Profil Robotique FISE 2021

Rapport de Projet de Fin d'Études – PFE

Cartographie Métrique et Sémantique

Auteur : Alexandre EVAIN

Encadrant École : Luc JAULIN

Encadrant Entreprise : Jacques CHODOROWSKI

Version provisoire du 24/08/2021

Résumé

Ce rapport décrit les travaux effectués lors de mon stage chez l'équipe MINDS d'Orange Labs Lannion du 29/03/2021 au 28/09/2021. L'objectif du stage est la mise en place de deux services sur le robot Geko, un d'inventaire, l'autre de vérification de présence d'objets à une position données; ainsi que la mise en lien des données du robot avec le réseau sémantique Thing'In développé par Orange. Pour cela, nous avons utilisé l'algorithme de reconnaissance d'objets YOLO (You Look Only Once) via son implémentation ROS (Robot Operating System), darknet_ros sur la plateforme NVIDIA Jetson. Le rapport décrit ensuite la mise en place des services, leur fonctionnement théorique; l'intégration de la Jetson sur le Geko ainsi que les résultats des expérimentations effectuées pour tester les services.

Mots-clés: Reconnaissance d'objets, darknet_ros, réseau sémantique, cartographie sémantique, You Look Only Once

Abstract

This report describes the work done during my internship at the MINDS team of Orange Labs Lannion from 29/03/2021 to 28/09/2021. The objective of the internship is the implementation of two services on the Geko robot, one of object inventory, the other of verification of the presence of objects at a given position; as well as linking the robot's data with the semantic network Thing'In developed by Orange. We used the YOLO (You Look Only Once) object recognition algorithm via its ROS (Robot Operating System) implementation, darknet_ros on the NVIDIA Jetson platform. The report then describes the implementation of the services, their theoretical operation; the integration of the Jetson on the Geko as well as the results of the experiments carried out to test the services.

Keywords: Object recognition, darknet_ros, You Look Only Once, semantic network, semantic mapping

Table des matières

Introduction

Présentation de l'entreprise

Orange, anciennement France Télécom, est une société multinationale de télécommunications française, présente dans 26 pays dont 8 en Europe. Il s'agit du huitième opérateur de réseau mobile dans le monde et du quatrième en Europe après Vodafone, Telefónica et VEON. Elle compte 259 millions de clients dans le monde et emploie 89.000 personnes en France, et 53.000 à l'étranger.

Bien qu'il s'agisse historiquement d'une entreprise des télécommunications, Orange s'est diversifiée pour devenir une entreprise du numérique, et propose aujourd'hui à ses clients des services financiers électroniques (Orange Bank & Orange Money), des services numériques pour les entreprises (Orange Business Services) et les clients (Orange Content).

Du fait de son secteur d'activité, la recherche et l'innovation sont essentielles pour Orange, qui a dédié 726 millions d'euros à la recherche et l'innovation en 2015, avec près de 8000 employés dans les Orange Labs, dont 650 chercheurs et 140 doctorants. Par conséquent, Orange est le 1er opérateur Européen en termes de brevets déposés et 11ème au classement général du palmarès 2019 (INPI 2020) avec 230 brevets déposés pour un total de 8000 brevets possédés.

Organisation:

Au sein d'Orange, la Division Technologie et Innovation (TGI) accompagne la transformation d'Orange en opérateur multi-services. Elle rassemble des activités autour de la création d'innovations stratégiques, la Recherche et la mise en œuvre des politiques techniques et data pour le Groupe. Les Orange Labs (*Orange Labs Services* (OLS), *Orange Labs Networks* (OLN) et *Orange Labs Research* (OLR)) font partie de la division TGI et effectuent les activités de recherche et développement du groupe.

Une des sections d'OLS, la direction Communication, Commerce and Mobile Banking (CCMB) développe, de la recherche au déploiement en passant par la maintenance, les services de communication, de mobile Banking et de solution de paiement. Le département *Communication Immersion Transaction qualitY* (CITY) de CCMB s'occupe de développer de nouvelles solutions de communication, et est divisé en différentes sections: *Modelisation and Objective evaluation of the Voice quality of services* (MOV), *Home and Finance Communication* (HFC), *Technologies and Processing of Sound* et *Architecture*, *Projects, and End-to-End Expertise*.

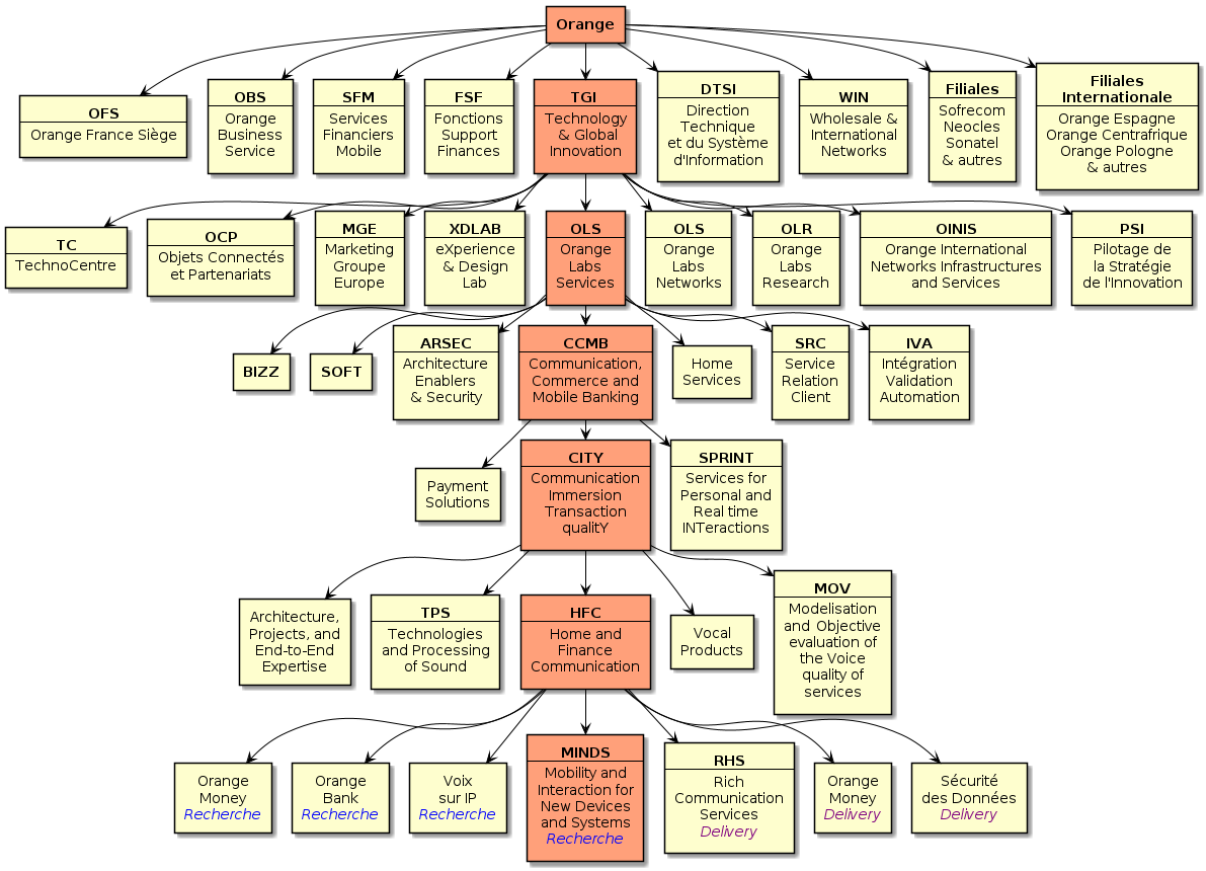


Figure 1 : La place de l'équipe MINDS au sein d'Orange

L'équipe MINDS

L'équipe MINDS (*Mobility and Interaction for New Devices and Systems*) fait partie de la section HFC du département CITY, et c'est en son sein que j'ai effectué mon stage de fin d'études. Son rôle est de produire des services et solutions robotique au service des autres équipes et départements d'Orange Labs. Elle est composée de sept personnes, un chef de projet, un chercheur, un thésard, un apprenti et trois stagiaires. L'équipe travaille sur différents domaines de la robotique tels que les interactions homme-machines via la reconnaissance vocale, le suivi de locuteur, la localisation et cartographie simultanées ou encore l'analyse de flux d'images.

Objectifs et enjeux du stage

L'objectif principal de mon stage était de reprendre les travaux précédemment effectués par Achraf et de les réutiliser afin d'aboutir à des services utilisables dans le cadre de l'inspection des datacenters d'Orange.

Nous avons identifiés deux services pouvant être rendus par le robot Geko: d'une part, un service de vérification/détection d'objets déjà connus, d'autre part, un service d'inventaire/catalogage des objets.

Définitions des services

Service d'inventaire/catalogage des objets:

Déplacement du robot dans un lieu connu déjà cartographié, sans connaître les objets

Objectif: cataloguer les objets et enregistrer leur type, taille et position

-Patrouille suivant un parcours prédéfini par l'utilisateur

-Détection et identification des objets, soit en continu soit à certains points d'intérêt

-Enregistrement des informations de type et de localisation des objets détectés

-Conversion des informations relevées vers le formalisme BIM (Building information modeling)

Optionnel:

-Exploration autonome du robot sans waypoints prédéfinis

Service de vérification/détection d'objets:

Déplacement du robot dans un lieu connu déjà cartographié, où les objets et leurs positions sont connus

Objectif: Vérification de présence d'objets déjà connus:

-Importer les informations de localisation des objets sur BIM vers la carte locale utilisée par le robot

-Patrouille suivant un parcours prédéfini pour aller vérifier les points d'intérêt

-Vérification sur place que l'objet est bien présent

-Prise en photo de l'objet (pour potentielle vérification humaine après passage du robot)

Optionnel:

-Mise en place de waypoints automatique pour que le robot aille devant chaque objet et puisse les voir avec la caméra de profondeur

-Demander au robot d'aller patrouiller à un endroit particulier

Matériel à disposition

Pour remplir les objectifs du stage, les deux services (inventaire et vérification de présence d'objets) doivent être fonctionnels sur le robot Geko de *Nuzoo robotics* qui servira de plateforme prototype.

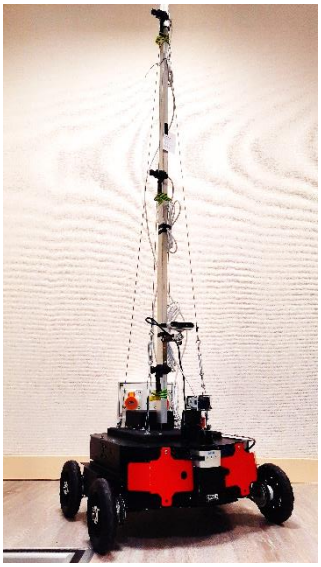


Figure 2 : Photographie du Geko équipé du mât et des trois caméras.

Ce dernier a été modifié par l'équipe au cours des précédents stages, et a été équipé de différents composants pour effectuer ses missions. De base, il possède un ordinateur de bord Intel NUC, un capteur LIDAR ainsi qu'une caméra de navigation et une centrale inertielle, afin de pouvoir se déplacer dans l'espace.

Lors des précédents stages, il a été progressivement équipé avec un mât sur lequel se trouvent trois caméras, une caméra de profondeur Realsense ainsi que les équipements nécessaires pour utiliser ces nouveaux capteurs.

L'Intel NUC n'étant pas assez puissant pour faire tourner Darknet_ros (l'algorithme de reconnaissance d'objets, nécessaire aux deux services), j'ai eu à ma disposition un kit de développement NVIDIA Jetson Xavier NX, un ordinateur de bord qui est bien plus puissant et qui a été conçu dès le départ pour faire fonctionner des réseaux neuronaux de manière optimisée avec faible une utilisation de la batterie (la consommation est de l'ordre de 10W). C'est donc sur la Jetson que sont implémentés Darknet_ros et les deux services créé pendant le stage. En ce qui concerne l'intégration de la Jetson au sein du robot, ce sujet est abordé dans la partie "Création et Implémentation".

GPU	NVIDIA Volta architecture with 384 NVIDIA CUDA® cores and 48 Tensor cores
CPU	6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6 MB L2 + 4 MB L3
DL Accelerator	2x NVDLA Engines
Vision Accelerator	7-Way VLIW Vision Processor
Memory	8 GB 128-bit LPDDR4x @ 51.2GB/s
Storage	microSD (not included)
Video Encode	2x 4K @ 30 | 6x 1080p @ 60 | 14x 1080p @ 30 (H.265/H.264)
Video Decode	2x 4K @ 60 | 4x 4K @ 30 | 12x 1080p @ 60 | 32x 1080p @ 30 (H.265) 2x 4K @ 30 | 6x 1080p @ 60 | 16x 1080p @ 30 (H.264)
Camera	2x MIPI CSI-2 DPHY lanes
Connectivity	Gigabit Ethernet, M.2 Key E (WiFi/BT included), M.2 Key M (NVMe)
Display	HDMI and display port
USB	4x USB 3.1, USB 2.0 Micro-B
Others	GPIO, I^2C, I^2S, SPI, UART
Mechanical	103 mm x 90.5 mm x 34.66 mm

Figure 3 : Spécifications techniques du kit de développement NVIDIA Jetson Xavier NX

De plus, le geko n'étant pas toujours disponible car fréquemment utilisé par le reste de l'équipe, j'ai aussi eu accès au Rosbot2 précédemment utilisé par Achraf, ainsi qu'à un Turtlebot3 modèle waffle_pi; et c'est ce dernier qui a servi pour l'ensemble des manipulations avant d'utiliser le geko.

Déroulement du stage

Calendrier

Avril	Recherche sur l'état de l'art sur RTAB-MAP et les travaux des précédents stagiaires Etude théorique de et prise en main de l'algorithme RTAB-MAP, Simulation sur Gazebo et prise en main de Geko => Abandon de RTAB-MAP et changement de sujet
Mai	Appropriation des travaux d'Achraf Mise en place de la reconnaissance d'objet via Darknet_ros Présentation du format BIM Mise en place de la localisation de l'objet dans la carte globale
Juin	Création du catalogage d'objet (enregistrement de position, taille, type et probabilité, filtrage pour éviter les doublons) Mise en place du service de patrouille (combiner reconnaissance et patrouille), création des waypoints automatique Vérification de présence d'objets connus
Juillet	Réorganisation du code et packages Mise à jour de la documentation Tests sur Turtlebot Enregistrement des objets dans un fichier de sauvegarde Intégration de la jetson sur le geko - Début
Août	Tests sur le geko Travail sur Thing'In : Faire le lien entre les données du robot et l'univers sémantique Palliatifs aux problèmes détectés lors des expérimentations
Septembre	Présentation prototype et services Documentation et sauvegarde des travaux Rendu produit final

Figure 4 : Calendrier de déroulement du stage

Difficultés rencontrées lors du stage

Dû à une mauvaise utilisation de l'intégration de package dans le répertoire git créé par Achraf, aucun de ses changements n'ont été enregistré dessus, ce dernier était donc vide en conséquence, rendant la réutilisation de ses travaux impossible. J'ai aussi passé du temps à chercher les fichiers interne de la Jetson sur laquelle il a travaillé ainsi que les back-ups, sans succès. J'ai dû par conséquence reprendre une bonne partie des travaux à zéro. Néanmoins, son rapport m'a permis de comprendre le fonctionnement général de ses travaux, et j'ai réutilisé certains algorithmes qu'il a mentionnés. Cependant, nous avons retrouvé ses travaux par chance au milieu du mois de Juillet, mais entre-temps la majorité des services ont été entièrement refait.

Bien que la perte temporaire de ces travaux ait été problématique, cela a néanmoins permit de souligner l'importance de disposer de back-ups et de bien utiliser le git afin de s'assurer que les fichiers que nous avons modifié soient bien envoyés sur le master et soient donc disponibles.

Un des autres problèmes auxquels j'ai fait face fut l'absence totale de commentaires, d'instructions et de directions sur les codes d'Achraf et d'Erwan, rendant la réutilisation de leurs travaux difficile; à un point que par moment il fut tout simplement plus simple de reprendre depuis le départ plutôt que de tenter de comprendre leur code. Par conséquent, j'ai pris le temps pendant mon stage d'une part de commenter mon code afin qu'une personne extérieure puisse comprendre son fonctionnement, d'autre part d'ajouter des instructions afin de reproduire les manipulations ou les simulations que j'ai effectué.

Une autre source de difficulté furent les robots Rosbot et Turtlebot. Bien que pouvant être démarré, le Rosbot (créé par la compagnie Husarion) n'arrivait pas à lancer ROS, malgré une bonne configuration et une réinstallation de ROS. Ce robot utilisant des packages et des configurations crées par Husarion, les correctifs classiques n'ont pas marché, et mes recherches sur les forums d'assistance technique d'Husarion fut infructueuse, et il fut décidé de le laisser de côté plutôt que de tenter une réinstallation. Quant au Turtlebot, il n'était tout simplement pas fonctionnel, j'ai dû changer son ordinateur de bord réinstaller son OS, initialement sur Raspbian puis sur Ubuntu à cause de problème de compatibilité avec realsense2-camera.

Sujet initial : RTAB-Map

Sujet et enjeux initiaux

L'objectif initial de mon stage était de faire correspondre des relevés 3D obtenus par le robot avec des cartes en format BIM déjà existante dans le but d'améliorer ces dernières dans le cadre de l'inspection de datacenters. L'application concrète étant qu'un technicien devant opérer dans un de ces datacenter puisse accéder facilement à un modèle en 3 dimensions du datacenter afin de se repérer facilement dans les lieux.

Pour cela, j'ai repris les travaux d'un précédent stagiaire, Erwann Landais ayant travaillé avant moi chez Orange Labs sur l'algorithme RTAB-Map, dans le cadre d'une stratégie de relocalisation d'un robot mobile terrestre. Bien que la relocalisation ait été laissée de côté dans le cadre de mon stage, j'ai néanmoins réutilisé les connaissances acquises par Erwann pour me former sur RTAB-Map

Principe de fonctionnement

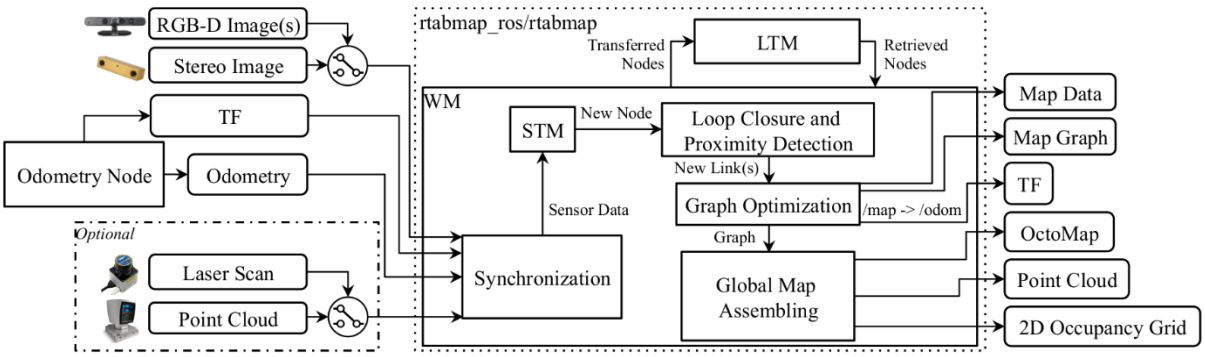


Figure 5 : Diagramme de fonctionnement de RTAB-Map [2]

En entrée, nous avons besoin de fournir des données odométriques, les transformations entre les capteurs et le robot, et enfin un nuage de point (venant soit d'une caméra RGBD ou d'une caméra stéréo). En sortie, il renvoie la position du robot (TF), la carte globale en 2D (Occupancy Grid) et 3D (Point Cloud), ainsi que les données du graph (nœuds et liens).

La structure de la carte créée par RTAB-Map est un graphe avec des nœuds et des liens. Après la synchronisation des capteurs, le module de mémoire à court terme (Short Term Memory/STM) crée un nœud mémorisant la pose de l'odométrie, les données brutes du capteur et des informations supplémentaires utiles aux modules suivants. Les caractéristiques

visuelles de l'image RVB sont extraites via des algorithmes comme SURF, SIFT, ORB ou BRIEF. Ces caractéristiques visuelles serviront ensuite à effectuer la détection de boucle ou de proximité. Lorsque l'odométrie visuelle est utilisée, il est possible de réutiliser les caractéristiques déjà extraites pour l'odométrie pour la détection de fermeture de boucle.

Il existe trois types de liens : Les liens de voisinage, de fermeture de boucle et de proximité. Les liens de voisinage sont ajoutés dans la STM entre des nœuds consécutifs avec une transformation odométriques. Les liens de fermeture de boucle et de proximité sont ajoutés par la détection de fermeture de boucle ou la détection de proximité, respectivement. Tous les liens sont utilisés comme contraintes pour l'optimisation du graphe.

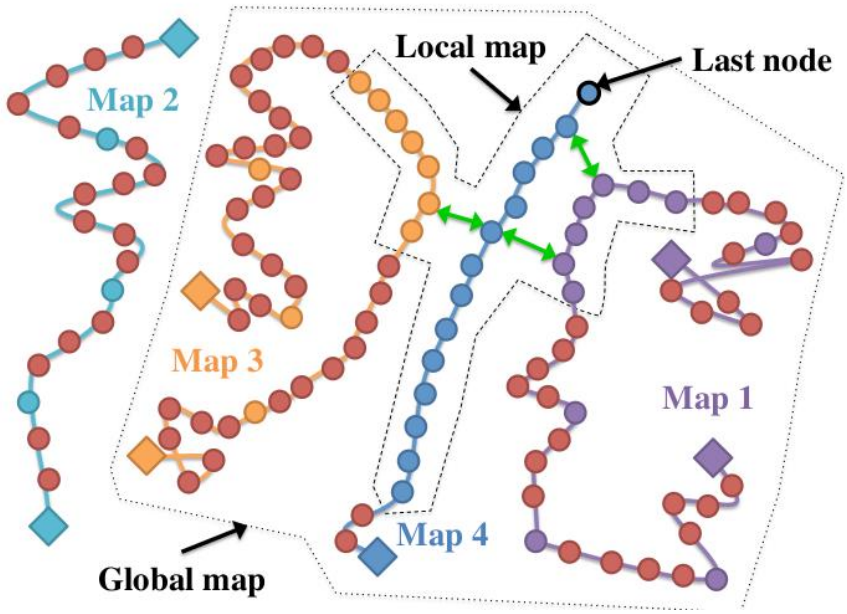


Figure 6 : Le graphe de fonctionnement de RTAB-MAP [1]

Plutôt qu'une carte, RTAB-Map créé un graphe global des différentes positions du robot contenant les données relevées. Ce graphe évolue à chaque fermeture de boucle ou lien de proximité, et c'est à partir des relations entre les nœuds que l'algorithme est capable d'assembler un nuage de point global.

Lorsqu'une nouvelle fermeture de boucle ou un nouveau lien de proximité est ajouté au graphe, le module d'optimisation du graphe corrige l'erreur calculée sur l'ensemble du graphe, afin de réduire la dérive de l'odométrie. De plus, si la transformation d'un lien dans le graphe après optimisation dépasse un seuil, tous les liens de fermeture de boucle et de proximité ajoutés par le nouveau nœud sont rejetés, gardant le graphe optimisé comme si aucune fermeture de boucle ne s'était produite, afin d'éviter les transformations invalides. Une fois le graphe optimisé, les sorties (Point Cloud et 2D Occupancy Grid) sont assemblées et publiées dans des modules externes.

Gestion de la mémoire

La méthode de gestion de la mémoire RTAB-Map est utilisée pour limiter la taille du graphe afin qu'il puisse être utilisé dans de grands environnements.

Sans gestion de la mémoire, au fur et à mesure que le graphe grandit, le temps de traitement des modules (par exemple, la fermeture de boucle ou la détection de proximité) peut finir par dépasser les limites du temps réel et le temps d'acquisition des nœuds, et RTAB-Map ne sera plus en mesure de traiter ces derniers en temps réel.

Fondamentalement, la mémoire de RTAB-Map est divisée en une mémoire à court terme (Short Term Memory/STM), une mémoire de travail (Working Memory/WM) et une mémoire à long terme (Long Term Memory/LTM). Lorsqu'un nœud est transféré dans la mémoire à long terme, il n'est plus disponible pour les modules de la mémoire de travail ou la mémoire à court terme.

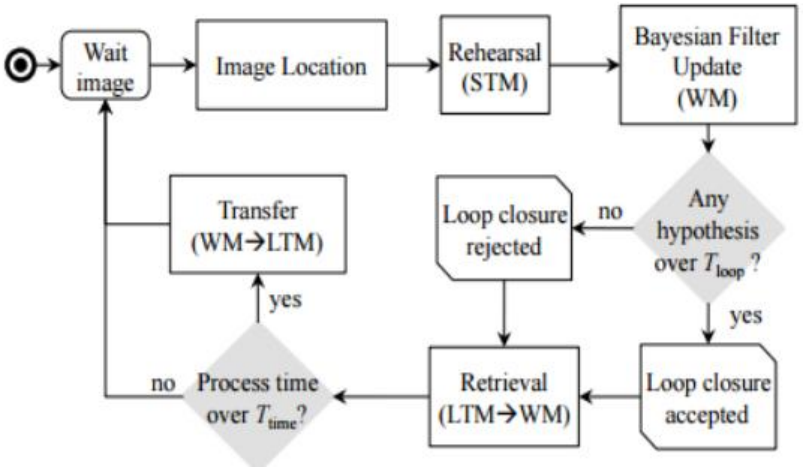


Figure 7 : Schéma de la gestion de la mémoire de RTAB-Map [4]

Afin de déterminer quels nœuds sont déplacés vers la mémoire à long terme, un poids est attribué à chaque emplacement au moment de sa création. Plus un lieu est observé longtemps, plus il est important, et il doit donc être laissé en mémoire de travail. Lorsqu'un nouveau nœud est créé, il est visuellement comparé au nœud précédent dans le graphe : s'ils sont similaires, le poids du nouveau nœud est augmenté de un plus le poids du nœud précédent. Le poids du dernier nœud est remis à zéro, et ce dernier est écarté si le robot ne se déplace pas, afin de ne pas augmenter inutilement la taille du graphe.

Lorsque le temps de mise à jour de RTAB-Map dépasse un seuil de temps fixe (T_{time}) ou un seuil de mémoire (utilisé pour définir le nombre maximum de nœuds que la mémoire de travail peut contenir), certains nœuds de la mémoire de travail sont transférés vers mémoire à long terme pour limiter sa taille et ainsi réduire le temps de mise à jour. Les nœuds les plus anciens et les plus petits sont transférés en premier vers la mémoire à long terme.

A l'inverse, lorsqu'une fermeture de boucle se produit avec un emplacement dans la mémoire de travail, les nœuds voisins de cet emplacement peuvent être ramenés de la mémoire à long terme pour les détections de proximité ou de fermetures de boucle. Lorsque le robot se déplace dans une zone précédemment visitée, il peut alors utiliser emplacements antérieurs pour étendre la carte assemblée actuelle et se localiser.

Expérimentations

Afin de prendre en main RTAB-Map avant de l'utiliser en réel, j'ai commencé par l'utiliser dans le cadre de différentes simulations, en premier lieu sur turtlebot3, car RTAB-Map possède déjà des fichiers de lancement compatibles avec Turtlebot, permettant ainsi de commencer à tester l'algorithme en action avec les paramètres par défaut. Ensuite, j'ai ajusté les simulations du robot geko pour y intégrer une caméra de profondeur, bloquer le mat d'observation (sans cela, il tournait librement avec l'inertie empêchant d'observer les lieux correctement), effectuer des simulations d'un geko se déplaçant dans le datacenter.

Lors de ces simulations, j'ai dès le départ fait face à différents problèmes. Le premier, est la taille des fichiers impliqués. Malgré le fait que RTAB-Map possède un module d'optimisation de graph, les nuages de points 3D restent très lourds, et l'export final de la carte assemblée dépasse facilement les 2/3 Gb au bout de 5 minutes de simulations. Cette taille élevée est problématique, d'une part car elle nécessite un stockage dédié dans le cas de cartographie de grands espaces; ensuite car la taille élevée rends la sauvegarde du nuage de point instable (RTAB-Map ou parfois même l'ordinateur crashent lorsque j'essaie de sauvegarder la carte résultante); et enfin car une taille élevée rends le nuage de points difficile à traiter ensuite.

Pour faire face à ce problème de taille de fichier, le seul moyen est de réduire la résolution de la caméra de profondeur et donc du nuage de point. Cependant, même en divisant par 4 la résolution, les ficher générés par RTAB-Map restent néanmoins très lourds quand nous voulons cartographier l'ensemble d'un bâtiment.

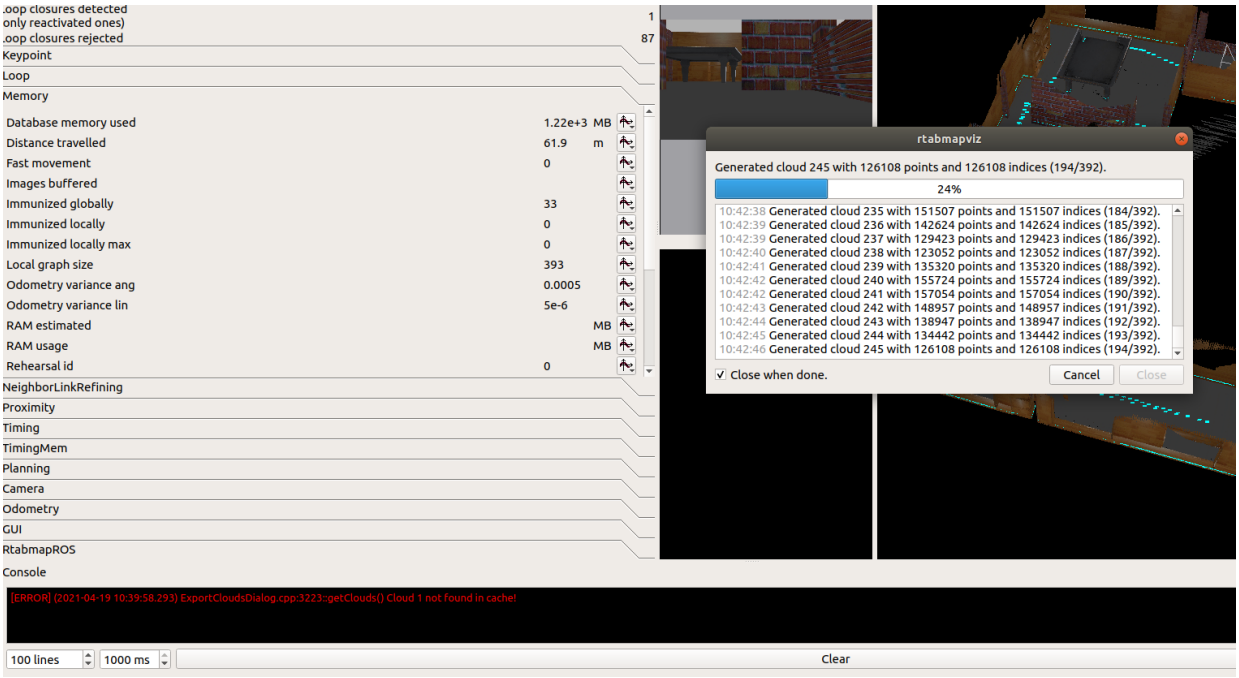


Figure 8 : Capture d'écran de RTAB-Map

1,2 Gb de mémoire utilisée, 126108 points, beaucoup trop lourd pour être utilisable, RTAB-Map a crashé avant de pouvoir terminer cet export

L'importante taille des fichiers est aussi gênante dans le cadre de la correspondance entre les plans existants du bâtiment et le nuage de point. Il faudrait applique des algorithmes de détection de surface sur des fichiers très lourds contenant beaucoup de points, ce qui risque de renvoyer trop de surfaces détectées pour que le résultat soit utilisable pour effectuer une correspondance. Bien qu'il soit possible de repasser à la main derrière pour corriger ce problème, un des objectifs du stage est justement de ne pas avoir à le faire afin de rendre le processus plus rapide et de nécessiter aussi peu d'intervention humaine que possible.

Enfin, la taille des fichiers rend directement leur utilisation, leur téléchargement et leur visualisation peu ergonomique, limitant ainsi le service rendu aux techniciens devant se rendre dans le datacenter, qui sont censés être les utilisateurs premiers du service.

Le second problème rencontré lors de l'utilisation de RTAB-Map est celui des mauvaises fermetures de boucles: dans des environnements où les pièces se ressemblent, ou dans le cas du datacenter dans la salle de serveurs où il y a plusieurs rangées de serveurs identiques; la haute ressemblance visuelle provoque une fermeture de boucle erronée, c'est à dire, le robot pense passer à un endroit où il est déjà passé, alors qu'il s'agit juste d'un autre endroit y ressemblant. Ces faux positifs provoquent dans la majorité des cas une déformation importante de la carte, la rendant ainsi totalement inutilisable.

Il est possible de contrer ces fermetures de boucles dans le cas de différentes pièces similaires, via le paramètre *MaxLoopClosureDistance*, permettant d'ignorer les fermetures de boucles à partir d'une certaine distance, permettant ainsi de bloquer une fermeture de boucle entre deux pièces identiques à 20 mètres de distance par exemple.

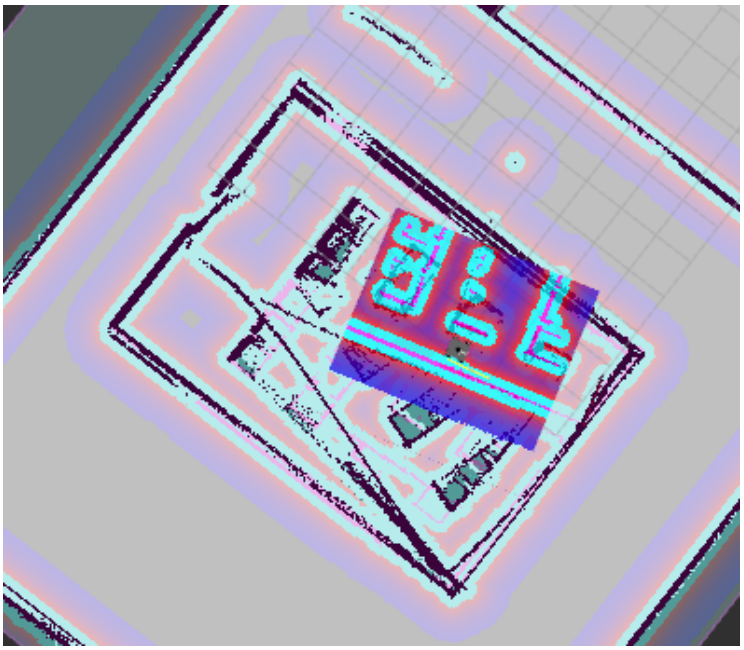


Figure 9 : Un exemple de fermeture de boucle erronée

Cette mauvaise fermeture de boucle a déformé et rendu la carte générée jusqu'à présent totalement inutilisable.

Cependant, ce paramètre ne convient pas dans le cas des datacenters, les serveurs sont très proches les uns des autres, nécessitant l'utilisation d'une faible distance de fermeture de boucle maximale, ce qui risque d'empêcher des fermetures de boucles légitimes après une longue durée à cause de la dérive odométrique.

Conclusions et abandon du sujet

Aux problèmes de taille de fichier et de fermeture de boucles s'ajoutent des problèmes d'odométrie liés au matériel présent sur le geko, les problèmes d'utilisation des nuages de points par l'algorithme ainsi que les retours d'expériences des précédents tests de RTAB-Map qui ont été assez décevant.

De plus, au-delà de l'utilisation de RTAB-Map, ce sujet aurait impliqué une partie très mathématique pour effectuer la correspondance entre les nuages de points et les plans des bâtiments, ce qui n'aurait pas eu beaucoup de rapport avec mes compétences en robotique.

En prenant en compte la somme de ces problèmes, nous avons estimé que tenter de faire marcher RTAB-Map nécessiterait beaucoup d'investissement, pour un rendu qui risque d'être décevant en plus de ne pas apporter beaucoup d'informations directement utilisables. Il fut donc décider d'abandonner ce projet, et de reprendre à la place les travaux d'Achraf (un autre stagiaire) afin de créer des services utilisables.

État de l'art

YOLO – *You Look Only Once*

La reconnaissance visuelle d'objets est essentielle aux deux services (inventaire et vérification d'objets). Afin d'effectuer cette dernière, j'ai utilisé Darknet_ros, une implémentation ROS de l'algorithme YOLO qui a aussi été utilisée par Achraf avant moi, et qui permet d'effectuer une détection d'objets en temps réel sur un topic d'image que nous lui fournissons.

Présentation

YOLO (*You Look Only Once*) est un algorithme unifié de détection en temps réel d'objets. Cet algorithme utilise un seul réseau neuronal entraîné dès le départ sur des images complètes pour prédire directement à partir d'images complètes les boîtes de délimitation et les probabilités de classe de ces boîtes en une seule évaluation. Avec cette méthode, il suffit donc de regarder une seule fois (YOLO) une image pour prédire quels objets sont présents et où ils se trouvent.

YOLO est utilisable sur ROS via le package Darknet_ros, disponible pour les versions Melodic, Noetic, Foxy et ROS2 (différentes branches sont disponibles sur le Git).

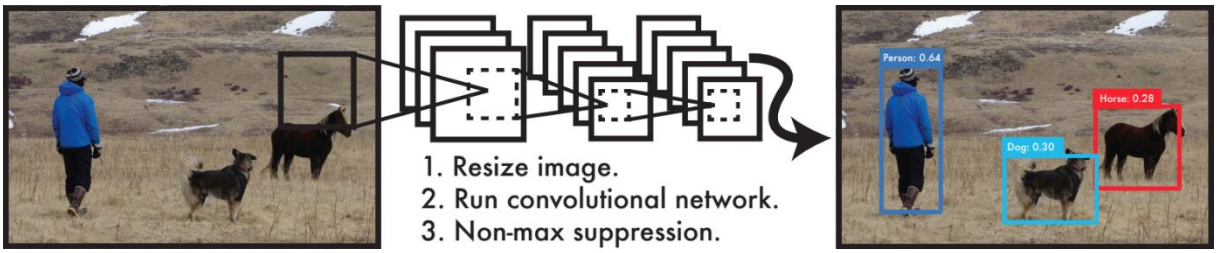


Figure 10 : Le système de détection YOLO [5]

Le traitement des images avec YOLO est simple et direct. L'algorithme redimensionne l'image d'entrée à 448 × 448 (1), exécute un seul réseau convolutif sur l'image (2), et seuille les détections résultantes en fonction de la confiance du modèle (3).

Avantages et limites de l'algorithme

Le premier avantage de cette méthode est sa rapidité. Tandis que d'autres algorithmes utilisent une fenêtre glissante (ils appliquent le classifieur sur seulement une portion de l'image qu'ils font bouger) ou délimitent l'image en régions, nécessitant d'appliquer des classifieurs à

chacune de ces régions; YOLO n'exécute qu'un seul réseau convolutif sur l'image, le rendant bien plus rapide comparé à ces algorithmes du fait de l'absence de pipeline complexe. Lors de l'utilisation de Darknet_ros sur la Jetson Xavier, nous observons entre 60 et 200 reconnaissances faites par seconde avec le modèle yolov2-tiny, et 30 images par secondes avec le modèle yolov3. Cependant, il est important de configurer Darknet_ROS correctement, et d'utiliser le GPU (via CUDA sur des cartes Nvidia) afin d'obtenir ces performances, sinon Darknet_ROS tourne avec une vitesse entre 0.5 et 2 images analysées par seconde.

Ce point-là est très important, puisqu'il permet de traiter les images sans latence, et donc d'utiliser Darknet_ROS en temps réel.

Ensuite, l'utilisation dès de départ d'image globales fait que YOLO prend en compte le contexte de l'image, là où les méthodes à fenêtre glissante (comme Fast R-CNN, un algorithme de reconnaissance d'objets concurrent) ne peuvent pas le faire du fait de l'utilisation de parties de l'image séparées. Cela permet d'éviter de confondre des taches d'arrière-plan avec des objets, YOLO fait moins de la moitié du nombre de faux positifs à cause de l'arrière-plan par rapport à Fast R-CNN (4.75% pour YOLO contre 13.6% pour Fast R-CNN).

Cependant, YOLO reste moins précis que ces algorithmes en ce qui concerne les petits objets, en particulier leur localisation, où le taux d'erreur est de 19.0% pour YOLO face à 8.6% pour Fast R-CNN.

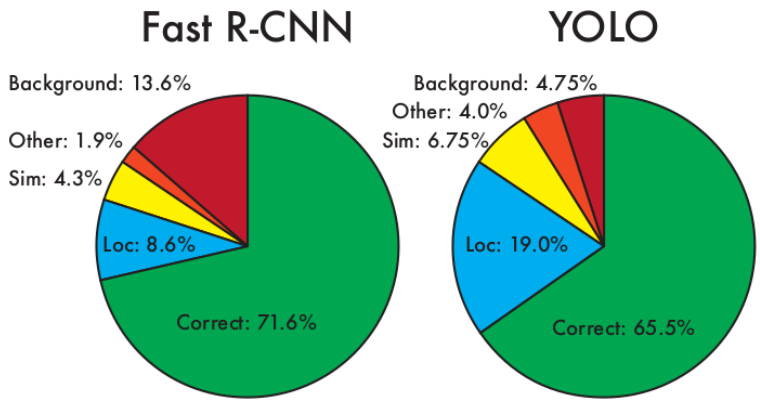


Figure 4: Error Analysis: Fast R-CNN vs. YOLO These charts show the percentage of localization and background errors in the top N detections for various categories (N = # objects in that category).

Figure 11 : Comparaison des types d'erreurs entre YOLO et Fast R-CNN [5]

Néanmoins, malgré ces imprécisions, YOLO propose à ce jour le meilleur rapport entre vitesse d'utilisation et précision, et est l'algorithme le plus adapté pour utilisation dans le cadre de la robotique embarquée.

Principe théorique

L'image est en premier lieu divisée en S x S cases. Si le centre d'un objet se trouve dans une cellule de la grille, cette cellule est responsable de la détection de cet objet.

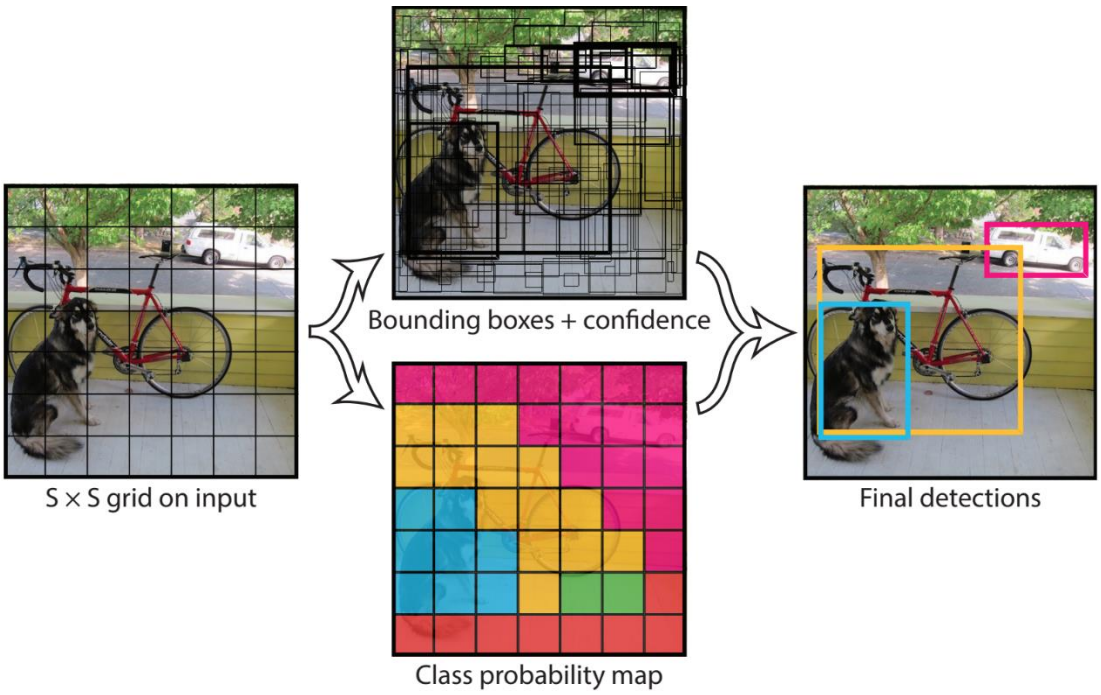


Figure 12 : Illustration du principe de fonctionnement de YOLO [5]

En parallèle, YOLO utilise les caractéristiques de l'image entière pour prédire des boîtes de délimitation d'objets ainsi que les scores de confiance qui y sont associés (confiance du modèle dans le fait que la boîte contient un objet ainsi que la précision de cette boîte). Chaque boîte de délimitation est définie par les coordonnées de son centre, sa taille ainsi que son score de confiance.

Enfin, l'algorithme associe à chaque cellule de la grille des probabilités conditionnelles de classe, c'est à dire des probabilités que la cellule contient un objet de cette classe. En multipliant les prédictions de confiance des boîtes de délimitation avec les probabilités conditionnelles de clase, nous obtenons des scores de confiance spécifiques à chaque classe pour chaque case. Ces scores codent à la fois la probabilité que cette classe apparaisse dans la boîte et la façon dont la boîte prédite correspond à l'objet.

Expérimentations avec Darknet_ros et Darknet_ros_3d

Avant d'implémenter Darknet_ros sur les robots, nous l'avons d'abord testé directement en branchant simplement la caméra à l'ordinateur. Il est important de le configurer correctement afin d'utiliser le GPU.

Par défaut; un problème dans l'algorithme fait que le format de l'image est mal interprété, et l'image RGB est convertie incorrectement en BGR. Comme les couleurs jouent un rôle dans la reconnaissance des objets, nous convertissons donc au préalable les images RGB en images BGR afin que la seconde conversion effectuée par Darknet_ros rétablisse l'image dans son format correct.

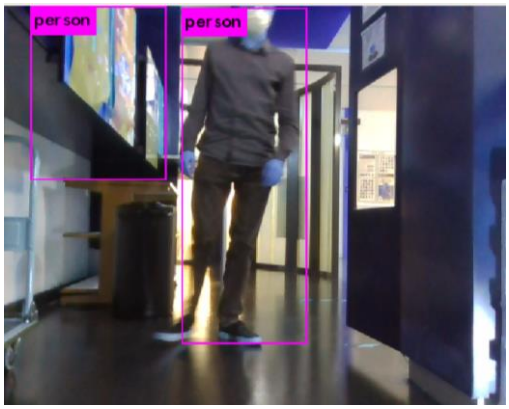


Figure 13 : Sortie de Darknet_ros avec canaux de couleur rouges et bleus inversés.

Malgré le format de couleur erroné, Darknet_ros arrive à détecter sans problème une personne qui se trouve devant la caméra (ici avec une confiance de 86%). Il y a aussi un faux positifs, cependant ce dernier a une confiance faible (30%) ce qui permet de le filtrer.

Lors des expérimentations, nous avons vu que Darknet_ros marche correctement dans la plupart des cas (avec le modèle yolov2-tiny). Cependant, le modèle n'est pas parfait, en particulier dans le cas d'objets que nous voyons partiellement, des petits objets, ou dans le cas d'un ensemble d'objets proches les uns des autres sur le bureau. L'algorithme a tendance à considérer toutes les surfaces noires comme des écrans TV, y compris les claviers où la visibilité des touches dépend beaucoup de l'éclairage. L'algorithme a du mal à délimiter correctement les chaises, et voit souvent deux chaises là où il n'y en a qu'une seule. Cependant, l'algorithme ne se trompe presque jamais lors de la détection d'humains (probablement dû au fait qu'il n'y a pas de classes y ressemblant).

Néanmoins, lorsque l'algorithme est en difficulté et commet des erreurs, sa confiance dans la présence des objets anormalement détectés est faible, et avec un simple filtre sur cette confiance nous pouvons obtenir facilement uniquement les objets avec une confiance élevée. Généralement, au bout de 80% de confiance il n'y a plus d'erreurs, et l'algorithme monte très rarement au-delà de 90%, sauf pour les humains.

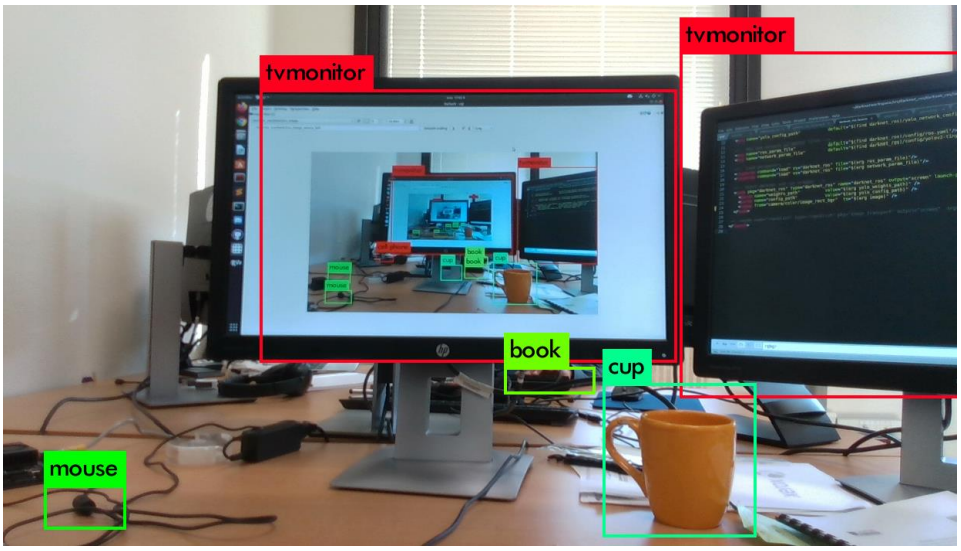


Figure 14 : Darknet_ROS en action.

Malgré quelques faux positifs, Darknet_ROS leur attribue une faible probabilité, ce qui permet de les filtrer sans problèmes dans nos services.

Nous avons aussi testé les modèles yolov2, yolov3, et yolov4. Yolov3, la dernière version "officielle" de yolo crée par Joseph Redmon marche aussi correctement, et est plus lente que yolov2-tiny, tout en restant adapté pour une utilisation en temps réel malgré tout. Le modèle yolov2 est plus lent que yolov2-tiny, sans offrir la précision de yolov3, et il n'y a aucune raison de l'utiliser quand nous pouvons utiliser yolov2-tiny (pour la vitesse) ou yolov3 (pour la précision) à la place. J'ai voulu tester yolov4, cependant cette version nécessite beaucoup plus de mémoire que disponible, et ne fonctionne ni sur l'ordinateur ni sur la jetson. Enfin, yolov5 ne dispose pas d'implémentation directe sur darknet_ros, et fut laissé de côté par conséquent.

Quant à Darknet_ros_3d, il s'agit d'un package combinant la sortie de darknet_ros avec le nuage de points fournit par la caméra de profondeur, afin de générer des boîtes de délimitation en 3 dimensions. Il n'y a pas d'informations disponible sur comment ce package marche sur le git, néanmoins, un examen des fichiers internes révèle qu'il s'agit simplement d'extraire les dimensions minimales et maximales du nuage de points dans les trois axes sur l'aire délimitée par les boîtes de délimitation 2D générées par darknet_ros. Le package marche correctement et a peu d'impact sur la précision de la reconnaissance et de la localisation d'objets, la justesse des boîtes 3d dépendant surtout de celle des boîtes 2d et donc de darknet_ros.

Création et implémentation des services

Principe de fonctionnement des services

Scripts et fonctions communes

Bien qu'étant différents, nos deux services disposent malgré tout de scripts, fonctions et package utilisés en communs.

Les premiers packages utilisés par nos deux services sont Darknet_ros et Darknet_ros_3d, utilisés pour effectuer la reconnaissance et localisation locale des objets, ils sont par conséquent au cœur des deux services. Darknet_ros utilise le flux d'image envoyé par la caméra et effectue une reconnaissance d'objets sur ce flux. Il renvoie en sortie les objets détectés, leur type, leur probabilité de détection, ainsi que leur position sur l'image. Ces informations sont ensuite couplées avec le nuage de points par darknet_ros_3d afin de localiser localement l'objet en trois dimensions par rapport à la caméra; renvoyant en sortie le type des objets détectés, leur probabilité de détection, et les coordonnées en trois dimensions de la boite les contenant dans le repère de la caméra. Darknet_ros est complémenté par le script *color_bgr.py*, qui s'occupe de convertir l'image du format RGB au format BGR.

Nous avons aussi un service ROS *serveur_image.py*, qui permet d'enregistrer les images captées par la caméra ou les images renvoyées par Darknet_ros avec annotations chaque fois qu'il est appelé. Nous l'utilisons pour enregistrer les images à des fins de vérifications pour tester les résultats des deux services.

Ensuite, via le script *generate_waypoints.py* nous créons une liste de waypoints depuis une liste d'objets à observer et une carte des lieux: à partir de la carte des lieux, le script applique un traitement d'érosion (qui élargit les bords des obstacles, afin d'éviter d'avoir un waypoint adjacent à un mur ou autre obstacle), crée sur chaque objet un cercle noir qui a pour rayon la distance minimale d'observation, et enfin trouve le point blanc le plus proche de l'objet afin d'obtenir les coordonnées du point d'observation et l'orientation à adopter pour voir l'objet. Les waypoints générés ainsi sont enregistrés dans le fichier *waypoints.yaml* afin de permettre leur utilisation ultérieure par le nœud de vérification de présence d'objets.

Figure 15 : Le script de génération de waypoint pour observation d'objets.

Les positions des objets à observer sont affichées en magenta, les points d'observation en vert. À gauche, la carte originale, à droite, la carte après érosion et ajout de cercles autour des objets.

Enfin, nous utilisons un script *move_goal.py*, qui définit un *serveur d'action ROS* permettant de se déplacer à un point sur la carte, et de bloquer l'action du script tant que le déplacement n'est pas terminé. Nous avons aussi en retour le statut du déplacement permettant de savoir s'il s'est bien effectué ou non.

Inventaire

Pour le nœud d'inventaire, Nous utilisons les informations obtenues par darknet_ros_3d et les informations de positions, afin de répertorier les objets détectés sur la carte globale. Cette partie est commune aux nœuds d'inventaire et de vérification de présence d'objets. Cependant, nous devons effectuer un filtrage des entrées avant de sauvegarder les objets.

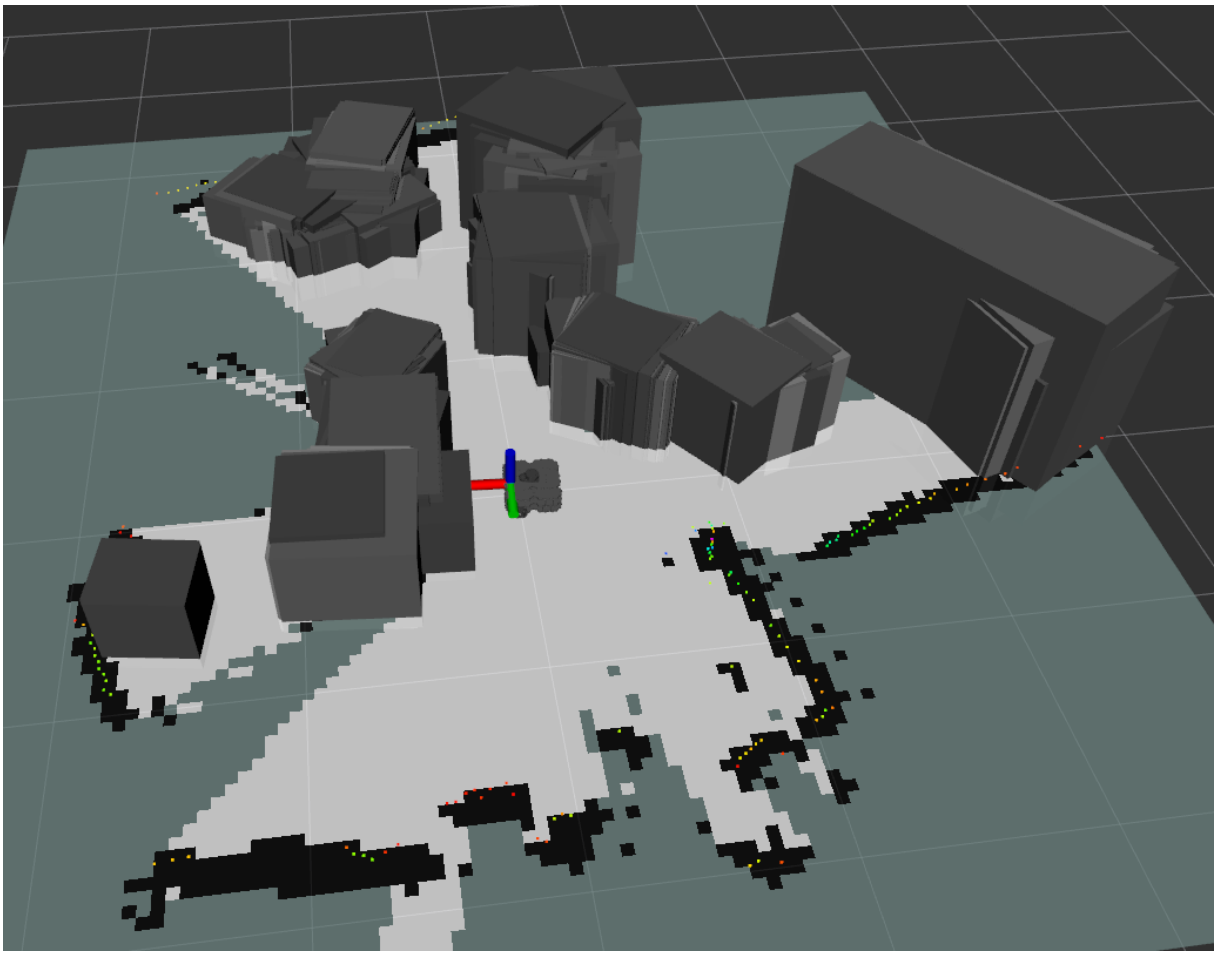


Figure 16 : Le nœud d'inventaire en action sans aucun filtre.

Les mêmes objets sont enregistrés des dizaines de fois ainsi que tous les faux positifs, et le résultat final est inutilisable.

Le premier filtre est un filtre de probabilité: le nœud n'enregistra jamais un objet dont darknet n'est pas absolument sûr du type, ici le seuil a été fixé à 80% empiriquement (70% avait trop de faux positifs, 90% était trop élevé pour la majorité des objets), les objets en dessous seront ignorés complètement par les tests. Ce filtre est très important, car Darknet_ros est très susceptible d'avoir des faux positifs, en particulier dans des environnements complexes (plusieurs objets proches, certains en cachant d'autres, nombreux câbles)

Le second filtre est un test qui a pour but d'empêcher d'enregistrer plusieurs fois le même objet. Tant que nous n'avons croisé aucun objet, nous sommes sûr que les objets que le robot croisera n'auront jamais été vus auparavant par le robot, ils peuvent donc être enregistrés sans problème par le robot. Cependant, une fois les premiers objets dans la liste d'inventaire, tous les objets suivants vus par le robot seront comparés aux objets existants, pour être sûr de ne pas enregistrer plusieurs fois le même objet.

Pour que deux objets détectés soient considérés identiques, il faut qu'ils aient :

- le même type (par exemple deux chaises)
- des volumes similaires (différence de volume maximale définie par l'utilisateur dans un fichier de configuration, généralement l'algorithme tolère un volume maximal 4 fois plus grand/petit
- des positions proches (distance maximale définie par l'utilisateur, dépends du cas d'usage, empiriquement définie à 50cm).

Dans le cas de l'inventaire, comme nous voulons éviter les faux négatifs (c'est à dire, que l'algorithme considère un objet déjà dans la liste comme nouveau à cause de légères différences), les critères sont allégés, afin qu'un même objet soit considéré comme identique à lui-même d'un instant à l'autre.

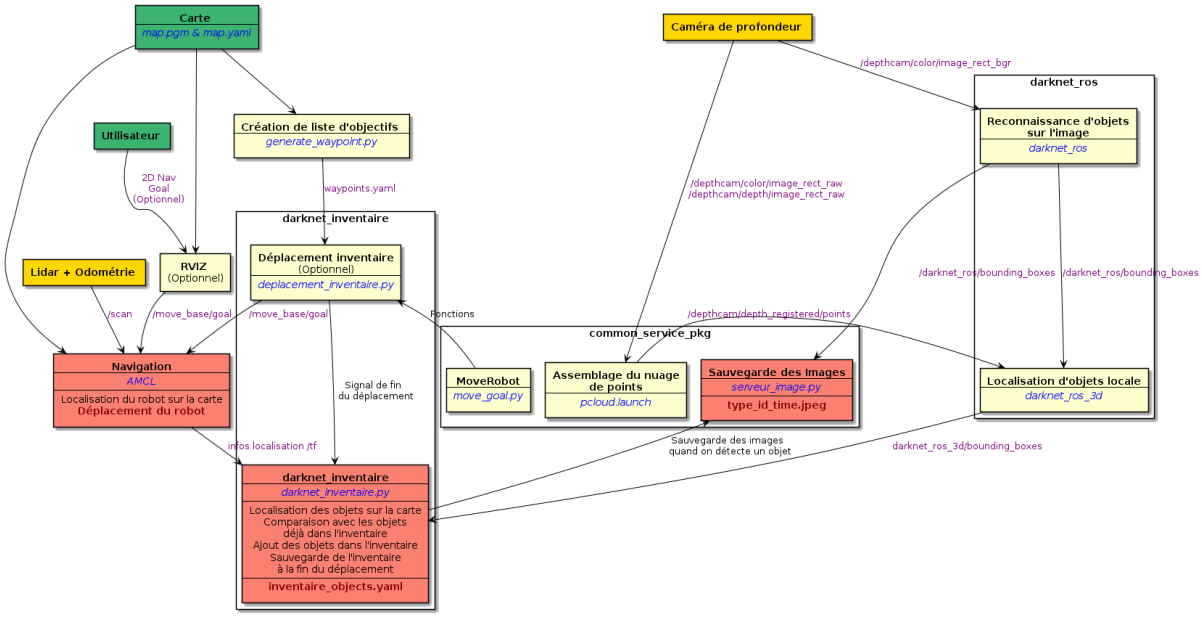


Figure 17 : Schéma de fonctionnement du nœud d'inventaire
(Version plus grande en annexe)

Une fois tous les tests passés et que nous sommes sûr que nous détectons un nouvel objet, d'une part nous enregistrons cet objet dans la liste, d'autre part nous enregistrons aussi une photographie de l'objet, afin de permettre une vérification humaine une fois le nœud d'inventaire terminé.

Vérification de présence d'objets

Le fonctionnement du service de vérification de présence d'objets est simple à comprendre. A partir d'une liste d'objets dont nous connaissons les positions, tailles, et types,

nous générons une liste de waypoints (avec le script generate_waypoint.py) qui sera importée par le nœud principal. Lors du lancement du nœud principal, celui-ci demande à l'utilisateur de fournir l'id de l'objet à vérifier via le terminal. Une fois l'id fournit au robot, celui-ci envoie un signal au nœud de détection indiquant d'une part qu'il faut commencer la détection, d'autre part l'id de l'objet à vérifier. Ensuite, pendant que le nœud de détection tourne en parallèle, le nœud principal commence le déplacement vers le point d'observation via le serveur d'action de déplacement, bloquant celui-ci jusqu'à la fin du déplacement.

Si au cours du déplacement l'objet est détecté par le nœud de détection, celui-ci l'indique à l'utilisateur via le terminal, enregistre que l'objet a été détecté et appelle le service de sauvegarde d'image. Une fois le déplacement terminé et une attente de quelques secondes, le nœud principal envoie un signal au nœud de détection lui indiquant de mettre fin à la détection et lui demandant le statut de l'objet. Si l'objet a déjà été détecté, il est inutile de prendre une capture d'image puisque cela a déjà été fait lors de la détection de l'objet. Si au contraire l'objet n'a pas été détecté, nous faisons appel au service d'enregistrement d'image pour en prendre une afin d'être en mesure de vérifier si l'absence de détection est normale ou non. Enfin, le script demande à nouveau à l'utilisateur de fournir l'id de l'objet qu'il veut vérifier. A tout moment, plutôt que d'envoyer un id, l'utilisateur peut aussi envoyer la commande "exit" pour mettre fin au service.

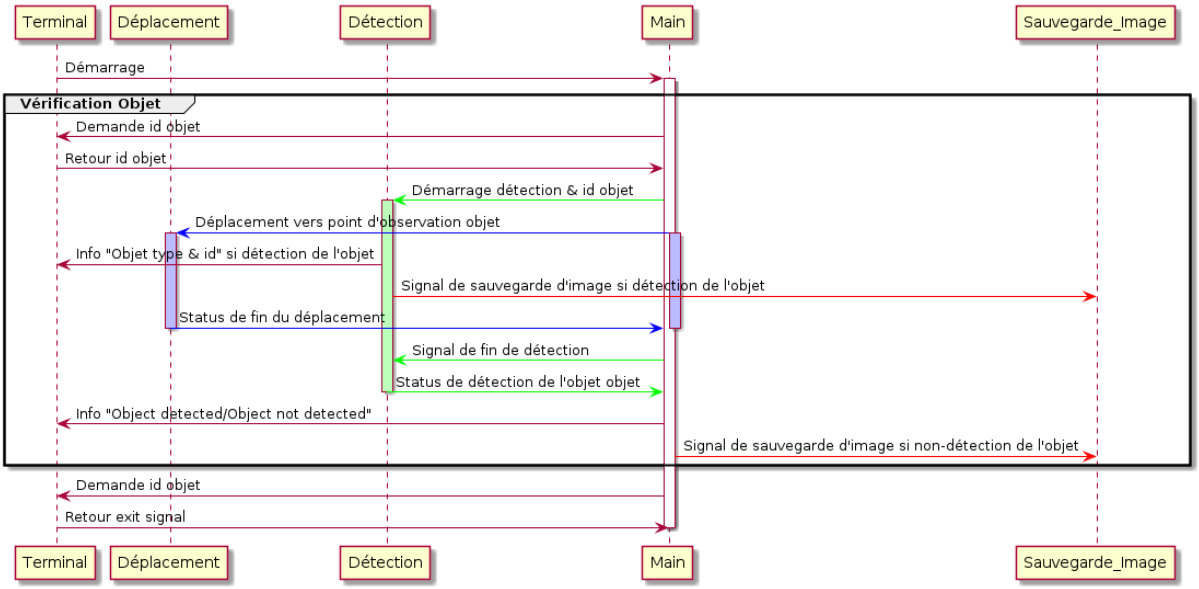


Figure 18 : Le call-flow du service de vérification de présence d'objets

Comme pour le nœud d'inventaire, nous couplons les informations de positions du robot avec les informations de localisation des objets dans le repère de la caméra pour localiser ces derniers dans la carte globale. Nous réutilisons le même filtre de probabilité, pour à nouveau éviter les objets parasites.

Tandis que le nœud d'inventaire enregistre à chaque fois les objets qu'il détecte, dans le cadre de la vérification de présence d'objets nous connaissons déjà les objets qui nous intéressent, leurs types, positions et tailles. Les objets détectés par darknet ne sont donc pas enregistrés. Ils sont juste comparés aux listes d'objets à détecter, afin de déterminer si l'objet fait partie de la liste ou pas.

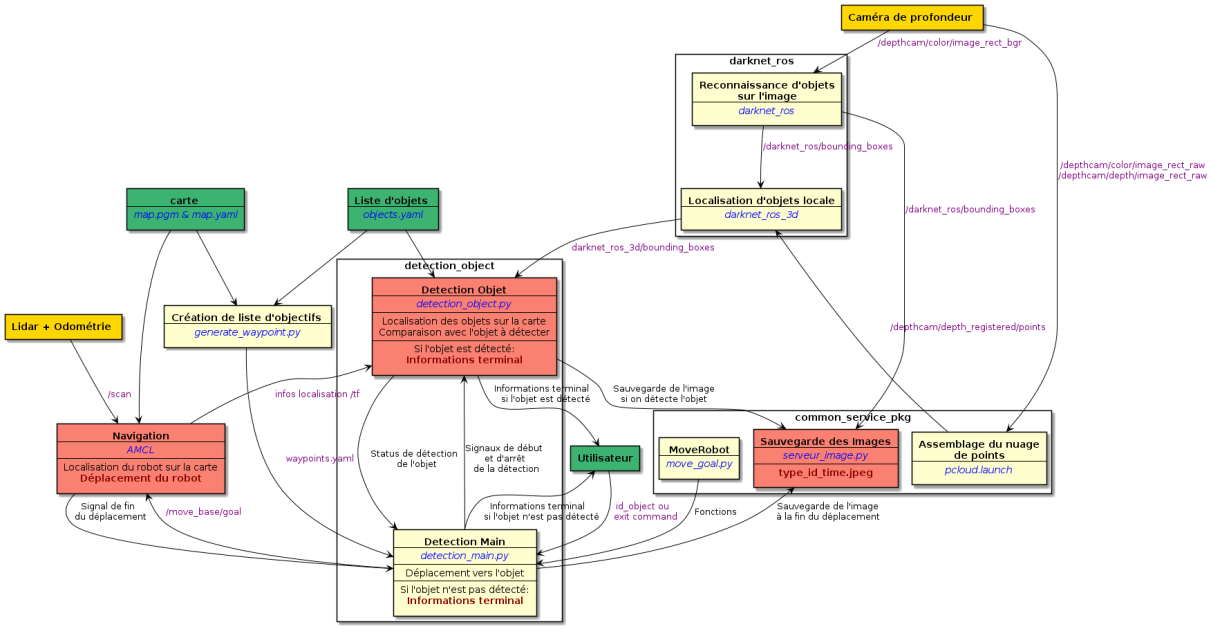


Figure 19 : Schéma de fonctionnement du nœud de vérification de présence d'objets (Version plus grande en annexe)

A l'inverse de l'inventaire, nous voulons éviter à tout prix les faux positifs, c'est à dire des objets détectés comme faisant partie de la liste alors qu'ils sont différents dans la réalité. Les tests pour savoir si les objets sont considérés comme identiques restent les mêmes, mais les critères sont donc renforcés, afin que seul un objet qui fait bien parti de la liste soit accepté.

Intégration de la Jetson sur le Geko

Nécessité de l'intégration de la Jetson

Afin de pouvoir effectuer la reconnaissance, la détection et la localisation d'objets, nous utilisons les algorithmes darknet_ros et darknet_ros_3d. Cependant, ces algorithmes nécessitent beaucoup de ressources pour pouvoir fonctionner.

Darknet_ros fonctionne à environ 0.1/0.2 images par seconde avec un temps de latence d'environ 10/15 secondes sur le geko. Par comparaison, lors des tests sur la jetson, celui-ci fonctionnait à 30-50 images par seconde, sans latence. Pour utiliser darknet_ros de manière fluide sur le geko, nous avons donc besoin de passer par la Jetson Xavier NX, conçue pour ce type d'applications.

Intégration électrique

Le port d'alimentation de la Jetson est un Power jack standard, de dimensions 5.5×2.5×9.5 mm (OD x ID × longueur). La consommation de la Jetson est de l'ordre de 10W par défaut, mais il y a des modes de consommation à 15W; dans les deux cas la Jetson nécessite une alimentation entre 9 et 20V (elle ne prend que la tension dont elle a besoin), avec un ampérage maximal est de 4,4A. Or, notre batterie fournit en sortie une tension de 26V, nous sommes donc obligé d'utiliser un limiteur de tension afin d'obtenir un voltage d'entrée utilisable par la Jetson.

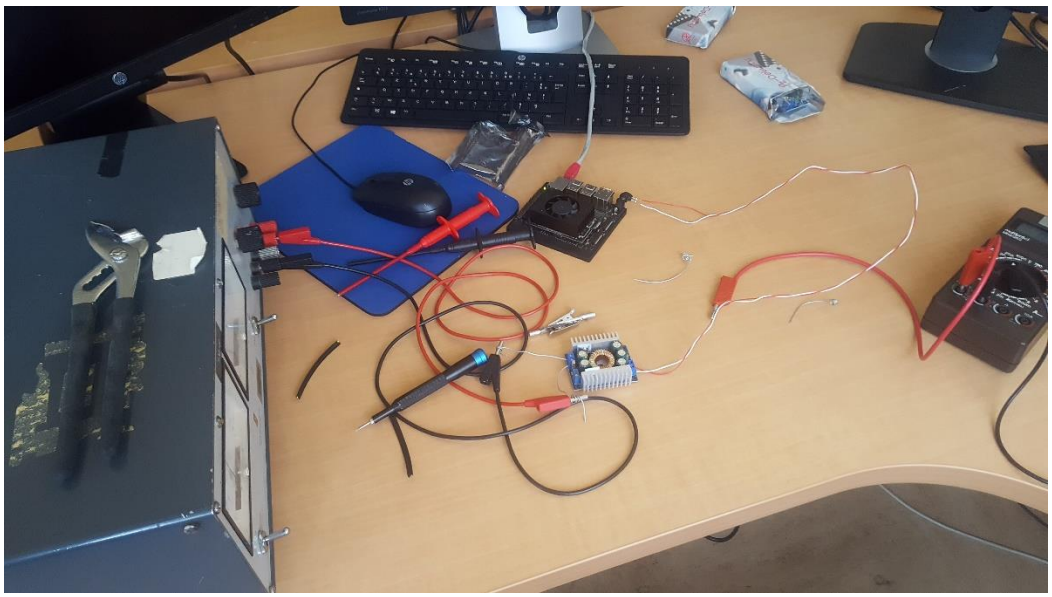


Figure 20 : Test du limiteur de tension abaissant la tension d'entrée de 26V à 12V

Intégration réseau

L'algorithme darknet_ros_3d utilise en entrée d'une part les sorties de darknet_ros, d'autre part un nuage de points 3d. Or, les nuages de points sont trop volumineux pour être transmit rapidement via le wifi. Par conséquent, il est nécessaire d'utiliser un câble Ethernet pour relier la Jetson au Geko, afin de transmettre rapidement les informations du nuage de point pour leur traitement sur la Jetson.

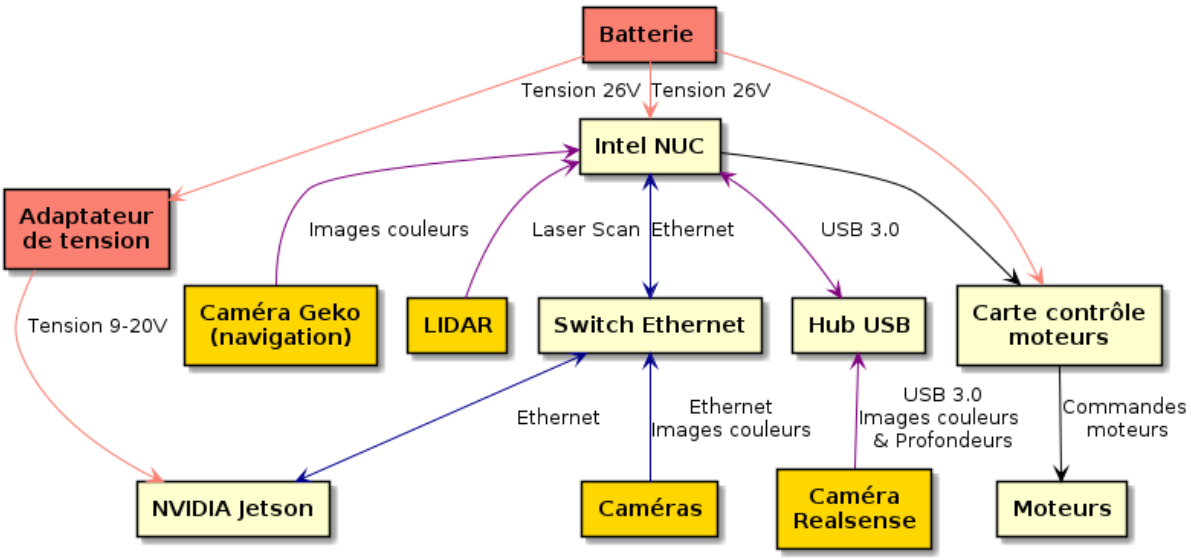


Figure 21 : L'architecture matérielle de Geko, après intégration de la NVIDIA Jetson.

Réseau sémantique

Définitions

Un réseau sémantique est une base de connaissances qui représente les relations sémantiques, c'est à dire les liens de significations entre différents concepts, entités et classes au sein d'un réseau. Les principales relations sémantiques sont la relation d'équivalence, la relation hiérarchique et la relation associative. Un réseau sémantique peut prendre plusieurs formes, par exemple, une base de données de graphes ou une carte conceptuelle.

Une ontologie est un réseau sémantique qui définit des catégories, des propriétés et des relations entre des concepts, des données et des entités au sein d'un univers étudié. Plus simplement, une ontologie est un moyen de montrer les entités d'un univers et la façon dont elles sont liées, en définissant un ensemble de concepts et de catégories qui représentent cet univers.

Thing'In

Thing'In (aussi appelé Things in the future) est une plateforme ontologique créé par Orange ayant pour objectif de maintenir des descriptions sémantiques d'environnements complets (par exemple bâtiments ou complexes industriels) dans lesquelles se trouvent des objets.

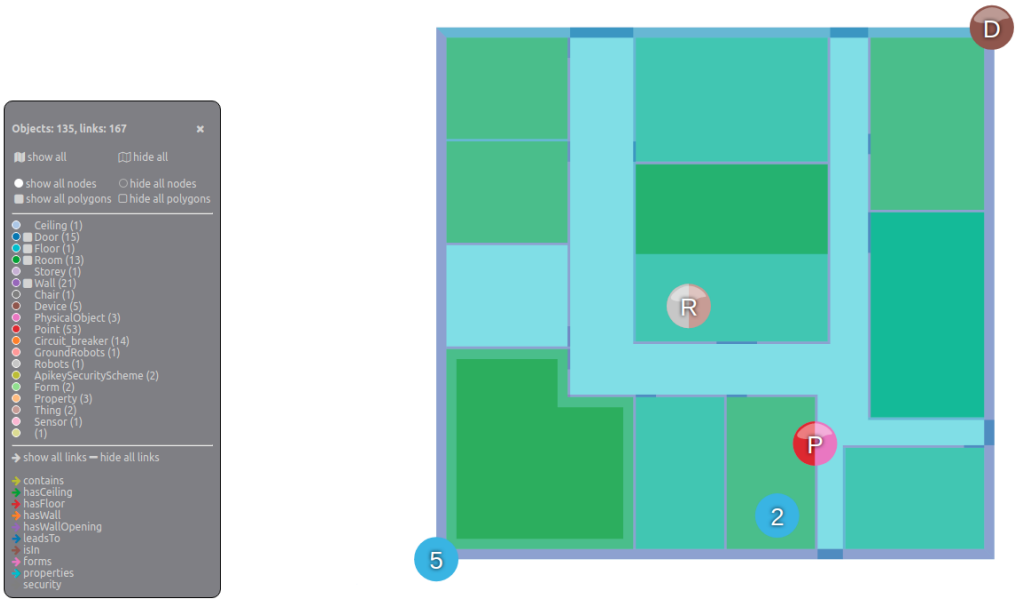


Figure 22 : Visualisation de plan 2D sur Thing'In.

Ces descriptions peuvent prendre la forme d'un graph en 2 ou 3 dimensions, mais aussi la forme de plan, que ce soit géoréférencé sur une carte ou dans le référentiel de l'environnement.

Thing'In ne relaie pas les données entre les différentes entités mais simplement les relations entre elles; dans le but de permettre à différentes applications IoT (Internet of Things) d'utiliser ces connaissances.

L'interface Thing'In permet de visualiser les différentes ontologies sous différentes formes: La première de ses formes est un graph 2D, qui sert à visualiser les différents liens ontologiques entre les différents objets. Les graphs 3d fournissent les mêmes informations que le graph 2d, mais la 3ème dimension permet de rendre plus simple leur visualisation. Enfin, Thing'In fournit la possibilité de voir les objets dans l'espace, soit en visualisant un plan 2D des objets dans un référentiel global ou en affichant les objets sur une carte vie les informations GPS qui ont été fournies à Thing'In.

Dans le cadre de mon stage, j'ai utilisé Thing'In afin de faire le lien entre l'ontologie et les données et objets détectés par le robot; afin de rendre ces derniers accessibles par d'autre applications. Pour créer, mettre à jour des objets sur Thing'In ou au contraire les importer, nous passons par l'API (Application Programming Interface) de Thing'In, qui est aussi intégrée au site web.

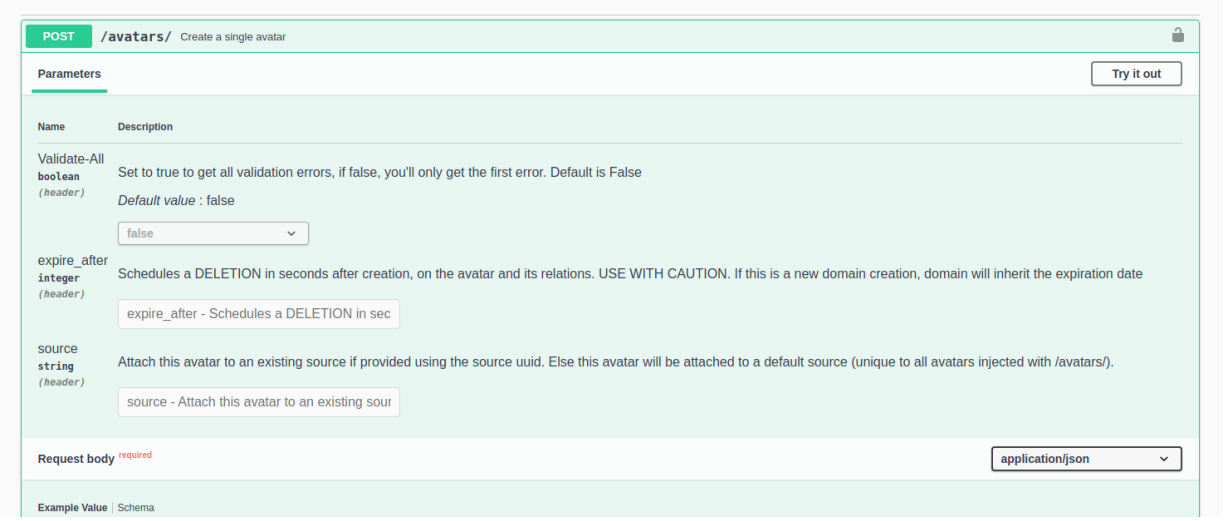


Figure 23 : L'API de Thing'In

Les scripts d'Achraf permettent d'utiliser facilement cette API depuis Python, et nous avons modifié ces derniers pour pouvoir d'une part importer certains d'objets (par exemple pour ensuite demander au service de vérification de présence d'objets de déterminer si ces objets

sont toujours au même endroit), d'autre part pour au contraire exporter les objets détectés par le nœud d'inventaire vers la plateforme Thing'In.

Faire le lien entre l'univers sémantique et le robot

Le lien entre la carte du robot est l'univers sémantique est réalisé à la main, en fonction de la situation.

Cas n°1: Nous considérons que la carte du robot est suffisamment précise pour correspondre à celle de Thing'In. Si nous utilisons directement les fichiers .yaml créé par les deux services, les coordonnées sont déjà exprimées en mètre dans le repère de la carte. Sinon, si nous voulons utiliser l'image de la carte, via le fichier de configuration .yaml de la carte, la résolution et l'origine de celle-ci nous permet de passer de coordonnées en pixels à des coordonnées réelles. Dans les deux cas, passer du repère de la carte à celui de Thing'In consiste juste à effectuer une translation et une rotation.

Cas n°2: Nous considérons que les déformations odométriques empêchent la correspondance entre les deux cartes d'un point de vue global, mais que les cartes peuvent correspondre localement.

Dans ce cas:

-Soit nous faisons une correspondance des cartes section par section, et chaque section a alors sa propre translation et rotation, qui sera appliquée à tous les objets se trouvant dedans.

-Soit nous faisons la correspondance objet par objet en utilisant des points de références proches. Nous fournissons à un script deux points de référence sur la carte du robot, les coordonnées des deux points correspondant sur Thing'In; et les coordonnées de l'objet dans le repère du robot pour que le script calcule sa position dans Thing'In.

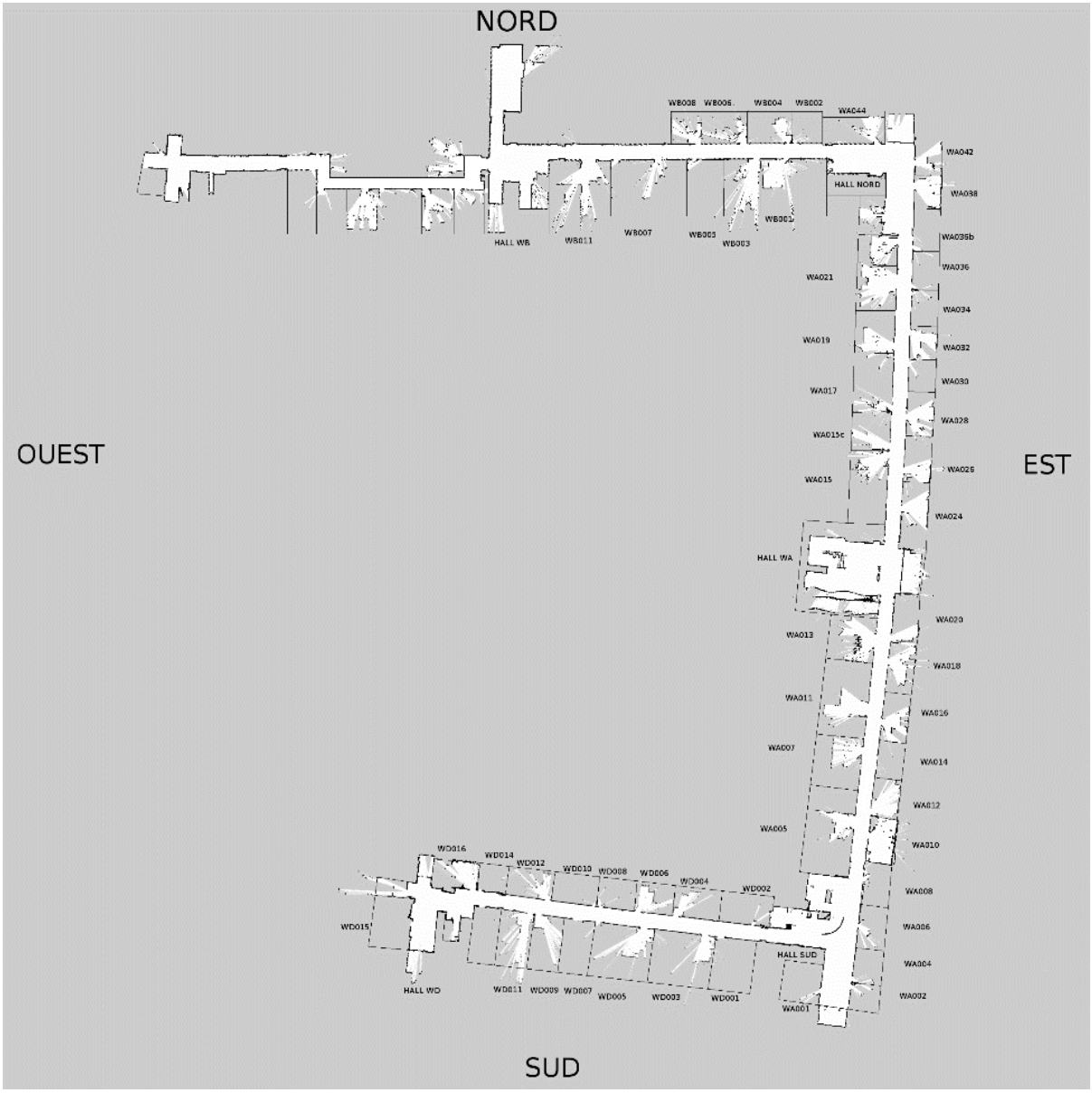


Figure 24 : Carte du bâtiment W créé avec le Geko et gmapping.

Les erreurs odométriques s'accumulent au fil du temps, et il n'est pas possible de faire correspondre la carte du robot et celle de Thing'In dans leur totalité.

Manipulations et expériences sur le terrain

Manipulation sur Turtlebot

En attendant de recevoir les pièces nécessaires à l'intégration de la jetson sur le geko, il fallait malgré tout tester les différents services en réel, afin de pouvoir vérifier qu'ils fonctionnent correctement même hors simulation. Pour cette raison, nous avons utilisé le robot Turtlebot mis à notre disposition, équipé d'une Raspberry Pi4 avec Ubuntu 18.04 (initialement, elle utilisait Raspbian, mais du fait de l'incompatibilité des packages avec Raspbian, nous avons décidé de la passer sur Ubuntu pour plus de simplicité).

Manipulation sur Turtlebot et caméra relié au PC

Afin de pouvoir utiliser darknet_ros_3d, nous avons besoin d'un nuage de points en entrée. Or, les nuages de points sont lourds; et ne peuvent pas être transmis fluidement via le wifi. Ainsi, lors de tests avec rosbot, le nuage de point envoyé directement par le robot à l'ordinateur était inutilisable, empêchant donc le bon fonctionnement de darknet_ros_3d (qui ne peut fonctionner que sur l'ordinateur ou la jetson, du fait de sa forte consommation de ressources de calcul).

Il n'est pas non plus possible de se passer d'un robot lors des test, nous avons besoin d'informations de localisation de la caméra afin de localiser les objets observés, et bien qu'il serait potentiellement possible d'essayer d'utiliser les données IMUs de la caméra pour la localiser, d'une part la localisation obtenue serait peu précise par rapport à la localisation obtenue par un robot et un nœud de navigation (la caméra ne peut utiliser que ses données IMUs, tandis qu'un robot dispose de données odométriques, de données de scans LIDAR, etc…), d'autre part il est bien plus simple d'utiliser tout simplement un robot pour ça.

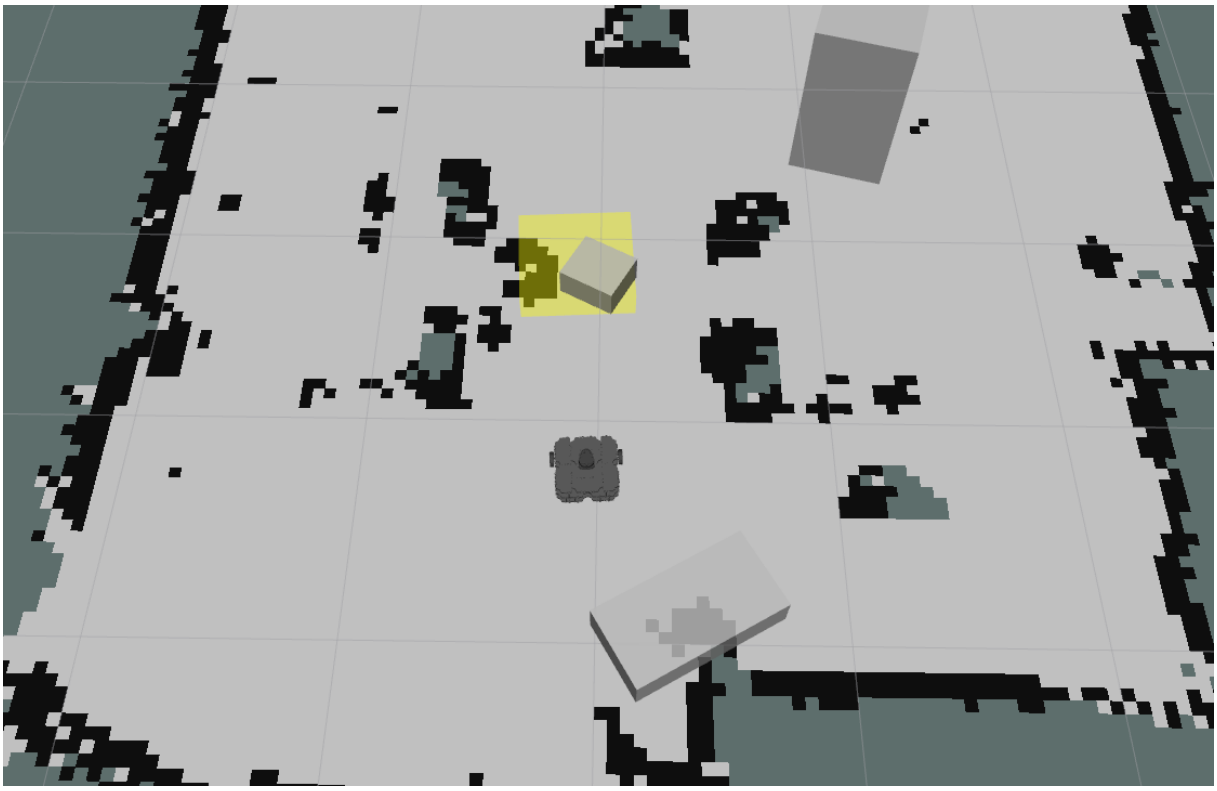


Figure 25 : Le nœud d'inventaire en action vu sur RVIZ

Cependant, comme il faut malgré tout faire des manipulations avec darknet_ros_3d, la méthode utilisée temporairement en attendant l'intégration de la jetson au geko consiste à utiliser la caméra de profondeur realsense branchée directement au PC, afin d'éviter de passer par le wifi, tout en la fixant sur le robot, qui enverra au PC les informations de localisation (laser scans, position et orientation du robot, et les cartes de l'environnement global où se trouve le robot). Le problème de cette méthode est que le mouvement du robot est ainsi limité à la longueur du câble, mais cela permet quand même de tester la localisation des objets via darknet_ros_3d.

Grâce aux informations de localisation du robot, et à darknet_ros_3d, nous pouvons alors connaître la position des objets détectés, leur taille, l'orientation de la caméra lors de leur observation, leur type et leur probabilité, ce qui nous permet de tester les deux services.

Manipulation avec caméra branché au Turtlebot.

Suite à une suggestion de Rémi Rigal, j'ai testé une séparation du nuage de point. Par défaut, il n'y a qu'un seul launch file sur la caméra, rs_rgbd.launch, qui permet de lancer la caméra, synchroniser et aligner les images de couleur et de profondeur, et d'assembler le nuage de points. Or, comme expliqué dans la section précédente, transmettre un nuage de

point n'est pas possible sur le Wi-Fi, ils sont beaucoup trop gros pour ça. Pour contourner ce problème, j'ai dissocié le launch file en deux parties, une qui lance la caméra et transmet les images de couleur et de profondeur, l'autre qui effectue l'alignement, synchronisation et l'assemblage du nuage de points.

Cette méthode une fois testée marche correctement, l'assemblage du nuage de point marche correctement et est utilisable par darkknet_ros_3d; mais est cependant assez fragile: observer les nuages de points ou les topics des images alignées & synchronisées provoque souvent une rupture du lien entre l'assemblage du nuage de point et les sorties de la caméra, et généralement il vaut mieux simplement éviter de toucher et d'observer le nœud d'assemblage. Dans le cas du geko, il n'y aura pas ce problème car le jetson et le geko seront en communication via des câbles Ethernet plutôt que via le Wi-Fi, ce qui rendra la communication plus robuste (ou alors, si ce n'est pas le cas et que le lien continue à être rompu souvent, nous réutiliserons directement *rs_rgbd.launch* et le nuage de point sera transmis via Ethernet).

Tests du nœud d'inventaire

Avant de commencer les tests sur la détection d'objets, nous avons effectués ceux sur le nœud d'inventaire, car nous réutiliserons l'inventaire généré par le service pour ensuite effectuer les tests de détection d'objets.

Les tests du nœud d'inventaires servent en premier lieu à ajuster les critères pour juger si un objet détecté par le robot est déjà présent dans l'inventaire ou non. Ces critères sont généralement empiriques, et dépendent de la taille des objets que nous ciblons.

Darknet_ros a du mal à détecter/localiser précisément les petits objets et ceux-ci subissent parfois d'importantes variations de volumes, couplées avec des erreurs de la localisation du robot; il faut donc ajuster les critères en conséquence pour qu'ils soient plus tolérant d'une variation de volume. En pratique, cela se traduit par le robot détectant parfois une même bouteille à 20cm de sa position originelle avec un volume deux fois plus élevé. Il est possible de rendre les critères encore plus tolérants, mais dans ce cas il ne sera plus possible de détecter deux bouteilles identiques à 20 cm l'une de l'autre par exemple.

Dans le cadre de plus grand objets, nous avons beaucoup moins d'erreurs de positions et de volume. Cependant, nous faisons néanmoins face à un autre type d'erreurs: il arrive que la caméra détecte un objet, même en le voyant partiellement. L'avantage, c'est que cela permet de répertorier quand même ces objets plutôt que de les ignorer, l'inconvénient, c'est qu'une fois un objet détecté partiellement, c'est le volume de la partie détectée et vue par l'image qui est enregistrée par le nœud d'inventaire, et non le volume complet de l'objet. Du fait de

comment marche darknet_ros, il n'est pas possible de savoir si nous voyons juste une partie de l'objet ou l'objet complet; et il faudra donc ajuster le résultat à la main.

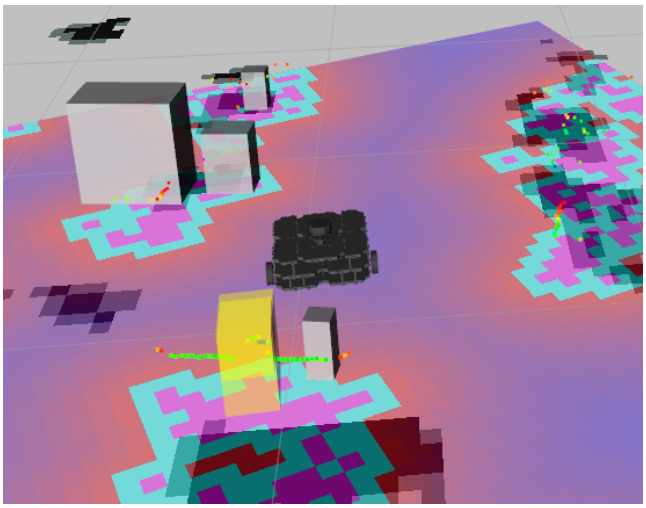


Figure 26 : Erreurs du nœud d'inventaire vu sur RVIZ

Dans cet exemple, le nœud d'inventaire a détecté deux fois la même bouteille à cause d'erreurs de localisation à la fois de l'objet et du robot. Le second enregistrement de la bouteille a à la fois un volume plus important et une position éloignée de plus de 20 cm. Rendre plus stricts ces paramètres risque d'empêcher de répertorier différents objets du même types proches l'un de l'autre

Par conséquent, dans le cadre de l'utilisation du nœud d'inventaire, je suggère de toujours revérifier à la main les résultats du nœud, d'une part en vérifiant les images capturées à chaque ajout d'un objet dans l'inventaire (c'est à ça qu'elles servent), et d'autre part en affichant les objets sur la carte (soit via le *marker_array/darknet_map* publié par le nœud d'inventaire pendant qu'il est en marche, soit en utilisant *generate_waypoint* une fois le service d'inventaire éteint). Si nous avons des petits objets, il faut vérifier l'absence de faux positifs, si nous avons des grands objets, il faut vérifier qu'ils sont bien détectés en totalité.

Tests de la génération de waypoints et du nœud de patrouille

Une fois les tests sur le nœud d'inventaire terminés et les résultats obtenus ajustés, nous effectuons la génération des waypoints pour le nœud de patrouille. Bien que celle-ci soit automatisée, il faut l'ajuster en fonction de la situation. Par exemple, dans le cas des tests du Turtlebot, ceux-ci se sont effectués dans des passages assez étroits, et il a donc été nécessaire de diminuer les distances minimales pour les observations d'objets ainsi que le nombre d'itération de l'érosion. Au contraire, dans le cas où le robot se déplace au sein de couloirs larges et où les objets se trouvent dans des pièces spacieuses, aucun ajustement n'est nécessaire.

Une fois les waypoints générés, le déplacement du robot dépend uniquement de l'algorithme de navigation utilisé, le nœud de patrouille se contentant de lui envoyer les waypoints à atteindre en fonction des objets que veut voir l'utilisateur.

Expérimentations sur Geko

Une fois les expérimentations sur le Turtlebot terminée, nous avons commencé les expérimentations sur le geko. Pour cela, nous avons simplement équipé ce dernier de la caméra realsense, le reste du robot étant déjà fonctionnel. Nous avons fait des tests sans intégration de la Jetson, car intégrer la jetson nécessiterai d'ouvrir le geko, ce dont seul Thibaud et Jacques sont capables de faire (l'agencement et le câblage interne font qu'il est très simple de provoquer des dégâts matériels au robot sans expérience). L'image de la jetson a donc été transmise via le Wi-Fi, et le nuage de point n'est pas transmis par le wifi (étant beaucoup trop lourd pour ça) mais généré directement sur la jetson depuis les images fournies par la caméra.

La majorité des algorithmes se trouvant sur la Jetson et tournant dessus, il y a peu de différences fonctionnelles entre le geko et le Turtlebot. Dans le cas des deux robots, ils servent à fournir leur positions et à se déplacer via le nœud de navigation, ainsi qu'à transmettre les images de la caméra. Le reste des algorithmes (assemblage du nuage de points, darknet_ros, darknet_ros_3d, nœuds d'inventaire/de vérification de présence d'objets) fonctionne sans problème sur la jetson.

Néanmoins, parmi les différences constatées entre le geko et le Turtlebot:

-La position de la caméra est plus élevée sur le geko, ce qui permet de voir la majorité des grands objets dans leur totalité, là où le Turtlebot ne voyait que leur moitié inférieure dans la majorité des cas. Néanmoins, il faut malgré tout maintenir une distance d'1m50 ou de 2m entre le geko et l'objet pour observer correctement ce dernier dans son intégralité.

-De plus, du fait de cette distance minimale entre le geko et l'objet, darknet_ros a beaucoup plus de mal à détecter les petits objets, la confiance de la détection de ces derniers est plus faible, et leur positions et volumes sont aussi estimés de manière moins précise.

Résultats des expérimentations:

Que ce soit sur le geko ou sur le Turtlebot, dans les deux cas les fonctions de déplacement des algorithmes fonctionnent correctement via l'usage des nœuds de navigations natifs aux deux robots. Les callflows marchent aussi sans problème, chaque service démarrant ou s'arrêtant aux bons moments sans problème de délais. Les scripts utilitaires (génération de waypoints, sauvegarde d'image) sont aussi fonctionnels, et n'ont pas de problème, il faut cependant ajuster le script de génération de waypoint à la main afin de s'assurer qu'il soit

adapté à la situation et vérifier ses résultats via la visualisation de la carte et des waypoints générés par le script. Néanmoins, certaines parties de l'algorithme ont des problèmes, qu'il n'est pas toujours possible de corriger:

-L'assemblage du nuage de point rencontre de temps en temps des problèmes. Lorsque cela arrive, darknet_ros_3d ne reçoit plus de nuage de points et ne renvoie plus les objets détectés, ce qui rend la détection d'objets non-opérationnelle dans le cadre des deux services. Il convient donc de surveiller via RVIZ les sorties de darknet_ros_3d et celle de darknet_ros via rqt/image_view, si darknet_ros détecte des objets et que darknet_ros_36d ne revoit rien, soit darknet_ros_3d a été mal configuré (par exemple, la catégorie de l'objet détecté n'est pas renvoyée par darknet_ros_3d), soit il y a un problème avec le nuage de points. Dans mon expérience, une fois que l'assemblage du nuage de point a eu un problème une fois, il n'en a plus une fois qu'il est redémarré à nouveau.

-Toujours dans le cadre des petits objets, leur volume et dimensions ont tendance à varier selon l'angle de vue et la distance de la caméra.

-De temps en temps, lorsque le robot tourne, l'image est floue, mais darknet_ros essaie quand même d’analyser des objets dessus, et détecte de temps en temps des faux positifs.

-En fonction de l'objet, darknet_ros a plus ou moins du mal à le détecter. Par exemple, certains modèles de chaises sont détectés sans problème par darknet_ros, tandis que l'algorithme est incapable de reconnaitre d’autres modèles, détectant à la place un feu tricolore par exemple.

-Le cas des objets partiellement visible est problématique. En effet, d'une part, le volume et les dimensions associées à l'objet sont alors incorrects, d'autre part, cela nuit à la qualité de la détection et facilite les faux positifs de manière générale. Un morceau de bureau vu à travers une porte risque donc d'être détecté comme une chaise ou un autre objet, darknet_ros ne disposant pas d'un contexte suffisant pour identifier correctement l'objet.

Les deux derniers problèmes sont propres au modèle de reconnaissance yolo-v2-tiny, et nous ne pouvons pas les corriger. Les seules choses que nous pouvons faire pour lutter contre eux sont soit d'être plus exigeant sur les critères de confiance requis pour prendre en compte un objet, au risque d'ignorer des objets correctement détectés ; soit d’utiliser un autre modèle de reconnaissance (yolo-v3), c’est cette dernière option que nous retiendrons.

Palliatifs

Erreurs liée aux mouvements

Afin de corriger les erreurs de détections en mouvement ou en rotation lors de l'utilisation du service d'inventaire; bien qu'il ne soit pas possible de mettre en pause darknet_ros, nous pouvons néanmoins empêcher l'enregistrement d'objets lors du mouvement. Pour cela, nous avons modifié le code du service d'inventaire pour ne pas garder en mémoire les objets par défaut. Lorsque le robot est immobile, après un temps d'attente défini par l'utilisateur, si le robot est bien resté immobile pendant ce temps (n'importe quel déplacement réinitialise l'attente) le nœud deplacement_inventaire.py envoie au nœud de référencement des objets (darknet_inventaire_node.py) un signal lui disant qu'il peut garder en mémoire les objets qu'il détecte. Quand le robot est en déplacement, le nœud de déplacement envoie alors un signal indiquant d'arrêter référencer les objets. Les erreurs de darknet_ros liées au mouvement ou à la latence entre déplacement et image ne sont alors plus enregistrées, permettant ainsi de contourner ce problème.

Nous avons au passage amélioré le nœud deplacement_inventaire.py en lui rajoutant trois modes de mouvements: soit un suivi de waypoint prédéterminés, soit une entrée manuelle de waypoint de la part de l'utilisateur, soit un mode de déplacement libre (permettant d'envoyer des commandes de déplacement indépendamment du service). Quant au cas du nœud de détection d'objet, les erreurs de positions des objets liées au mouvement n'ont pas d'importance, n'étant pas enregistrées, il suffit juste d'attendre un peu devant l'objet pour que nous soyons assuré que l'objet en face du robot est bien détecté à la bonne position.

Erreurs liées à la précision

Concernant la précision de manière générale, étant donné que le robot n'enregistre désormais la reconnaissance d'objets qu'à l'arrêt, la fluidité et la vitesse de darknet_ros a moins d'importance, et nous sommes passé du modèle yolo-v2-tiny conçu pour tourner en temps réel de manière très fluide (100-120 images par secondes) au modèle yolo-v3 (20-30 images par secondes). Ce dernier s'est relevé être bien plus précis, détectant sans problème des objets que yolo-v2-tiny avait du mal à reconnaître, ce modèle est même capable d'atteindre une confiance de 100% par moments. Ce modèle permet donc d'utiliser des critères de confiance plus élevés, mais ne permet pas d'obtenir une position ou un volume plus précis, le seul moyen de garantir la précision de la localisation d'un objet reste donc d'attendre à l'arrêt.

Annexes

Tables des figures

Sources

[1] Mathieu Labbé · François Michaud: *Long-Term Online Multi-Session Graph-Based SPLAM with Memory Management:* https://introlab.3it.usherbrooke.ca/mediawiki-introlab/images/8/87/LabbeAURO2017.pdf

[2] Labbé M. & Michaud F. (2018): *RTAB-Map as an Open-Source Lidar and Visual SLAM Library for Large-Scale and Long-Term Online Operation:* https://introlab.3it.usherbrooke.ca/mediawiki-introlab/images/7/7a/Labbe18JFR_preprint.pdf

Labbé M. - IntRoLab - Universite de Sherbrooke (2010-2016): *Parameters.h - RTAB-Map library:* https://github.com/introlab/rtabmap/blob/master/corelib/include/rtabmap/core/Parameters.h

[4] Mohammad Omar Salameh, Azizi Abdullah, Shahnorbanun Sahran: *Ensemble of Bayesian Filters for Loop Closure Detection:* https://www.jaist.ac.jp/jaist-sast2015/images/pdf/IS_EntertainmentTechnology_Azizi-ABDULLAH_Ensemble-of-Bayesian-Filters-for-Loop-Closure-Detection.pdf

[5] Redmon J., Divvala S., Girshick R. & Farhadi A. (2016): *You Only Look Once: Unified, Real-Time Object Detection:* https://pjreddie.com/media/files/papers/yolo_1.pdf

[6] Joseph Redmon, Ali Farhadi (2017): *YOLO9000: Better, Faster, Stronger:* https://pjreddie.com/media/files/papers/YOLO9000.pdf

[7] Joseph Redmon, Ali Farhadi (2018): *YOLOv3: An Incremental Improvement:* https://pjreddie.com/media/files/papers/YOLOv3.pdf

Schémas

Nœud et services ros lors du fonctionnement du nœud d'inventaire

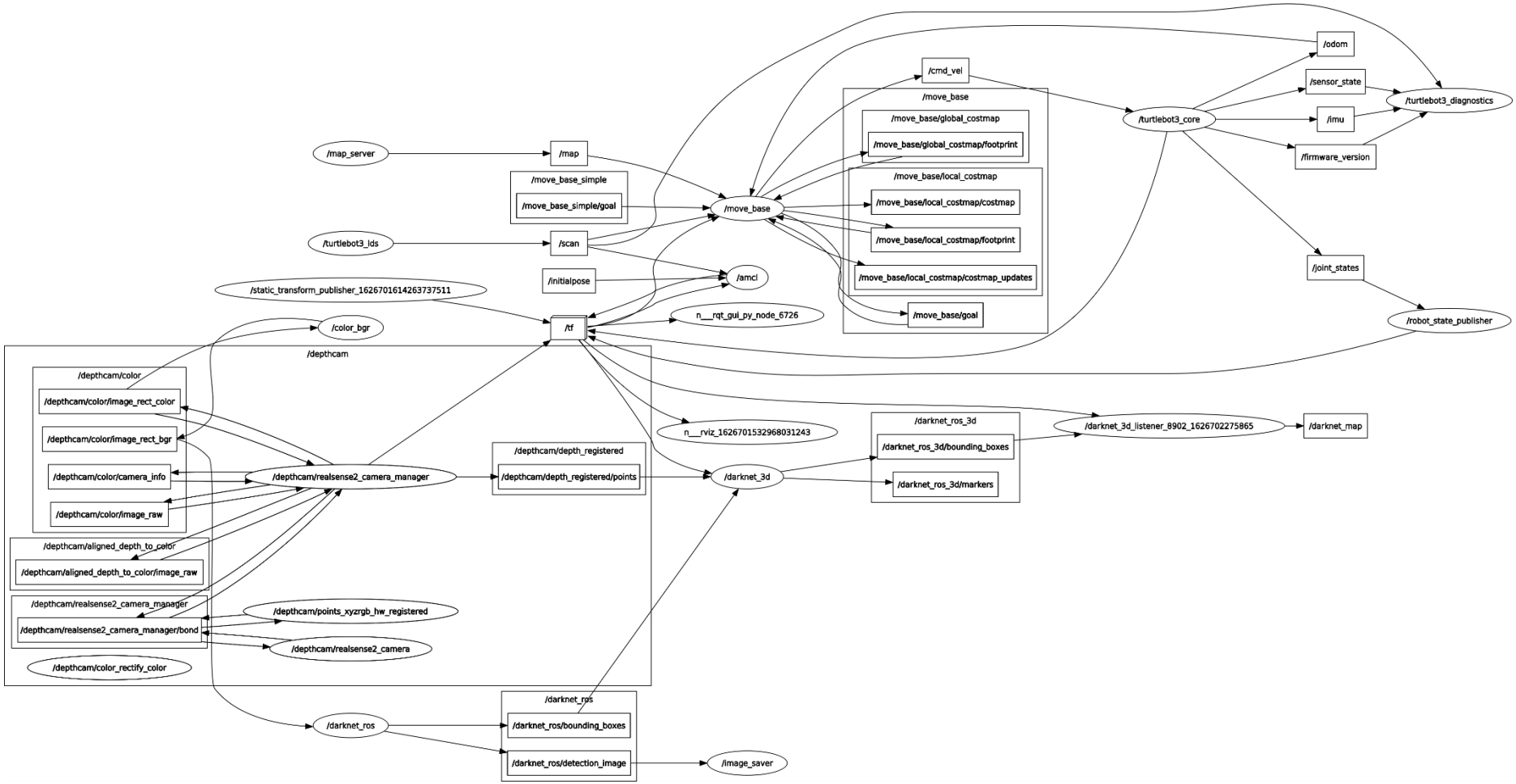


Schéma du service d'inventaire

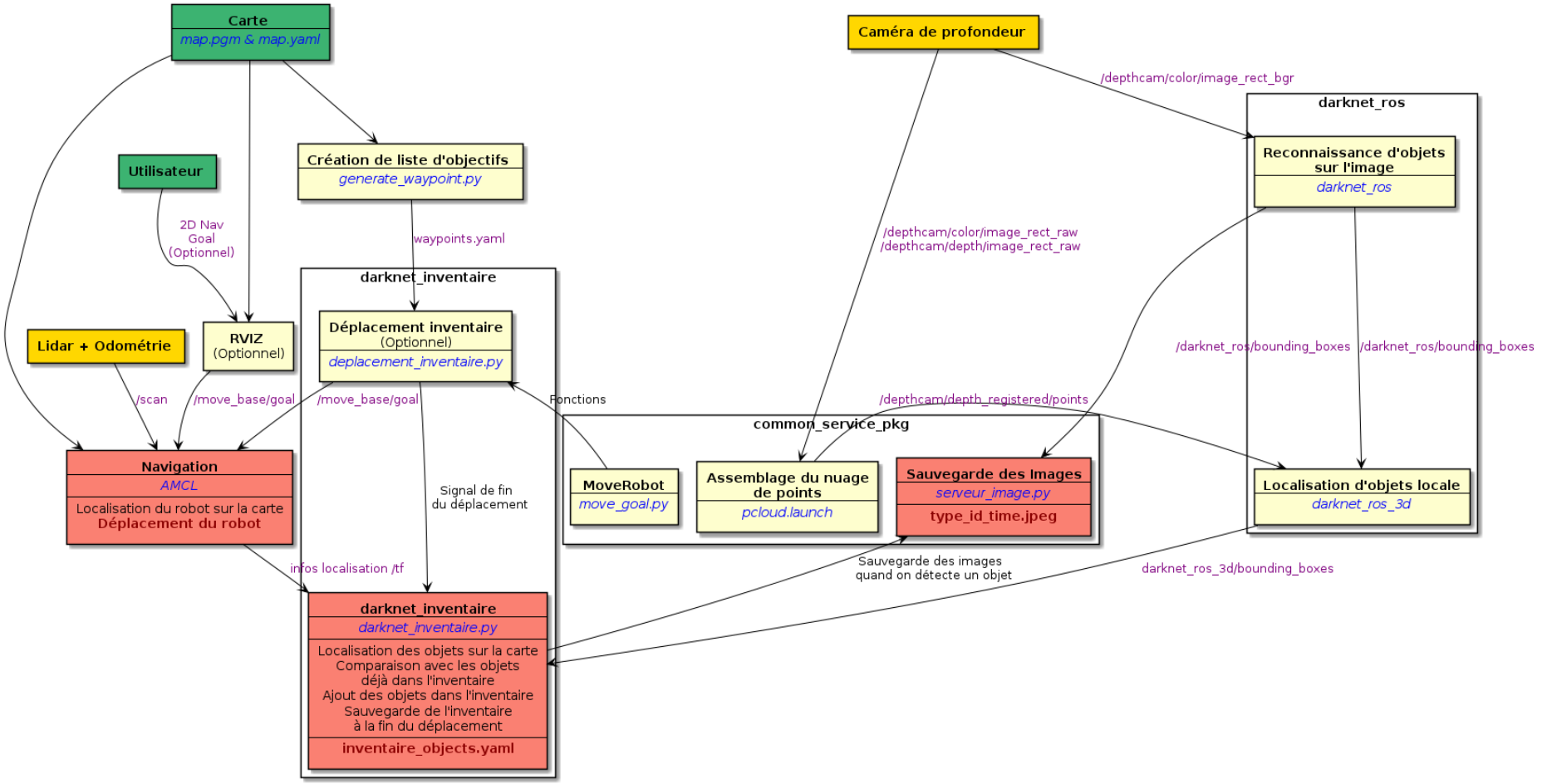
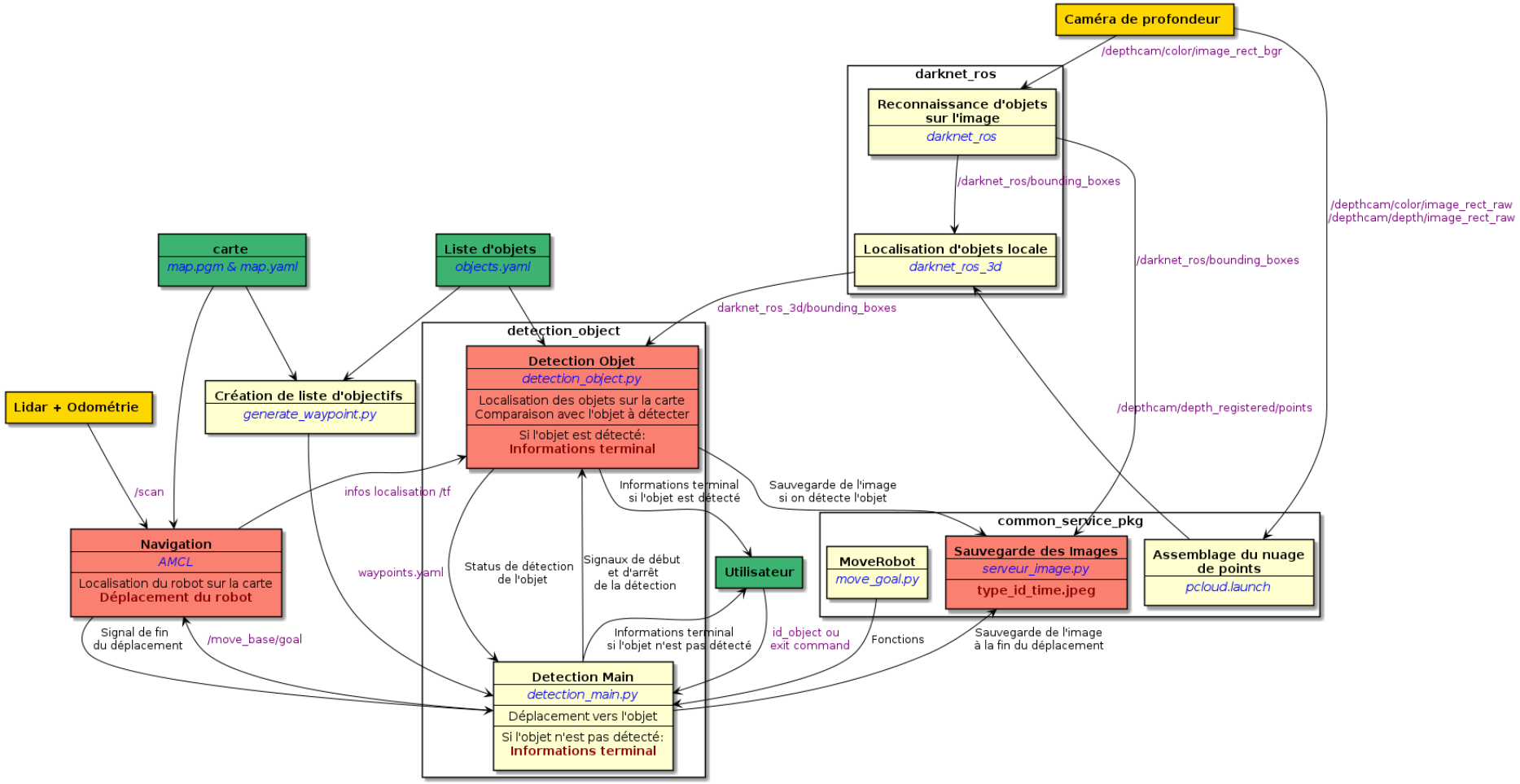


Schéma du service de détection



Call flow du service de vérification d'objets