethingsnetwork.org/docs/lorawan/classes/.

[18] Slim Loukil, Lamia Chaari Fourati, Anand Nayyar & K.-W.-A. Chee, "Analysis of LoRaWAN 1.0 and 1.1 protocols security mechanisms", *Sensors*, vol. 22, n.º 10, pág. 3717, 13 de mai. de 2022, ISSN: 1424-8220. DOI: 10.3390/s22103717. URL: https://www.mdpi.com/1424-8220/22/10/3717.

[19] *End device activation*, The Things Network, Section: lorawan. URL: https://www.thethingsnetwork.org/docs/lorawan/end-device-activation/.

[20] *Glossary The Thing Stack*. URL: https://www.thethingsindustries.com/docs/reference/glossary/.

[21] N. Sornin & A Yegin, *LoRaWAN Specification v1.1*, 11 de out. de 2017. URL: https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-1.

[22] Xingda Chen, Margaret Lech & Liuping Wang, "A Complete Key Management Scheme for LoRaWAN v1.1", *Sensors*, vol. 21, n.º 9, pág. 2962, 23 de abr. de 2021, ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/21/9/2962.

[23] *Introducing LoRaWAN 1.1 support | Mbed*. URL: https://os.mbed.com/blog/entry/Introducing-LoRaWAN-11-support/.

[24] *Mbed OS 5.8 release: Focus on production, robustness and connectivity | Mbed*. URL: https://os.mbed.com/blog/entry/Mbed-OS-58-release/.

[25] *STMicroelectronics*, em *Wikipedia*, Page Version ID: 1141359760. URL: https://en.wikipedia.org/w/index.php?title=STMicroelectronics&oldid=1141359760.

[26] *STMicroelectronics: Our technology starts with you*. URL: https://www.st.com/content/st_com/en.html.

[27] *LoRaWAN products - STMicroelectronics*. URL: https://www.st.com/en/wireless-connectivity/lorawan-products.html.

[28] *Arduino-LMIC library ("MCCI LoRaWAN LMIC Library")*, 9 de set. de 2016. URL: https://github.com/mcci-catena/arduino-lmic.

[29] *BasicMAC*, 31 de out. de 2019. URL: https://github.com/LacunaSpace/basicmac.

[30] *The MicroPython project*, original-date: 2017-03-24T14:26:06Z, 23 de jul. de 2023. URL: https://github.com/pycom/pycom-micropython-sigfox (acedido em 18/09/2023).

[31] Semtech, *Repositório LoRaMac-node*, GitHub. URL: https://github.com/Lora-net/LoRaMac-node/blob/v4.7.0/src/mac/LoRaMac.c (acedido em 18/12/2023).

[32] LoRa Alliance, *LoRaWAN 1.0.4 End Device Certification*, 2020.