# DeSem

DeSenet laboratory report

Rémy Macherel

January 14, 2022

**Authors :**
Macherel Rémy
Sterren Thomas

**Teacher :**
Rieder Medard

# Table des matières

# 1. Introduction

Dans le cadre du cours MA-DeSem, il nous a été demandé de réaliser le protocole DeseNET sur un système de type STM32 Nucleo. Les spécifications du protocole de communication ont été fournies pour ce travail.
La structure de base du projet fût fournie et nous avons du apporter les modifications nécessaires au bon fonctionnement du protocole.

# 2. Modifications du code

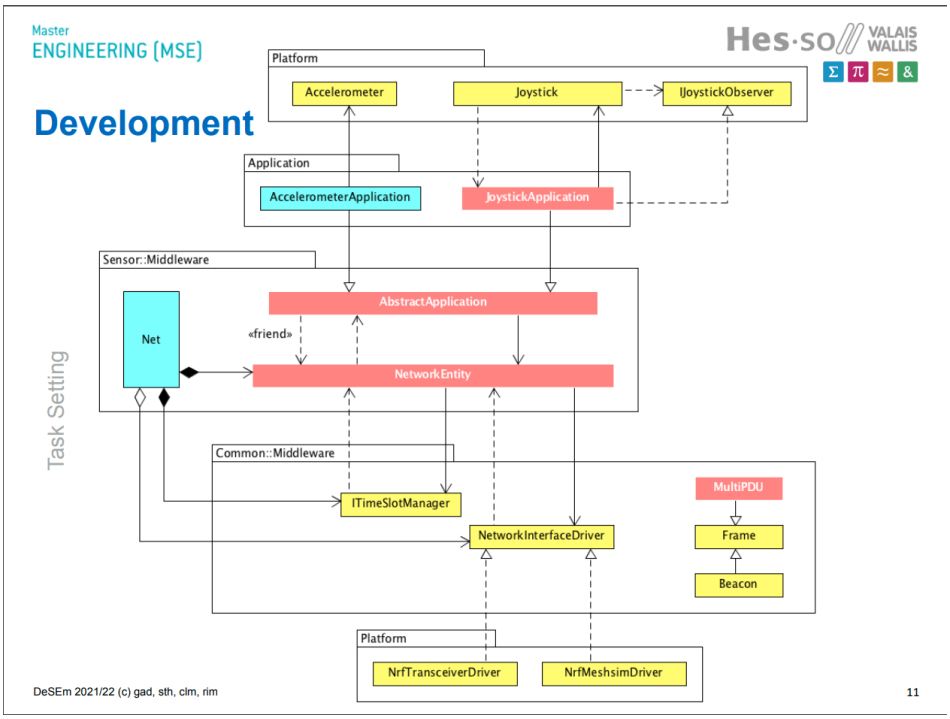


Figure 2.1: Aperçu des modifications à apporter

La figure suivante nous a été présentée afin d'illustrer les classes que nous devions compléter/créer. Les classes à créer sont les suivantes :

1. JoystickApplication (.h et .cpp)
2. MPDU (.h et .cpp)

Alors que les classes à compléter sont :

1. AbstractApplication (.h et .cpp)
2. NetworkEntity (.h et .cpp)

D'autres fichiers ont cependant également été modifiés comme par exemple Factory.cpp afin d'y ajouter l'initialisation du Joystick ainsi que le setObserver().

# 3. Classes implémentées

## 3.1 MPDU

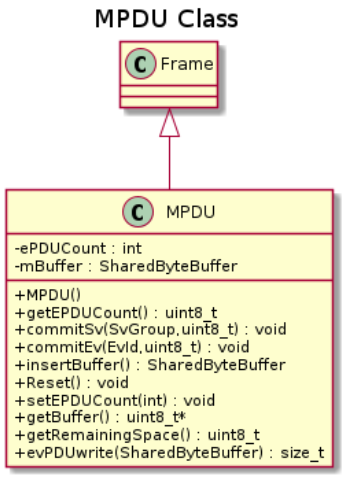


Figure 3.1: Diagramme de la classe MPDU

Comme on peut le voir sur cette figure, la classe MPDU hérite de la classe *Frame*. Ceci va permettre de réutiliser de nombreuses fonctions déjà implémentées dans celle-ci.

### 3.1.1 Méthodes implémentées

#### 3.1.1.1 MPDU()

Cette méthode est le constructeur, elle permet d'initialiser également la classe parent *Frame* en lui passant en paramètre la variable *Mtu* (égale à 37) qui représente la taille maximale d'une Frame.

#### 3.1.1.2 getEPDUCount()

Cette méthode permet d'obtenir le nombre de EPDU qui sont actuellement dans la Frame. La valeur de ce compte est également écrite dans la Frame en question.

#### 3.1.1.3 commitSv(SvGroup,uint8_t)

Cette méthode permet de finaliser l'écriture d'une *SampleValue (Sv)* en ajoutant dans la Frame le header de celle-ci. Le header est composé de trois champs :

- count : la taille en bytes de la valeur
- svGroupOrEvId : Le numéro du groupe auxquel l'application s'est inscrite ou l'Id de l'évènement (pour les valeurs event, voir plus tard)
- type : 0 pour sampleValue et 1 pour event

Les paramètres de cette fonction sont l'identifiant de groupe (type SvGroup) et le byteCount (type uint8_t).

#### 3.1.1.4 commitEv(EvId,uint8_t)

Comme la précédente méthode, celle-ci se charge de finaliser l'écriture d'un event dans la trame. Son fonctionnement est identique hormis qu'elle inscrira dans le champ type du header la valeur 1. Les paramètres sont l'identifiant de l'évènement ainsi que le byteCount.

#### 3.1.1.5 insertBuffer()

Retourne un proxy du buffer principal (issu de Frame) qui permet d'inscrire des valeurs dans le buffer du MPDU. Le début du buffer renvoyé par cette fonction varie en fonction du contenu du buffer du MPDU.

#### 3.1.1.6 reset()

Cette méthode sert à réinitialiser le MPDU afin qu'il soit à nouveau prêt à être réécrit pour de nouvelles données.

#### 3.1.1.7 setEPDUCount()

Cette méthode sert à écrire au bon endroit dans le buffer le compte actuel de EPDU contenus dans le buffer.

#### 3.1.1.8 getBuffer()

Cette méthode fut utilisée dans networkentity au moment de transmettre le MPDU. Elle ne fait qu'appeler la méthode buffer() de la classe Frame mais celle-ci étant déclarée protected, elle ne peut être appelée depuis l'extérieur de la classe MPDU. La méthode getBuffer() sert donc à contourner ceci.

#### 3.1.1.9 getRemainingSpace()

Cette méthode est utilisée lors de l'écriture des EvPDU dans le networkentity afin de connaître l'espace disponible restant dans le buffer du MPDU.

#### 3.1.1.10 evPDUWrite(SharedByteBuffer

Utilisée afin d'écrire les données d'un evPDU (ici une position du Joystick) dans le buffer du MPDU. Les evPDU étant basés sur un évènement et donc pas inscrits dans le publishArray, j'ai créé cette méthode afin d'inscrire la valeur de l'event dans le buffer.

### 3.1.2 Séquence générique d'écriture d'un MPDU

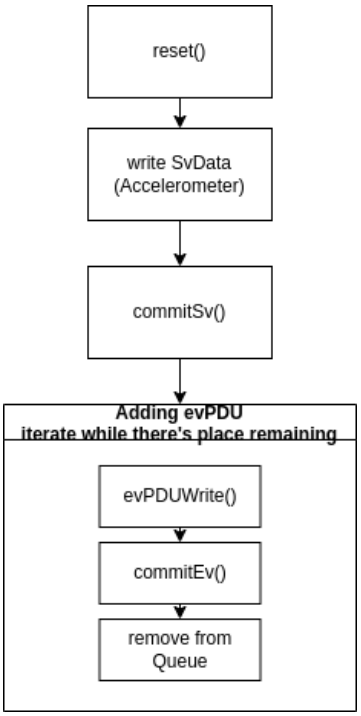


Figure 3.2: Séquence d'écriture du MPDU

Ci dessus sont les étapes suivies lors de l'écriture d'un MPDU, ces étapes utilisent les méthodes précédemment décrites de la classe MPDU et sont réalisées dans la méthode *onReceive()* de la classe *NetworkEntity*.

### 3.1.3 svPDU

Un svPDU représente une *Sample Value*, c'est-à-dire une valeur mesurée par un capteur (dans notre cas l'accéléromètre). Afin d'écrire la valeur dans le MPDU (étape 2 de la séquence de la figure 3.2), on vérifie que le beacon réceptionné demande bien les valeurs du groupe dans lequel notre mesure est effectuée. Si le beacon demande cette mesure, on va appeler la méthode *svPublishIndication* de l'application contenue dans le groupe souhaité (ici l'accéléromètre) en lui passant en paramètres un proxy du buffer du MPDU dans lequel cette méthode se chargera d'inscrire ses valeurs (exemple pour notre cas : la méthode *svPublishIndication* dans le fichier *AccelerometerApplication.cpp*). La méthode *commitSv()* permettra ensuite de finaliser l'écriture de cette sample value.

### 3.1.4 evPDU

Le deuxième type de valeurs reçues dans ce projet sont les valeurs reçues par évènement. Ces valeurs sont dans notre cas des positions de click sur le Joystick. Pour l'écriture de ces valeurs, le protocole prévoit d'ajouter autant de evPDU qu'il reste de place dans le buffer du MPDU. Lorsque soit la queue d'évènements est vide ou que le MPDU n'a plus de place, le MPDU est prêt à être transmit. Note : S'il reste des évènements dans la queue, celle-ci sera vidée lorsque le MPDU est plein, ceci peut engendrer

la perte de certains évènements mais permet également de ne pas "polluer" les prochains timeSlots et ainsi retarder la réception de certains évènements.

### 3.1.5 Transmission du MPDU

Le projet est défini sur un système de Time Slots, c'est-à-dire que chaque station a sa période durant laquelle elle peut transmettre des données au Gateway. Afin de transmettre le MPDU, on utilisera la méthode *onTimeSlotSignal()* de *NetworkEntity*. Dans cette méthode on vérifie que le signal du timeslot correspond bien à un start de notre timeSlot (OWN_SLOT_START) puis on appelle transceiver().transmit(responseMPDU.getBuffer(),responseMPDU.length()) qui va transmettre au Gateway le MPDU construit selon les étapes ci-dessus.

## 3.2 JoystickApplication

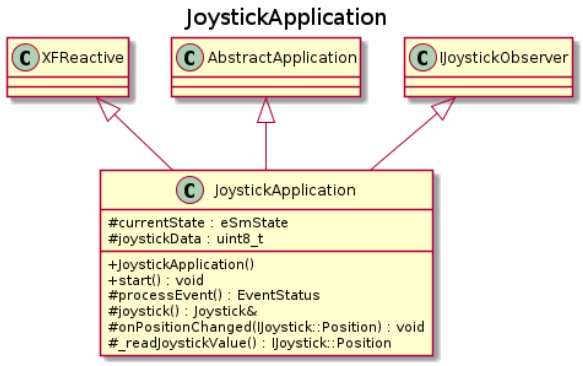


Figure 3.3: JoystickApplication UML Diagram

Cette classe fut construite en se basant sur la classe *AccelerometerApplication*, et hérite de AbstractApplication (classe abstraite de base pour les applications), de IJoystickObserver (afin de pouvoir utiliser la méthode *onPositionChanged()*), et de XFReactive (utilisé pour la machine d'état).

### 3.2.1 Propriétés

Les propriétés de cette classe sont *currentState* qui contient l'étape actuelle de la machine d'état de l'application et *joystickData* qui contient la valeur de la position du joystick.

### 3.2.2 Méthodes implémentées

#### 3.2.2.1 JoystickApplication()

Le constructeur de cette classe permet d'initialiser ses propriétés. Il permet également de définir l'état initial de la machine.

#### 3.2.2.2 start()

Cette méthode permet d'appeler la méthode *startBehavior()* héritée de XFReactive qui permet de démarrer la machine d'état via XFReactive.

#### 3.2.2.3 processEvent()

Cette méthode est utilisée afin d'implémenter la machine d'état (qui sera décrite plus bas) et de traîter les évènements.

#### 3.2.2.4 joystick()

Cette méthode sert à récupérer le singleton correspondant au Joystick() depuis la Factory.

#### 3.2.2.5 onPositionChanged(IJoystick::Position)

Cette méthode héritée IJoystick permet de générer un évènement XFReactive de type *POSITION_CHANGED_EVENT* (voir explications de la machine d'état).

#### 3.2.2.6 _readJoystickValue()

Cette méthode utilise l'instance de Joystick afin de récupérer sa position actuelle.

### 3.2.3 Machine d'état

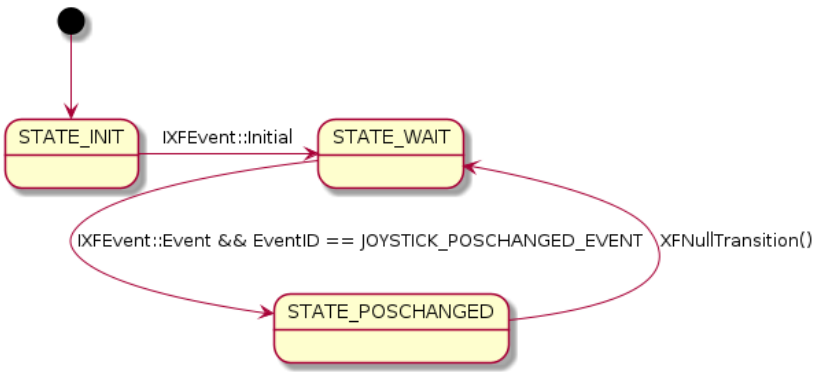


Figure 3.4: JoystickApplication machine d'état

La machine d'état implémentée pour la classe *JoystickApplication* est celle présentée dans la figure 3.4, lors de la réception d'un évènement de type *IXFEvent::Initial*, on passe dans l'état wait, puis, à la réception d'un évènement ayant comme Id JOYSTICK_POSCHANGED_EVENT, on passe dans l'état STATE_POSCHANGED. La machine d'état a été implémentée selon le principe du double switch vu en cours.

# 4. Diagramme de séquence de l'application

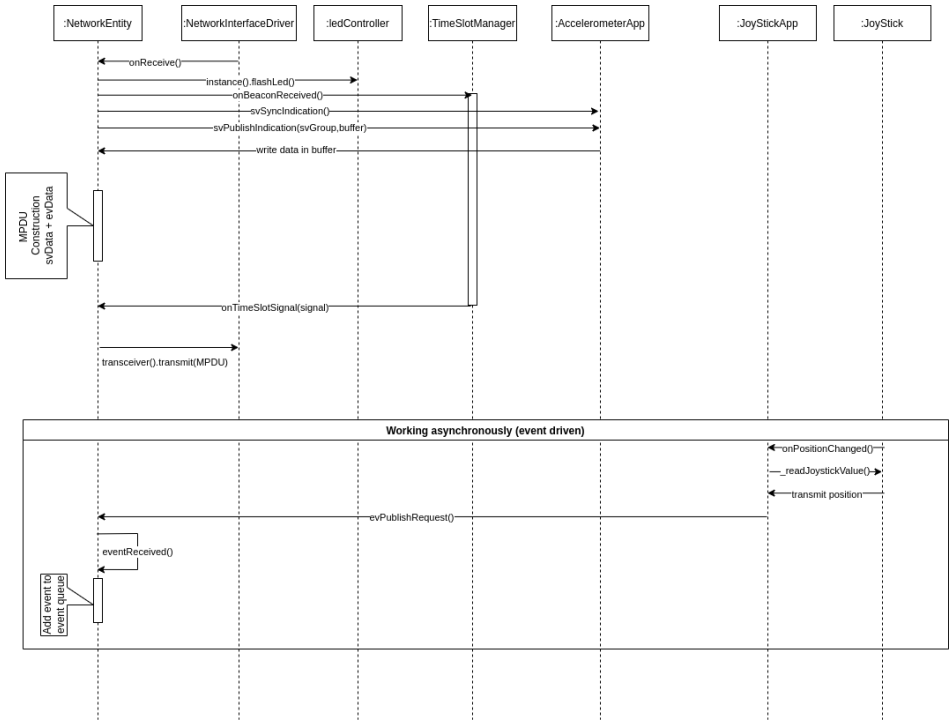


Figure 4.1: Diagramme de séquence de l'application

La figure 4.1 illustre la séquence appliquée lors de la réception d'un beacon. On peut voir qu'une partie est exécutée de manière asynchrone et n'est pas soumise à un enregistrement pour diverses indications. Il en est ainsi car les valeurs du Joystick sont gérées de manière évènementielle et comme on peut le voir sur le diagramme, lorsque la position du joystick change, la joystickApplication lit la nouvelle valeur puis transmet directement au NetworkEntity qui va stocker l'information dans une queue. C'est seulement au moment ou le NetworkEntity aura terminée de recevoir la sample value qu'il va placer dans le MPDU autant de données d'évènements qu'il y a de place restante dans celui-ci. La partie supérieure du diagramme illustre la synchronisation et la collecte de sample value à chaque réception d'un beacon.