Rapport de stage : Projet Hermes-Vocal

Rémi Dulong - Louis-Baptiste Trailin
Tuteur : Mostafa Bellafkih

Juin - Septembre 2017

Résumé

Ce rapport est le résultat d'un stage de 3 mois à L'institut National des Postes et Télécom de Rabat, dont l'objectif était de réaliser un programme d'assistant vocal personnalisable, dans le but de l'intégrer à un robot. Nous avons donc commencé par faire un état de l'art de ce genre de technologies, puis nous avons mis en pratique ce que nous avons appris afin d'aboutir à un programme répondant à nos contraintes. La totalité de notre travail a été pensée pour fonctionner sur des distributions GNU/Linux et sur Android.

Table des matières

Chapitre 1

Objectif du stage

1.1 Le projet Hermes

Le stage que nous avons réalisé s'inscrit dans le cadre du développement d'un robot qu'une équipe de 6 étudiants de Télécom SudParis a conçu lors de l'année scolaire 2016-2017 dans le cadre des projets Cassiopée. Il s'agit d'un robot roulant qui aurait pour objectif de guider les personnes qui en ont besoin au sein du campus de Télécom SudParis. Pour cela, le robot Hermes doit être capable de se déplacer mais aussi de communiquer avec l'utilisateur ayant besoin d'être guidé. Nous avons déjà réalisé (au moins partiellement) la partie de ce projet qui consistait à créer le robot et assurer ses déplacements. Cependant, il restait à trouver le meilleur moyen de communication entre le robot et ses utilisateurs : c'est pour cela que nous avons pensé à un système de communication orale, sans encore savoir, à ce moment là, comment nous allions y parvenir. Ce robot est équippé d'un ordinateur central embarqué qui peut être une Raspberry Pi ou une Beaglebone Black fonctionnant sur une distribution Linux minimaliste customisée. Il embarque également une tablette sous Android afin d'assurer l'interaction avec les utilisateurs.

1.2 Sujet du stage

Après réflexion avec notre tuteur de stage, nous avons abouti au sujet de stage suivant :

Dans le contexte plus vaste de l'étude d'un robot déambulatoire pour Télécom SudParis, on souhaite doter celui-ci d'éléments d'interaction homme-machine permettant la reconnaissance et la compréhension de requêtes parlées ou écrites d'une part, et la synthèse de réponses d'autre part. L'étude d'un tel système s'appuiera sur une étude approfondie de l'état de l'art du domaine, cette étape permettant d'orienter la phase de conception. Un des aspects essentiels est la conception d'un modèle pouvant servir tant à représenter les requêtes qu'à élaborer les réponses. L'étude s'attachera à fournir un prototype implémentant et validant ces concepts. L'intégration de ce système au sein du robot déambulatoire fera en outre l'objet d'une étude particulière, compte tenu des limitations des calculateurs embarqués. Ainsi il sera éventualisé la délocalisation du système sur un serveur distant.

Chapitre 2

Etat de l'art

2.1 Les assistants vocaux existants

Aujourd'hui, il existe deux grands types d'assistants vocaux : les projets propriétaires et les projets Open Source. Voici les exemples les plus pertinents que nous avons étudié :

2.1.1 Siri, OK Google, Cortana...

Récemment, nous avons vu apparaître plusieurs assistants vocaux créés par les géants que sont Apple, Google et Microsoft. Ces assistants vocaux peuvent s'avérer très pratiques, et nous montrent bien que les technologies de compréhension et de synthèse vocale sont assez développées pour l'utilisation que nous souhaitons. En effet, après quelques tests, on remarque que la qualité de la transcription de la parole en texte (Speech To Text) est très satisfaisante, ce qui est indispensable pour créer un générateur de réponses fiable. Cependant, nous ne pouvions utiliser aucun de ces assistants existants, parce qu'ils ne sont pas du tout personnalisables, nécessitent une connexion internet, et ne peuvent pas réellement être utilisées pour un système embarqué. Ils restent néanmoins des preuves du fonctionnement de ce genre de technologies, et nous ont inspiré pour la création de notre propre système.

2.1.2 Mycroft

Après quelques recherches sur les assistants existants, nous sommes tombés sur le projet Mycroft. Il s'agit d'un projet d'assistant Open Source, prévu pour fonctionner sur Raspberry Pi (donc parfaitement adapté à un système embarqué). L'assistant est complètement personnalisable, et les développeurs encouragent les contributeurs qui peuvent ajouter des capacités (appelées "Skills") à l'assistant, c'est à dire des capacités à répondre à des questions. De plus, la communauté francophone est encore assez peu représentée dans ce projet, il y a donc de nombreux Skills basiques de Mycroft qui ne demandent qu'à être traduits pour fonctionner également en Français. Nous avons passé un bon moment à étudier le fonctionnement de Mycroft, à essayer de tester sa modularité et sa personnalisabilité. Nous avons notamment essayé de l'utiliser en français, en changeant les paramètres des programmes de compréhension de parole (STT) et

de synthèse vocale (TTS). Ce projet est très prometteur, et nous aurions pu l'utiliser comme base pour développer notre système, en ajoutant tout simplement les "Skills" nécessaires pour envoyer des ordres au robot Hermes. Cependant, nous avons choisi de procéder différemment, comme nous l'expliquerons dans la chapitre 3

2.2 Compréhension et synthèse vocale (STT et TTS)

Dans cette partie, nous parlerons des programmes de STT et de TTS. Le STT (Speech To Text) est un programme capable de convertir un enregistrement de voix en une chaîne de caractères afin de permettre son interprétation. Ces logiciels utilisent des dictionnaires de mots connus et leur décomposition en phonèmes (ce qui revient à de l'étude de syllabes) correspondante. Le fichier sonore est traité afin de décomposer le fichier en phonèmes selon un modèle, et cherche la correspondance la plus probable dans le dictionnaire. Ce programme peut utiliser des réseaux de neurones. Le TTS (Text To Speech) fait le travail inverse : il permet de faire de la synthèse vocale, afin de répondre à l'utilisateur à l'oral. Le programme génère un fichier de son à partir d'une chaîne de caractères. Il utilise pour cela un modèle (réalisé à partir de chaînes de Markov et de réseaux de neurones) afin d'associer la bonne prononciation à chaque mot. Cependant, le dictionnaire, le modèle de décomposition en phonèmes et le modèle de synthèse vocale dépendent de la langue que l'on utilise. Et malheureusement pour nous les francophones, les modèles les plus aboutis sont en Anglais. Les modèles de compréhension du français le sont beaucoup moins, à l'exception de certaines API propriétaires.

2.2.1 Les API en ligne

Il existe plusieurs API en ligne très performantes. C'est le cas de l'API de Google, mais également de celle de Mycroft. Leur défaut principal est de nécessiter une connexion internet permanente pour fonctionner, ce qui est un problème pour notre système. En effet, notre robot aura normalement une connexion Internet via Wi-Fi dans le Forum (l'Accueil de Télécom SudParis) mais pourrait potentiellement sortir de cet endroit pour aller dans d'autres zones du campus. Il faudrait, dans ce cas, que la reconaissance vocale fonctionne également. Nous devions donc choisir un système pouvant fonctionner hors ligne.

2.2.2 Les programmes hors-ligne

Nous avons donc cherché des programmes équivalents fonctionnant sans connexion Internet. Nous nous sommes donc penchés sur les deux logiciels que sont Espeak et PocketSphinx. Espeak est un TTS Open Source, qui permet donc d'utiliser une voix synthétisée à partir d'une phrase qu'on lui fournit. Si ses résultats sont indéniables en Anglais, il faut avouer qu'en Français, la voix générée par défaut est si "robotique" que l'on peine à la comprendre. Il existe un logiciel complémentaire appelé MBrola qui permet d'utiliser d'autres voix plus réalistes, mais cela reste assez peu intelligible. PocketSphinx est un

STT Open Source également, qui permet donc de générer du texte à partir du son d'une voix. Après avoir fait quelques tests, en installant péniblement les modèles et dictionnaires français, nous en sommes arrivé à la conclusion que ce programme, relativement efficace en Anglais, est très complexe à utiliser en Français encore aujourd'hui. Le programme n'arrêtait pas de reconnaître des mots que nous n'avions pas prononcé, ce qui aurait été très difficile à interpreter par la suite. La seule solution restante était d'utiliser l'application de Google intégrée dans Android, qui permet d'utiliser la reconnaissance et la synthèse vocale sur un téléphone ou une tablette. La contrainte d'intégrer une tablette au robot n'étant pas un problème (puisqu'il était prévu d'en placer une dans tous les cas). C'est un programme quasiment aussi précis dans la reconnaissance que l'API en ligne de Google, et avec la même synthèse vocale (elle utilise la même voix que pour Google Traduction). Le seul inconvénient étant que le programme sur tablette doit communiquer avec l'ordinateur central du robot, puisqu'on ne pourra pas gérer la reconnaissance et la synthèse vocale directement au niveau de cet ordinateur central.

2.3 Compréhension du langage et génération de réponses

En ce qui concerne la compréhension des requêtes et la génération de réponses associées, nous n'avons eu accès qu'au code de Mycroft, puisqu'il est le seul des projets précédents à être Open Source. Ainsi, nous avons pu analyser que Mycroft fonctionne de manière modulaire, avec des "Skills" qui sont tout simplement des correspondances entre une requête et sa réponse, ou le code qui doit être executé. Il s'agit d'un modèle dit "rule-based", en opposition avec les modèles plus complexes qui utilisent notamment des réseaux de neurones. Cela signifie tout simplement que l'on utilise une liste de requêtes connues et de leur réponse associée pour comparer chaque requête à ce que l'on connaît afin de vérifier si on peut répondre.
Il est très probable que les modèles utilisés par Google, Cortana et Siri soient plutôt des modèles basés sur des réseaux de neurones permettant la compréhension du langage naturel. Il s'agit de la technique la plus avancée afin d'interpréter les requêtes, car elle permet de répondre à des requêtes qui ne sont pas définies de manière absolue. Une légère modification dans la requête peut donner le bon résultat, ce qui donne une bien plus grande flexibilité au système. Cependant, cette technique demande de réaliser et d'entrainer des réseaux de neurones complexes, ce qui demande à la fois une grande quantité de calculs, une base de données de requêtes et de réponses associées énorme afin de faire l'apprentissage du réseau de neurones, et la maîtrise des concepts de réseaux de neurones que nous n'avons pas pour l'instant.

Chapitre 3

Réalisation finale

Après avoir fait un état de l'art des assistants vocaux proche de ce que nous souhaitions, nous avons pu nous consacrer à la réalisation de notre propre programme, adapté au mieux à nos besoins. Nous avons donc choisi les technologies suivantes :

3.1 STT et TTS

Tout d'abord, la seule solution potentiellement utilisable dans notre système semblait être PocketSphinx. Cependant, au vu des résultats que nous obtenions en l'utilisant en Français, nous avons choisi d'utiliser l'API hors ligne de Google installée nativement sur toutes les tablettes et téléphones Android.
Ce choix a donc conditionné l'architecture de notre système, qui a dû être pensé en Client - Serveur. Le client sera une application Android, et le serveur un programme en python. Les deux communiqueront à travers le réseau via un socket.

3.2 Compréhension et génération de réponses

N'ayant que trop peu de connaissances sur les réseaux de neurones, et un cahier des charges pouvant être respecté grâce à un modèle "rule-based", nous avons réalisé notre programme Serveur en suivant une architecture proche de celle de Mycroft, codé en Python 3.

3.2.1 Les "Skills"

Afin de rendre l'assistant très modulaire et ainsi simplifier la gestion des phrases auxquelles il sait répondre, nous avons gardé le concept de "Skills" du projet Mycroft. Un Skill est tout simplement un morceau de code que l'on peut importer dans le coeur du programme serveur, afin d'ajouter des phrases au dictionnaire de requêtes connues et leur associer une fonction de réponse.
Notre programme étant orienté objet, nous avons créé des classes possédant les spécifications suivantes :

La classe Skill

Les attributs :

— **keyPhrases** : la liste des phrases exactes connues. Par exemple, ["Quelle heure est-il ?", "Il est quelle heure ?"]
— **superWords** : Liste de mots ou motifs de mots clés permettant d'interpréter une phrase qui n'est pas dans keyPhrases. Dans notre exemple, ["heure", "quelle heure"]
— **badWords** : Liste "anti mots clés", contient des mots ou motifs qui nous montrent qu'il ne faut pas interpréter la phrase. Dans notre exemple, ["à quelle heure"]
— **result** : C'est la fonction à exécuter si la requête est confirmée comme étant un appel à ce skill. Dans notre exemple, il s'agit d'une fonction qui demandera l'heure au système d'exploitation, et la renverra sous la forme d'une chaîne de caractères telle que "Il est XX heures et XX minutes".
— **keyWords** *(attribut dérivé)* : Il s'agit d'un dictionnaire des mots contenus dans les keyPhrases. il est généré à la construction de l'objet en décomposant chaque keyPhrase. Pour notre exemple, il vaudra ["quelle", "heure", "est", "il"] (on retire les mots en doublon)

Les méthodes :

— **ask** : Méthode qui prend en argument la requête, et qui vérifie si la question posée par l'utilisateur est dans les keyPhrases. Renvoie un booléen.
— **similitude** : Méthode qui prend en argument la requête, et qui attribue à la requête un score en fonction de la "similitude" avec les keyPhrases, la présence de superWords ou de badWords. Les paramètres du calcul sont modifiables, par exemple, on a choisi d'attribuer +1 au score pour chaque keyWord présent dans la requête, +20 pour un superWord, et -40 pour un badWord. La fonction renvoie dont un entier représentant le score de similitude.
— **execute** : Méthode qui execute le Skill lorsqu'il est confirmé (c'est à dire qu'il appelle la fonction fournie pour result)

Comme nous l'avons vu avec l'exemple de la demande d'heure, la classe Skill nous permet de définir une capacité générique pour l'assistant vocal. Elle permet une simple association requête → fonction à éxécuter. C'est donc le modèle de Skill que nous utilisons pour la plupart des fonctionnalités ne demandant pas un traitement plus complexe. Cependant, nous avons jugé intéressant de créer deux classes filles afin de s'adapter aux fonctionnalités plus spécifiques. Il s'agit des classes TextSkill et ArgSkill.

La classe TextSkill

De nombreuses requêtes ont pour réponse une simple phrase. Par exemple, si l'on demande au robot *"qui es tu ?"* il pourra tout simplement répondre par une phrase connue à l'avance : *"Je suis le robot Hermes"*. Nous avons donc créé une classe spécifique pour ce type de requêtes.
Pour rendre l'interaction avec le robot plus authentique, nous avons souhaité que pour chaque question de l'utilisateur, le robot possède plusieurs réponses formulées différemment. Une de ces réponses est choisie aléatoirement avant

d'être renvoyée.
La différence entre un Skill et un TextSkill est que la fonction *return* est déjà connue pour un TextSkill : il s'agit du retour aléatoire d'un élément d'une liste de réponses connues. Cependant, si il ne faut plus fournir une fonction *return* dans le constructeur d'un TextSkill, il faut tout de même lui fournir la liste des réponses que l'on souhaite affecter à la requête.

La classe ArgSkill

Cette classe a été créée pour les Skills les plus complexes. En effet, il fallait que l'assistant puisse comprendre une requête contenant un "argument", comme par exemple la phrase *"Guide moi au bâtiment B"*. Pour cela, il doit être capable de décomposer la phrase en deux partie : la requête (*"Guide moi à"*) et l'argument (*"bâtiment B"*). Puis, il doit appeler le Skill correspondant à la requête *"Guide moi à"*, en lui fournissant l'argument du lieu où l'on doit guider, afin de pouvoir envoyer une requête de Path Finding vers l'endroit demandé au programme responsable du déplacement du robot.
Ainsi, la différence fondamentale entre le Skill et le ArgSkill, est qu'un ArgSkill possède une fonction *return* qui demande qu'on lui fournisse la requête de l'utilisateur afin de pouvoir en extraire l'argument et l'utiliser. De plus, un ArgSkill possède une méthode *ask* modifiée : en effet, on ne vérifie pas que la requête corresponde parfaitement à une keyPhrase connue, mais on vérifie à la place que la keyPhrase connue (par exemple *"Guide moi à"*) est **inclue** dans la requête. Sans cela, nous ne pourrions jamais identifier précisément les requêtes, puisque l'argument n'est pas connu à l'avance.

3.2.2 Le package core

Le package core contient l'essentiel de ce qui est à importer pour pouvoir utiliser le coeur du système. En effet, importer le fichier core.py suffit à créer tous les objets Skills, et il contient la fonction principale du programme que nous avons appelé *executeOrder(order)*. Cette fonction demande l'ordre envoyé en argument, et se charge d'interroger les Skills, afin de trouver celui qui correspond le mieux à la requête faite par l'utilisateur, avant de l'éxécuter. Bien entendu, l'interrogation des différents Skills se fait d'une manière précise :

Première vérification

Dès que la fonction *executeOrder* est appelée, le programme récupère la requête de l'utilisateur. Il va commencer par la "première vérification". Pour limiter au maximum la complexité (et donc le temps de réponse) du programme, la première vérification consiste à utiliser la méthode *ask* sur chacun des Skills instanciés. On vérifie donc si la phrase prononcée par l'utilisateur fait partie des phrases connues (il faut que la requête soit **exactement** une keyPhrase).
Si la requête est trouvée parmi les keyPhrases d'un Skill, alors on lance l'éxécution de ce Skill. Cependant, si la phrase n'est pas connue au mot près, on passe à la seconde vérification, plus fine mais qui demande plus de calculs.

Seconde vérification

Si la première vérification a échouée, on doit procéder à une analyse plus

fine de la requête de l'utilisateur. C'est à ce moment qu'intervient la méthode *similitude* des Skills. En effet, le programme va demander à chaque Skill son "score de ressemblance" avec la requête. Cela revient en somme à un calcul de la probablilité que la requête corresponde bien au Skill interrogé. Une fois que chaque Skill a calculé son score, l'algorithme repère le Skill qui a obtenu le meilleur score. Cependant, nous ne pouvons pas dire que ce Skill correspond bien à ce qui a été demandé par l'utilisateur ! Par exemple, si le meilleur score est de 1 (et donc que tous les autres sont de 0) cela signifiera qu'un mot banal (par exemple *"il"*) était commun entre la requête et une des phrases connues dans le Skill. Cela ne permet en aucun cas de dire que la question de l'utilisateur était *"quelle heure est il ?"*, parce que la le score est trop faible. Nous avons donc fixé arbitrairement un seuil, un score minimum, à partir duquel il est possible d'envisager que le Skill ayant le plus grand score corresponde bien ce que l'utilisateur a demandé. Cependant, si deux Skills obtiennent un score très élevé, mais trop proches l'un de l'autre, on supposera que l'on ne peut pas distinguer lequel des deux est le plus probable. Nous avons donc choisi arbitrairement un second seuil correspondant à l'écart de score minimal à respecter entre le Skill de score maximal et le second afin de valider le premier. En résumé, pour qu'un Skill soit validé à la seconde vérification, il faut :

- Qu'il soit celui qui obtient le score maximal
- Que son score soit au dessus d'un seuil minimal
- Que l'écart entre son score et le second plus haut score dépasse un certain seuil

Si ces trois conditions sont validées, alors l'algorithme déterminera qu'il est très probable que la requête de l'utilisateur corresponde à ce à quoi le Skill peut répondre, et lancera son éxécution. Nous avons essayé différentes valeurs pour les seuils arbitraires ainsi que pour les points que rapportent les mots clés et anti mots clés, afin de trouver un compromis intéressant qui est capable de répondre à une requête dont la forme est inconnue du système au préalable.

3.2.3 Le protocole de communication Client-Serveur

Comme nous avons choisi une architecture Client - Serveur, nous devions également prévoir un protocole de communication entre les deux programmes.

Fonctionnement du serveur

Pour éviter certains blocages qui pouvaient apparaître (par exemple quand le serveur demande une confirmation à l'utilisateur, comme "Voulez vous vraiment que je vous amène au Bâtiment C ?"), nous avons choisi de coder notre programme serveur en mode "stateless". Cela signifie que le serveur n'aura aucune information stockée sur l'état du client. Grâce à ce système, lorsque l'on demande une confirmation, le serveur n'est pas bloqué à attendre une réponse à la confirmation. Il attendra tout simplement un message comme les autres, qui pourra soit être une requête qui n'a rien à voir, soit la même requête que précédemment mais avec un petit indicateur pour dire que le client l'a confirmé.

Gestion des sockets

Afin de permettre la communication entre le client et le serveur, nous avons

choisi d'utiliser des sockets en TCP, afin de profiter de tous les avantages de ce protocole (acquittements, reprise sur perte...)
Comme le serveur est "stateless", il possède simplement un socket qui écoute sur un port et attend de recevoir une connexion et un message. Il répondra au client tant que la connexion est encore ouverte, puis fermera la connexion, et se remettra en attente.
De son côté, le client se connectera au serveur pour envoyer sa requête, attendra la réponse, et fermera la connexion. Et ce, pour chaque requête émise par l'utilisateur.

Format des messages échangés

Puisque nous utilisons des deux côtés des langages de programmation objet haut niveau, nous avons souhaité utiliser des messages au format JSON. Les messages JSON ont la particularité d'être directement interprétables en tant qu'objets, ce qui simplifie grandement le traitement des messages contenant plusieurs informations simultanées.

Voici un exemple de requête JSON envoyée par le client :

```
{
    "type": "question",
    "msg": "Quelle heure est-il ?"
}
```

Cet exemple correspond à une simple question, sans particularité, et elle aura pour réponse (après traitement par le serveur) un JSON de la forme :

```
{
    "type": "answer",
    "msg": "Il est 08 heures et 45 minutes."
}
```

Cependant, il existe des requêtes qui demandent une confirmation à l'utilisateur, afin d'être sûrs d'avoir bien compris ce qu'il demandait. Pour cela, nous allons utiliser d'autres champs JSON, en suivant le schéma suivant :

```
{
    "type": "question",
    "msg": "Amène moi en Amphi 10"
}
```

Etape 1 : Requête de l'utilisateur

```
{
    "type": "askConfirmation",
    "msg": "Voulez vous que je vous amène à Amphi 10 ?",
    "originalRequest": "Amène moi en Amphi 10"
}
```

Etape 2 : Réponse du serveur (demande de confirmation)

```
{
    "type": "confirmation",
    "msg": "Amène moi en Amphi 10",
    "answer": "Oui"
}
```

Etape 3 : Confirmation de l'utilisateur

3.3 Les interfaces utilisateur

Pour intéragir avec le serveur et le configurer, nous avons codé plusieurs clients graphiques :

3.3.1 L'application Android

Bien entendu, la principale interface que nous avons programmé est celle qui permet d'utiliser la tablette embarquée sur le robot pour dialoguer en utilisant l'API STT et TTS de Google.

3.3.2 Le programme Python

Afin d'utiliser le serveur à l'écrit en local, nous avons utilisé une petite interface de debug en python. Ce client permet de tester plus rapidement le serveur en cas de modification, et pourrait facilement être améliorée si la communication à l'écrit s'avérait utile.

3.3.3 Le SkillEditor

Enfin, pour générer des TextSkills plus rapidement et plus simplement, nous avons créé une interface permettant de modifier, ajouter ou supprimer les requêtes et réponses connues par le serveur. Chacun peut ainsi éditer le fichier JSON contenant les informations sur les TextSkills à créer lors du démarrage du serveur.

```
{
    "type": "answer",
    "msg": "OK, je vous guide à Amphi 10."
}
```

Etape 4 : Réponse finale du serveur

Chapitre 4

Conclusion