Rapport de Projet Individuel :
Logiciel de contrôle de robots Khepera II
-
Master 1 Informatique
Université de Lille1

Guillaume Libersat

29 mai 2007

Table des matières

1 Introduction **4**
1.1 Le projet 4
1.1.1 Contexte 4
1.1.2 Présentation du sujet 4
1.1.3 Objectifs 4
1.1.4 Contraintes et prérequis 5
1.2 L'environnement 5

2 Mise en oeuvre **6**
2.1 Découverte du robot 6
2.1.1 Physiquement 6
2.1.2 Logiciellement 7
2.1.3 Premiers tests 7
2.2 Etat de l'art 8
2.2.1 Logiciels officiels 8
2.2.2 Logiciels tiers 8
2.2.3 Conclusion 9
2.3 Analyse 9
2.4 La bibliothèque d'accès 10
2.4.1 Mise en place 10
2.4.2 Utilisation des autotools 10
2.4.3 Premiers problèmes 10
2.4.4 Changement d'architecture 11
2.4.5 Problèmes de synchronisation 12
2.4.6 La documentation 12
2.5 Le centre de tests 13
2.5.1 Motivations 13
2.5.2 Première version 13
2.5.3 Deuxième version 14
2.6 L'interpréteur de commandes 15
2.6.1 Utilisation de Python 16
2.6.2 SWIG 17
2.6.3 Résultat 18
2.7 Le serveur réseau 18

2.7.1 Objectif initial 18
2.7.2 Xml-RPC 19
2.7.3 Première version 19
2.7.4 Version finale 22
2.7.5 Problèmes finaux 22

3 Conclusion **24**
3.1 Bilan 24
3.2 Ouverture 24

A Configuration sous GNU/Linux **26**

Remerciements

Je tiens à remercier Monsieur Philippe PREUX de m'avoir permis de travailler sur ce sujet de stage ainsi que lui-même et Monsieur Manuel LOTH pour leur suivis tout au long du semestre.

Chapitre 1

Introduction

1.1 Le projet

1.1.1 Contexte

Le projet a été réalisé dans le cadre de l'unité d'enseignement "Projet Individuel" du second semestre de Master 1 d'informatique.

Ce stage se déroule dans un laboratoire, habituellement au sein d'une équipe de recherche des Universités de Lille et est encadré par une personne de l'équipe en question.

Le projet se déroule sur toute la période du semestre, c'est à dire de Février à Juin 2007. Les élèves sont mis en autonomies et doivent accomplir les tâches annoncées lors de la publication du sujet.

1.1.2 Présentation du sujet

Le sujet que j'ai choisi concerne la mise en place d'une interface logicielle de communication avec des robots de type Khepera II.

Ce type de robots est un modèle construit par la société K-Team Corporation qui a pour but la création de robots spécifiques à la recherche et à l'éducation.

Les robots de la K-Team sont internationalement reconnnus et sont utilisés par plus de 500 universités et industriels dans le monde entier.

1.1.3 Objectifs

Le premier objectif du projet est de **réaliser un interprêteur de commandes** permettant de donner des ordres aux robots (avancer, tourner, etc).

Il doit aussi être possible de **récupérer les informations liées à leurs capteurs**.

Le second objectif est de réaliser un logiciel qui permettra **l'accès à ces robots à travers le réseau**.

Enfin, si le temps le permet, le sujet indique qu'il est envisageable d'expérimenter des applications du robot sur le terrain : localisation, cartographie, etc.

1.1.4 Contraintes et prérequis

Le sujet énonce clairement les contraintes suivantes : le travail doit être réalisé sous **GNU/Linux** et de préférence dans le **langage C**.

Il est aussi précisé que toute la documentation disponible est en anglais et qu'il est donc nécessaire d'être capable de lire cette langue.

1.2 L'environnement

Le sujet a été proposé par Monsieur Philippe Preux de l'équipe du GRAPPA. GRAPPA signifie *Groupe de Recherche sur l'Apprentissage Automatique*.

Cette équipe, maintenant située dans les locaux de l'INRIA au Parc de la Haute Borne à Lille, est actuellement composée d'une vingtaine de personne et est dirigée par PHILIPPE PREUX.

Le thème général de l'équipe est l'apprentissage automatique, c'est à dire l'élaboration d'algorithmes qui s'améliorent avec l'expérience.

Quatre robots Khepera II ont été achetés par l'équipe dans le cadre d'expérimentations liées à l'apprentissage. C'est pourquoi, l'équipe nécessite une base applicative pour pouvoir mettre en oeuvre ses expériences.

Chapitre 2

Mise en oeuvre

2.1 Découverte du robot

Avant de réaliser quoi que ce soit avec le Khepera, j'ai préféré tout d'abord lire la documentation fournie avec celui-ci. En effet, je n'avais jamais travaillé avec ce genre de matériel et n'avait aucune idée sur le fonctionnement ni les capacités de ce genre de périphériques.

J'ai ainsi pu avoir un tour d'horizon des possiblités de la machine ainsi que les précautions à prendre pour l'utiliser.

2.1.1 Physiquement

Le robot Khepera II est un concentré de technologie contenu dans 70 mm de diamètre. Il se présente comme sur la figure 2.1.

FIG. 2.1 – Le Khepera II

Il dispose de deux roues, à vitesse variable, pour se déplacer.

Seize capteurs sont situés à intervalle régulier tout au long de sa circonférence.

On distingue deux types de capteurs :

1. Les capteurs infrarouges de présence, qui réagissent à l'approche d'un objet ;
2. Les capteurs de luminosité, sensibles à l'intensité de la lumière.

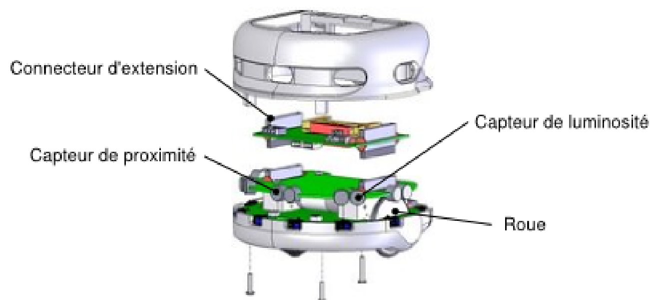

FIG. 2.2 – Anatomie d'un Khepera

On peut aussi lui associer des tourelles d'extension afin d'augmenter ses capacités (une pince, une caméra, etc).

Ces éléments sont décrits sur la figure 2.2.

2.1.2 Logiciellement

D'un point de vue fonctionnement, le Khepera II dispose de deux modes d'exécution :

1. **Autonome** : un programme interne, chargé dans sa mémoire dirige les composants. Ceci à l'avantage d'être plus efficace, mais il faut recharger le logiciel interne à chaque modification ;
2. **Contrôle distant** : Un protocole simple, par liaison série, permet d'envoyer des commandes au robot. Il est possible d'exploiter quasiment toutes les fonctionnalités de cette manière.

2.1.3 Premiers tests

Une fois familiarisé avec le fonctionnement théorique du Khepera, j'ai alors décidé de configurer le périphérique sur mon système d'exploitation et d'effectuer plusieurs tests afin de confirmer ou non ma compréhension du fonctionnement.

Une des questions que je me posais était de savoir si l'utilisation d'une tourelle sans-fil allait complexifier la méthode d'accès. En effet, chaque robot doit être contrôlé via une tourelle Bluetooth[1] et la documentation ne donnait pas d'informations à ce sujet.

[1] Une technologie radio courte distance destinée à connecter les appareils électroniques entre eux - www.bluetooth.org.

Suite à mes tests, je me suis rendu compte que le module est transparent et qu'il est possible de communiquer avec le robot comme sur la liaison série. Mieux encore, un port série est émulé au niveau du système d'exploitation.

Une de mes interrogations était donc réglée : j'allais pouvoir programmer mes logiciels indépendamment de la méthode d'accès.

Pour finir, en ce qui concerne l'utilisation même du robot, le manuel s'est avéré très clair et précis, je n'ai eu aucune surprise.

La procédure de configuration du robot est décrite en annexe.

2.2 Etat de l'art

Avant de commencer la phase d'analyse, j'ai voulu tout d'abord prendre connaissance des projets se rapprochant du mien, voire similaires.

En effet, je ne souhaitais pas réinventer la roue, et même si je me doutais que mon tuteur avait vérifié auparavant, je préférais être conscient de l'existant.

2.2.1 Logiciels officiels

Tout d'abord, j'ai analysé les solutions proposées par le distributeur du robot : deux méthodes d'accès étaient présentées.

La première consiste à se connecter directement au robot avec un émulateur de modem et lui envoyer les commandes. Cette méthode n'est clairement pas pratique et ne peut dépasser le cadre du test rapide.

La seconde quant à elle utilise le logiciel MATLAB[2]. Son principal inconvénient est d'être justement dépendante de ce logiciel et de ne pas être facilement scriptable (quasiment tout doit se faire à la souris). De plus, MATLAB ne prévoit rien de particulier pour interagir avec d'autres logiciels et n'est pas libre ; ce qui restreint beaucoup son utilisation.

Aucune des solutions proposées par le fabricant n'était donc valable dans notre cas.

2.2.2 Logiciels tiers

Après de longues recherches, il s'est avéré qu'il existe beaucoup de logiciels concernant les Kheperas, principalement en logiciels libres.

Seulement, la plupart d'entre eux sont des simulateurs ou sont écrits dans des langages de plus haut niveaux comme le *Java*[3] ou le *Python*[4].

Un seul candidat est sorti du lot : *kRobot*. Il s'agit d'une classe écrite en C++ qui permet de contrôler le robot. Cependant, la classe est devenue quasiment introuvable sur Internet et n'est pas vraiment exploitable (non maintenue, aucune documentation, etc).

---

[2]MATLAB (raccourci de matrix laboratory, laboratoire matriciel) est un logiciel de calcul numérique édité par la société américaine THE MATHWORKS.

[3]http ://java.sun.com

[4]http ://www.python.org

2.2.3 Conclusion

Je me suis donc aperçu qu'aucun logiciel existant ne proposait ce dont nous avions réellement besoin. Je devais donc repartir de zéro et implémenter ce qui était demandé en langage C.

2.3 Analyse

Je décidais donc de rassembler les éléments afin de pouvoir évaluer précisément le besoin et de préparer la phase de programmation.

J'ai tout d'abord interviewé les futurs utilisateurs de l'équipe afin de comprendre ce qu'ils attendaient réellement. J'ai ainsi pu clarifier certains points flous et m'assurer que je partais dans la bonne direction.

Je me suis ensuite attaché aux points techniques afin de prévoir les difficultés et de proposer des logiciels maintenables, réutilisables et pratiques.

Je décidais alors de programmer les éléments en couches afin d'être modulaire et de pouvoir aisément créer des applicatifs au long du stage selon l'évolution des demandes et des contraintes.

La première étape serait donc de créer une bibliothèque de manière à permettre l'abstraction du protocole Khepera. Ainsi, grâce à ce composant, il serait facile de doter n'importe quel logiciel d'un contrôle sur un robot.

Cela m'ouvrait donc les portes pour la suite : je savais que je pouvais écrire cette partie sans me soucier de la conception du serveur réseau.

Ainsi, ceci me permettait donc de me mettre au travail rapidement et de façonner les parties suivantes au fil de mon avancement.

Au final, l'architecture à laquelle je suis venue à ce moment est décrite sur la figure 2.3.

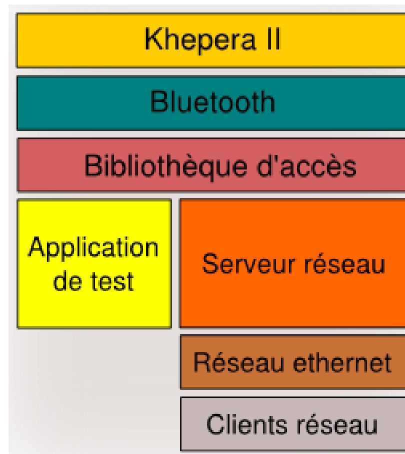

FIG. 2.3 – Une première idée d'architecture

Je décidais aussi d'utiliser un logiciel de gestion des révisions, *Subversion*[5], afin de garder une trace de mon travail, de pouvoir revenir sur les modifications en cas d'erreurs et de pouvoir travailler depuis plusieurs postes.

2.4 La bibliothèque d'accès

2.4.1 Mise en place

La bibliothèque d'accès a donc été la première brique logicielle à être programmée.

N'ayant aucune expérience en ce qui concerne la programmation de liaison série, j'ai préféré suivre un didacticiel et de créer quelques exemples pour me faire la main.

Une fois ceci terminé, j'ai alors catégorisé les possibilités du robot et en est venu à la conclusion suivante :

- Il existe deux grands types de commandes
- D'abord les "ordres" que répondent rien ;
- Ensuite les "requêtes" qui permettent d'obtenir une information.

J'avais donc à ce point tous les éléments nécessaires pour commencer à programmer.

2.4.2 Utilisation des autotools

Dans un soucis de maintenabilité, j'ai donc décidé d'utiliser les outils GNU *Automake* et *Autoconf* [6], qui permettent de créer des bibliothèques aisément et aident au bon fonctionnement sur diverses architectures.

Ainsi, cet ensemble de scripts fait que la bibliothèque pourra être compilée sur divers systèmes d'exploitation (GNU, Solaris, BSD, etc) et veillera au maximum à adapter la compilation pour le processeur de destination. Leur fonctionnement est synthétisé sur la figure 2.4.

Bien entendu, ces outils ne peuvent pas réaliser entièrement ces tâches à la place du programmeur, mais sont tout de même d'une grande aide.

Revers de la médaille, cette suite n'est pas évidente à prendre en mains. L'ayant déjà utilisée auparavant, j'ai pu réinvestir mon expérience et ai réussi à la mettre en place assez rapidement avec l'appui de la documentation en ligne qui fournit de bonnes explications illustrées.

2.4.3 Premiers problèmes

J'ai ensuite écrit les fonctions permettant d'ouvrir un canal de communication avec le robot. Le robot réagissait comme prévu, je n'ai pas eu de problème.

Suite à cela, j'ai donc écrit la partie commune aux deux classes de commandes : une partie logicielle qui s'occupera de transmettre les commandes de

[5]http ://subversion.tigris.org/
[6]Autoconf : http ://www.gnu.org/software/autoconf/

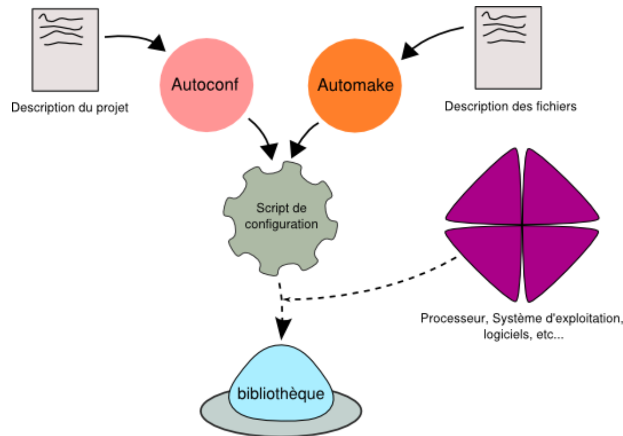

FIG. 2.4 – Fonctionnement simplifiée des Autotools

manière brute et d'en récupérer le résultat. Bien entendu, cette partie s'occupera aussi de transformer les informations en structures adaptées afin de rendre l'exploitation en C aisée.

Aux premières commandes envoyées, je me suis rendu compte qu'il y avait un problème : elles ne s'exécutaient pas la première fois qu'on se connectait au robot. A la suite de nombreux tests, j'ai découvert qu'une bannière était émise lors de la première connexion et qu'elle génait la communication.

Le problème peut paraître trivial, mais ça n'a pas été évident de le résoudre. Après plusieurs essais qui se sont soldés par un échec, j'ai décidé d'inspecter le code source de plusieurs programmes utilisant des ports séries. Le logiciel GPhoto[7], une bibliothèque d'accès aux appareils photographiques m'a été d'une grande assistance.

Il s'est avéré qu'une fonction peu référencée permet de résoudre ce problème ; je l'ai donc utilisée et le problème semblait résolu.

2.4.4 Changement d'architecture

Au début du développement, j'ai commencé la programmation sur une machine de type *Macintosh*. Au tiers du développement, j'ai acquis une nouvelle machine, et cette fois ci de type *PC/Intel*.

Comme je l'attendais, j'ai eu quelques problèmes de fonctionnement. Ceci à cause de deux facteurs : tout d'abord mon inaptitude à prévoir tous les cas de portabilité et surtout un fonctionnement légèrement différent des liaisons séries.

En effet, les paramètres que j'appliquais sur la liaison venaient s'additionner à la configuration "de base" du port. Or, cette configuration "de base" différait selon le type de machine.

[7]http ://www.gphoto.org

J'ai alors du changer mon programmer afin d'écraser totalement la configuration du port afin de le mettre dans le mode requis par le Khepera.

2.4.5 Problèmes de synchronisation

Une fois l'écriture terminée des "ordres", je me décidais donc de passer à l'écriture des "requêtes", qui elles, nécessitent une réponse.

J'ai donc réfléchi à des structures adaptées afin de recueillir les réponses du robot et ai commencé à programmer.

Je me suis vite confronté à des problèmes de synchronisation. Par exemple, une réponse était lue partiellement et la suivante contenait une partie de la précédente.

A la suite de nombreux tests, je n'avais toujours pas réussi à déterminer d'où cela venait. Il m'a fallu utiliser *GDB*[8] afin d'arriver à un diagnostic précis.

Il s'agissait d'un problème de mise en tampon au niveau du système d'exploitation, couplé avec des difficultés de transmission sur le Khepera.

Je pouvais éventuellement rendre le système synchrone en désactivant la mise en tampon, mais cela allait impacter de manière critique sur les performances.

Suite à de nombreuses recherches et tests, je suis parvenu à un système quasiment stable. Cela consiste à fonctionner en mode semi-synchrone et à se caler sur le temps d'un cycle d'exécution du Khepera. Cependant, le robot sature très rapidement et il arrive que suite à l'envoi de nombreuses commandes, il se mette à paniquer et redémarre inopinément.

Cela reste encore un problème à ce jour et je ne pense pas réussir à faire mieux sur ce point. Le problème étant dû, je suppose, au logiciel interne du Khepera.

2.4.6 La documentation

Toujours dans un souci de maintenabilité, ajouté à la volonté de rendre l'utilisation pratique de ma bibliothèque, j'ai décidé d'intégrer un système de génération automatique de documentation.

Mon choix s'est porté sur le logiciel *Doxygen*[9] qui est un programme éprouvé, fiable, complet et qui génère une documentation claire.

Il suffit d'annoter les éléments dans le code source du programme pour qu'ils soient repris dans la documentation. Cela permet donc d'obtenir un manuel pour les développeurs très rapidement.

De plus, plusieurs formats peuvent être générés (PDF, Html, texte, ...).

Le fonctionnement de ce programme est illustré sur la figure 2.5.

Cependant, comme évoqué précédemment, cet outil ne génère qu'une documentation pour les développeurs et ne remplacera pas un manuel.

En somme, le travail le plus compliqué a été d'intégrer *Doxygen* dans les *Autotools*. Mais ce temps passé a été compensé par le temps gagné à la rédaction d'une documentation.

[8]Le débogeur GNU : http ://sourceware.org/gdb/
[9]http ://www.doxygen.org

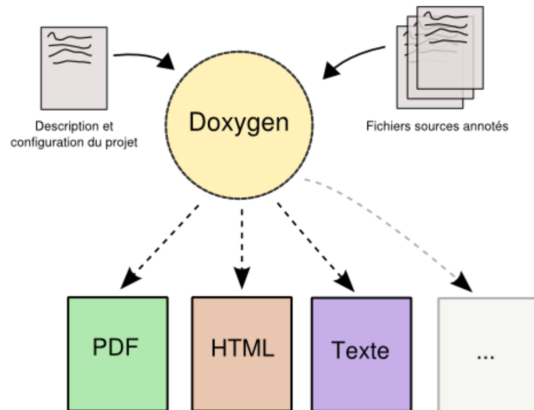

FIG. 2.5 – Fonctionnement de Doxygen

2.5 Le centre de tests

2.5.1 Motivations

Au début du développement de la bibliothèque, je me suis dit qu'il fallait que j'effectue toute la phase de programmation par étapes, au rythme des tests. J'ai donc emprunté quelques idées aux méthodes d'*Extreme Programming* [10] afin de ne valider mon code que lorsque les tests passaient avec succès.

Je n'ai cependant pas réalisé une vraie suite de tests comme il est décrit dans cette méthode mais j'ai décidé de programmer une application qui utilise les fonctionnalités de la bibliothèque.

Ainsi, dès lors qu'un dysfonctionnement apparaîtrait dans celle-ci, elle se répercuterait dans l'application et ce serait flagrant.

Je décidais donc d'avancer étape par étape et seulement lorsqu'un test avait validé le travail que je venais d'effectuer.

2.5.2 Première version

La première version du centre de tests était un simple programme en ligne de commandes qui appelait fonction par fonction et testait la validité des résultats.

Je me suis cependant vite aperçu que l'exécution de ces tests était très linéaire, quasi à intervalle régulier. De plus, ils ne représentaient pas ce qu'un Homme pourrait demander au robot.

C'est pourquoi, au milieu de l'écriture de la bibliothèque, je décidais de réécrire ce code de test et d'en faire quelque chose de contrôlable par l'utilisateur.

[10] http ://fr.wikipedia.org/wiki/Extreme_programming

2.5.3 Deuxième version

Cette nouvelle version donc, devait être capable de réagir aux interactions avec l'utilisateur et de l'informer en permanence sur l'état du robot.

J'ai à ce moment hésité à écrire l'interpréteur de commandes. Cependant, le problème de l'interpréteur est qu'il aurait eu les mêmes défauts que le code de test :

- L'utilisateur ne pourrait pas stresser le robot (saisir les instructions est plus pour nous que les exécuter pour le robot) ;
- Utiliser une script reviendrait à écrire un test linéaire en C.

Il me paraissait donc logique d'éviter l'interpréteur qui n'apporterait rien pour cette phase de tests.

Je choisissais donc de m'orienter vers une interface graphique qui permettrait la saisie très rapide des commandes et d'avoir un retour visuel direct.

J'ai alors prototypé rapidement une interface et est arrivé au cahier des charges suivant :

- Appel des commandes sur simple pression d'une touche ;
- Affichage constant de l'état des capteurs ;
- Affichage constant de la vitesse du robot.

L'interface graphique

Afin de construire l'interface graphique de mon application, il me fallait donc utiliser un toolkit[11] graphique.

J'ai alors envisagé plusieurs choix. Pour chacune de ces solutions, j'ai évalué le pour et le contre. Le résultat de cette analyse est regroupé dans le tableau 2.1.

Nom	Avantages	Inconvénients
Xlib	Très performant	Complexe, moyennement portable, laid
GTK, **Qt**, ...	Portable, visuellement agréable, beaucoup de widgets[12] disponibles	Evénements difficiles à capturer, temps de rafraichissement élevé
SDL, **Allegro**, ...	Adapté à la réception d'évenements, rafraichissement très rapide, portable	Aucun widget, assez bas niveau

Tab. 2.1 – Comparaison des toolkits graphiques

Mon choix s'est alors porté sur la bibliothèque *Simple Direct Layer*, ou *SDL*[13], car c'était pour moi la plus adaptée. En effet, l'interface que je souhaitais réaliser s'apparentait plus, d'un point de vue technique, à un jeu qu'à une application graphique "classique".

[11] Un ensemble d'outils facilitant l'écriture de logiciels dans un domaine particulier.
[13] http ://www.libsdl.org

Tout d'abord, j'avais besoin d'un rafraîchissement constant et paramètrable ; ensuite, je devais être capable d'utiliser divers périphériques d'entrée (l'équipe avait prononcé le souhait d'utiliser un joystick par exemple).

Bien que la *SDL* me laissait beaucoup plus de libertés et me permettait d'avoir exactement ce que je voulais, je ne disposais d'aucune assistance en ce qui concerne le rendu graphique.

En effet, cette bibliothèque se contente d'afficher des pixels sur l'écran et ne connaît quasiment aucun primitive plus évoluée. Il me fallait donc soit créer une boite à outils pour afficher des formes ou m'intéresser à ce qui se faisait déjà.

J'ai donc évidemment choisi de commencer par la deuxième solution, toujours dans le but d'éviter de réécrire quelque chose d'existant et de limiter les bogues.

J'ai alors découvert la bibliothèque *Cairo*, qui venait de faire son apparition en version stable. Cette bibliothèque permet de dessiner à l'écran en mode vectoriel et fournit toutes sortes de primitives pour manier les formes. Un des principaux avantages du vectoriel est qu'il s'adapte parfaitement à toutes les résolutions d'écrans sans sacrifier la qualité.

L'utilisation de *Cairo* en combinaison avec la *SDL* m'a alors paru idéal, cette première me paraissant un bon compagnon pour la dernière.

Description

L'interface se présente donc, en version finale, comme illustré sur la figure 2.6.

Au centre, on trouve un "pad" multidirectionnel qui indique quelle est la direction actuellement empruntée. Actuellement, il n'est possible de diriger le Khepera qu'avec les flèches du clavier, mais l'ajout d'un autre périphérique se révèle trivial grâce à la *SDL*.

Sous le "pad", se trouvent deux barres qui indiquent la résistance au sol. Elles sont calculées en effectuant la différence entre la vitesse demandée et la vitesse réelle.

Enfin, de chaque côté, on trouve 8 barres qui correspondent à l'état des différents capteurs. Sur la gauche se trouvent les capteurs de présence, tandis que sur la droite sont représentés les capteurs de luminosité.

2.6 L'interpréteur de commandes

Une fois la bibliothèque suffisamment complète et stabilisée grâce à l'outil précédemment cité, je décidais donc de m'occuper de l'interprêteur de commandes.

Comme demandé dans le sujet, il devait être possible de contrôler le robot au travers d'une invite de commande.

Ayant déjà eu l'occasion de réaliser un interprêteur de commandes, j'ai voulu éviter de retomber dans le même piège. En effet, réaliser ce type de logiciel n'est

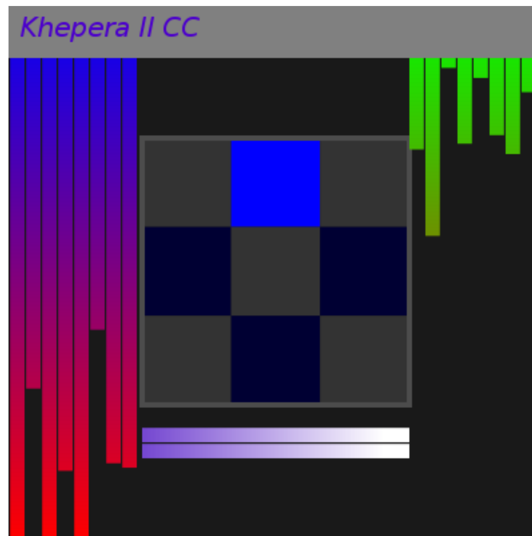

Fig. 2.6 – Le centre de contrôle

pas évident du tout. Beaucoup de bogues sont à prévoirs et le logiciel doit être éprouvé sur une longue phase de déboguage pour être considéré stable.

Il m'est alors venu l'idée d'utiliser un langage déjà existant possédant une interface en ligne de commande. Cela aurait deux avantages :

1. Tout d'abord, de disposer d'un interprêteur riche et convivial ;
2. Ensuite d'utiliser un langage éprouvé et connu.

Le candidat idéal m'a semblé être le langage *Python*[14] qui dispose des qualités requises sans aucun ajout de module externe.

2.6.1 Utilisation de Python

Pour utiliser *Python*, je devais donc écrire un module qui permettrait de réaliser la cohésion entre la bibliothèque en langage *C* et le langage *Python*.

En effet, ce dernier étant très divergeant du *C*, il n'est pas possible d'utiliser automatiquement les bibliothèques écrites en *C*. Elles doivent être soient écrites entièrement en python ou un module intermédiaire doit réaliser la conversion.

Toujours dans un souci de maintenabilité, j'ai alors cherché un générateur de “bindings”[15]. Ceci avait de nombreux avantages :

- Mise à jour synchronisée avec l'évolution de la bibliothèque *C* ;
- Gain de temps considérable ;
- Compatibilité avec les dernières versions de *Python* assurée ;
- Possibilité d'étendre les bindings vers d'autres langages (exemple : *Caml*).

[14]Python : http ://www.python.org
[15]Un code qui sert de passerelle entre différents langages

J'ai alors trouvé le logiciel *SWIG*[16] qui m'a paru complet, abouti et bien maintenu. Cependant, il m'a aussi paru assez ardu à utiliser. Mon intuition ne m'avait pas menti.

2.6.2 SWIG

Swig est un "générateur d'interfaces" qui permet de créer des bindings à partir de fichiers d'en-tête.

Il est actuellement capable de générer des modules pour plus de quinze langages et ce, d'une manière quasiment uniforme. Le mécanisme de génération est représenté sur la figure 2.7

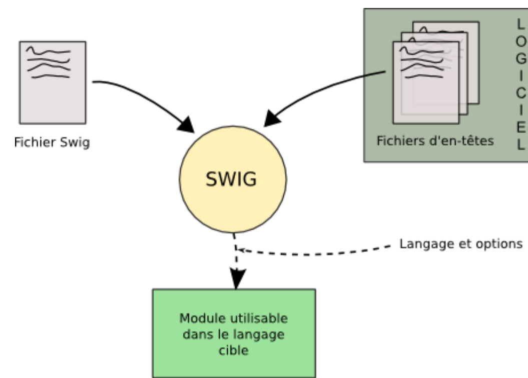

FIG. 2.7 – Mécanisme de génération de SWIG

Son utilisation est en revanche non triviale. Fort heureusement, la documentation est assez exhaustive et permet, avec de l'investissement, d'y trouver ce que l'on souhaite.

J'ai tout d'abord réalisé les bindings pour les commandes de type "ordre". Ceci s'est passé sans problème. Lorsque j'ai commencé à faire générer les commandes de type "requête", le travail est devenu beaucoup moins trivial.

Premièrement, *SWIG* ne supportant pas la déclaration d'union[17] dans une structure, il ne réussissait donc pas à convertir le type de retour pour une requête. J'ai du alors l'adapter. Le plus gros problème allait suivre. Je me suis rendu compte qu'il ne comprenait pas les pointeurs, si ce n'est d'une manière abstraite.

SWIG fonctionne ainsi : il ne se préoccupe pas de connaître les détails et se base sur des hypothèses. Si un élément déclare un pointeur, il va se contenter d'en regarder le type et de générer les mécanismes pour accéder au contenu pointé. Il ne se préoccupera jamais de faire quelque chose d'intelligent en fonction du type.

[16]Simplified Wrapper and Interface Generator : http ://www.swig.org

[17]Une union est une zone mémoire qui peut accueillir plusieurs types, ceci de manière exclusive.

Ceci a donc posé problème pour les tableaux. En effet, *SWIG*, suivant sa logique, permettait d'obtenir l'élément pointé par le "pointeur-tableau" et donc de n'accéder qu'au premier élément de celui-ci.

Cela posait vraiment un problème pour lire les capteurs par exemple. J'ai donc du utiliser ce qu'on appelle un "typemap", qui permet d'automatiquement adopter un comportement en fonction d'un type rencontré. J'avais donc maintenant la possiblité d'indexer le pointeur comme un tableau normal.

2.6.3 Résultat

Une fois tous les problèmes réglés, j'ai intégré *SWIG* aux *Autotools* afin de rendre la génération des "bindings" automatique et paramètrable à l'aide des commandes classiques.

Le résultat est un code *Python* utilisable naturellement par une personne qui connaît la bibliothèque *C*. Le grand avantage est donc que la documentation reste valide pour l'utilisation en python.

Voici un exemple très succint d'une session :

```
» from khepctrl import *
» h = khep_open("/dev/rfcomm0", KHEP_B115200)
» khep_cmd_set_speed(h, 10, 10)
» khep_close(h)
```

On obtient donc ce qui était demandé : il est possible d'exploiter tout le potentiel du robot à l'aide d'un interprêteur de commandes.

J'ai ensuite apporté quelques améliorations aux "bindings" en fin de projet.

Tout d'abord, les codes de retours dans le "style *C*" (*i.e.* des entiers) ont été remplacés par des exceptions *Python*, ce qui permet d'avoir une gestion des erreurs moins lourde.

Ensuite, j'ai commencé l'écriture d'une classe *Python* afin d'accéder aux robots sous forme d'objets; ce qui facilite l'exploitation et rend l'accès encore plus naturel.

2.7 Le serveur réseau

2.7.1 Objectif initial

Le serveur réseau, comme décrit dans le sujet, devait être accessible par socket[18]. La procédure à suivre était donc d'ouvrir un socket distant puis d'y écrire les ordres en caractères ASCII.

Comme on peut le remarquer, ceci présente de nombreux désavantages.

En premier lieu, on effectue un pas en arrière comparé à la bibliothèque : l'abstraction qu'elle apporte vis à vis du protocole n'existe plus.

[18] Un canal de communication brut entre deux points.

Ensuite, ceci signifie donc que l'on n'a plus aucun contrôle de types, du nombre d'arguments, etc... L'intérêt qui était gagné est ici perdu et est une porte ouverte aux fautes d'inattention lors de la phase de programmation.

Enfin, les sockets ne sont pas toujours évidents à gérer. Selon les langages, il est plus ou moins facile de les manipuler et sont parfois source d'erreurs. De plus, lorsqu'on écrit une application réseau, il est confortable de ne pas avoir à se soucier du transport des données.

J'ai alors proposé une autre solution à mon tuteur de stage : *Xml-RPC*. Après examen de celle-ci, il m'a conforté dans mon choix et m'a donné le feu vert pour l'exploiter.

2.7.2 Xml-RPC

Xml-RPC est un protocole de haut niveau qui n'utilise que des standards bien établis du Web. Il permet une communication facile et homogène à travers le réseau.

Tous les messages sont encodés en *XML*[19] selon des règles bien établies et sont transportés sur le réseau en utilisant le protocole *HTTP*[20].

Ceci permet de communiquer sans avoir à se préoccuper ni du langage source ni du langage cible. En effet, chaque bibliothèque respectant les mêmes normes, les données sont donc échangeables de manière transparente.

Le premier grand intérêt est donc qu'il sera possible d'utiliser le serveur avec quasiment tous les langages disponibles.

Ensuite, comme *Xml-RPC* utilise le protocole *HTTP*, n'importe quel serveur Web peut être utilisé pour gérer les connexions et le transport des données. Ceci apporte l'avantage de pouvoir se reposer sur des logiciels déjà existants, éprouvés et robustes.

Un autre avantage de cette solution est qu'il est possible d'utiliser les mécanismes internes du serveur Web comme la redirection automatique vers un autre serveur (maintenance, équilibrage de charge, etc), l'authentification ou encore le chiffrement *SSL*[21].

Le seul point noir que l'on pourrait attribuer à l'utilisation d'*Xml-RPC* est qu'il est nécessaire d'avoir une bibliothèque capable de comprendre ce langage à chaque point de communication. Cependant, comme dit précédemment, celles-ci existent pour la plupart des langages.

Voici un schéma regroupant le fonctionnement d'*Xml-RPC* (figure 2.8).

2.7.3 Première version

La première version que j'ai développée était écrite en langage *C*. J'avais choisi d'utiliser la bibliothèque "libxmlrpc"[22] qui se décrit comme un moyen rapide et facile à mettre en oeuvre tout en étant modulaire, portable et complet.

[19] Un langage de description utilisant des balises
[20] Le protocole utilisé pour transporter le "Web"
[21] La méthode utilisée sur le Web pour communiquer en chiffré
[22] LibXMLRPC : http ://xmlrpc-c.sourceforge.net/

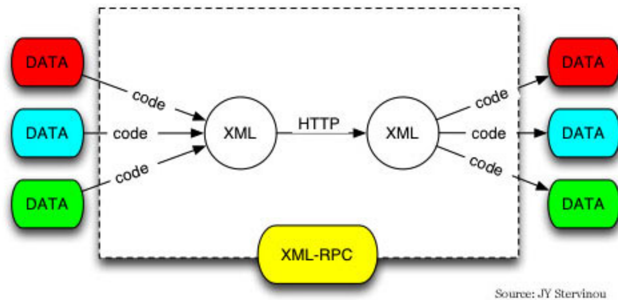

Fig. 2.8 – Fonctionnement global d'XML-RPC

Cette bibliothèque a aussi le gros avantage d'embarquer une copie d'un serveur Web très léger : *Abyss*[23], ce qui permet d'avoir un composant autonome et facile à déployer.

La documentation étant claire, fournie d'exemple, j'ai pu rapidement avoir un programme minimal fonctionnel. La bibliothèque semblait effectivement respecter ce qui était annoncé sur le site web. Je décidais donc d'intégrer ma bibliothèque Khepera avec celle ci.

Génération des fonctions

J'ai alors remarqué un point non négligeable : il n'est pas possible de générer les fonctions *Xml-RPC* à partir des fichiers d'en-têtes. J'ai trouvé cela fort dommage pour deux raisons :

1. Il faudrait mettre à jour le serveur réseau à chaque changement de la bibliothèque ;
2. Le processus de création de fonction *Xml-RPC* est très linéaire et réberbatif, donc sûrement automatisable.

J'ai donc écrit rapidement un outil qui, à partir d'un fichier d'en-tête et quelques informations, génère un fichier source *C* contenant les fonctions *Xml-RPC* correspondantes.

Problème significatif

A ce moment, j'avais donc un logiciel qui compilait, se lançait et était prêt à recevoir les requêtes *Xml-RPC*.

Je fus désagréablement surpris lors du premier test : j'obtenais comme erreur "*descripteur de fichier non valide*"[24] sur chacun de mes appels.

[23]Le serveur Web Abyss : http ://abyss.sourceforge.net/

[24]Un descripteur de fichier est un numéro qui permet d'identifier un fichier ouvert afin d'effectuer des opérations sur celui-ci.

Mes tests me montraient pourtant que les descripteurs de fichiers étaient bien attribués. Après de nombreuses tentatives diverses, j'ai enfin saisi le problème : le descripteur de fichier était bien ouvert à la demande, mais aussitôt fermé, donc inutilisable par les appels suivants.

Je devais donc trouver la raison de cette fermeture soudaine. Ce problème m'a pris un temps considérable car j'ai dû explorer de nombreuses pistes. Peu à peu, je me suis rapproché du fonctionnement interne du serveur Web et j'ai alors trouvé le problème.

En effet, la plupart des serveurs Web travaillent ainsi : lors de la réception d'une connexion, le programme est dupliqué en mémoire (un *fork* ou un *thread*), traite la demande puis se termine.

Or, sur un système POSIX[25], lorsqu'un programme quitte, tous les descripteurs de fichiers qui lui étaient associés sont fermés.

Le problème est illustré sur la figure 2.9.

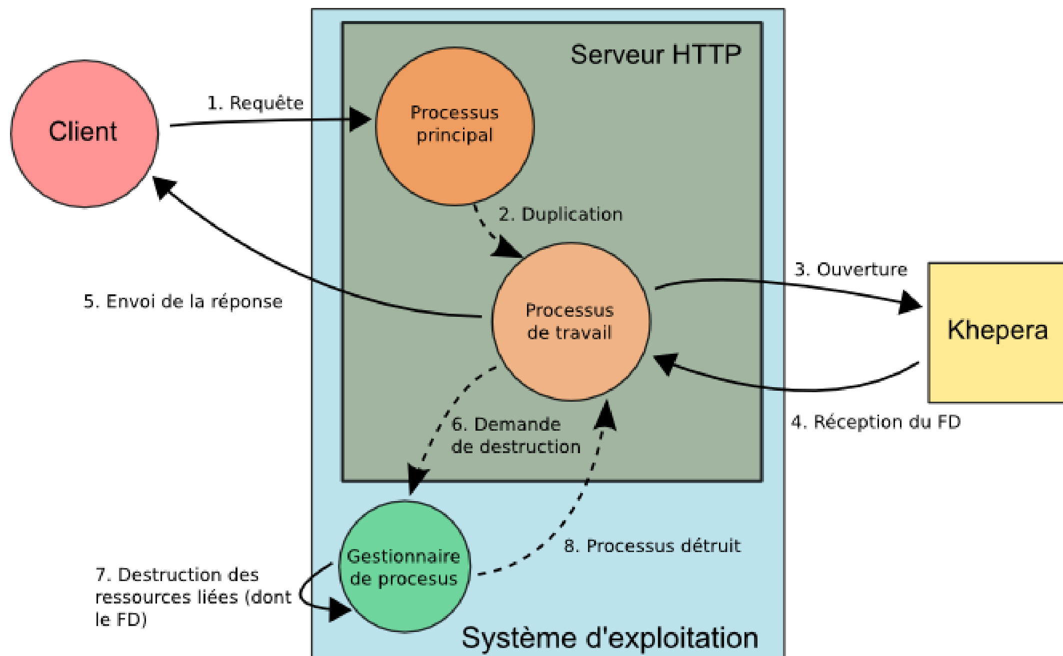

FIG. 2.9 – Mécanisme réalisé lors d'une requête d'ouverture d'un Khepera

J'avais donc trouvé mon problème et il n'était pas simple à résoudre. Il me fallait donc soit créer un serveur web spécifique ne fonctionnant qu'avec des threads pré-chargés, soit trouver un serveur web fonctionnant ainsi.

[25]Une famille très répandue de systèmes : http ://fr.wikipedia.org/wiki/POSIX

2.7.4 Version finale

J'ai alors pensé qu'il était peut être possible de tirer profit des bindings *Python*. En effet, le langage *Python* est basé sur une machine virtuelle qui utilise un pool[26] de threads pour exécuter les tâches qui nécessitent un processus différent.

L'utilisation de ce pool aurait l'avantage de ne pas fermer les descripteurs de fichiers lors de la fin de traitement et donc d'être utilisables pour le cas de figure recherché.

De plus, *Python* possédant naturellement de nombreux mécanismes d'introspection, j'avais l'intuition qu'il serait aisé de générer automatiquement les fonctions *Xml-RPC*.

Après m'être documenté sur les possibilités offertes par le module RPC de *Python*, j'en ai donc conclu qu'il s'agissait du candidat idéal.

La mise en pratique fût rapide et convaincante : quelques lignes de code ont suffit à la déduction automatique des fonctions exportées par la bibliothèque et les descripteurs de fichiers étaient bien gardés ouverts.

J'avais donc atteint mon objectif : je disposais d'un serveur Web pouvant contrôler plusieurs robots, pouvant avoir plusieurs clients (même par robot), se mettant à jour automatiquement et gardant la même API que la bibliothèque (donc la même documentation).

L'imbrication des composants est détaillée sur la figure 2.10.

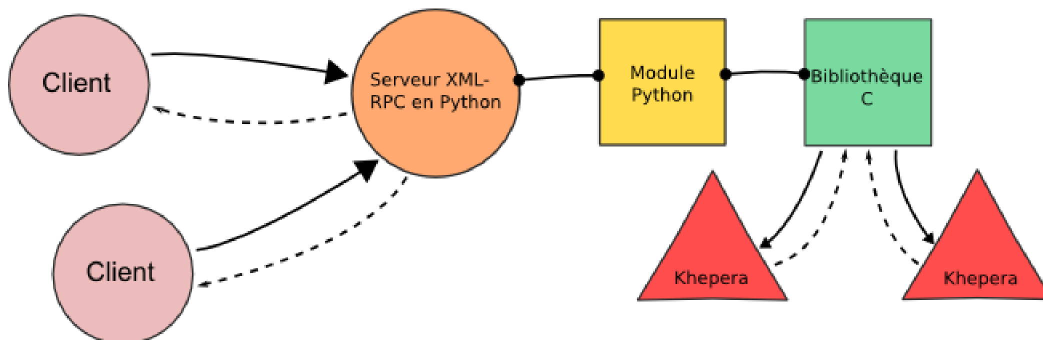

FIG. 2.10 – Imbrication des composants

2.7.5 Problèmes finaux

Bien que le serveur fonctionnait parfaitement pour les commandes de type "ordre", il n'en était pas de même pour certaines commandes de type "requête".

En effet, *SWIG* gérant les types d'une manière totalement générique, il crée ce qui est appelé un "proxy" permettant d'accéder à ces types.

[26]Un pool est un ensemble d'objets préchargés

Le problème est que pour qu'un objet puisse passer à travers le réseau, il doit être sérialisable[27]. Or, les proxies SWIG ne le sont pas.

J'ai donc dû préciser à *SWIG* comment les types non-natifs (*i.e.* hormis entiers, chaînes de caractères, etc...) pouvait être représentés d'une manière sérialisable.

Ce problème m'a pris beaucoup de temps car il m'a demandé d'utiliser des concepts avancés de *SWIG* qui ne sont pas évidents à prendre en main.

Une fois ce problème réglé, j'ai pu tester le serveur exhaustivement et d'une manière convaincante.

[27]Cela signifie qu'il est possible de créer une représentation textuelle d'un objet.

Chapitre 3

Conclusion

3.1 Bilan

Pendant toute la durée du stage, j'ai pu enrichir mon expérience sur deux aspects.

Tout d'abord, d'un point de vue technique, j'ai voulu mettre l'accent sur le côté génie logiciel car c'est, selon moi, un critère indispensable pour livrer un programme.

En effet, le fait de savoir que d'autres programmeurs auront à maintenir le logiciel par la suite impose de nombreuses contraintes.

Je me suis donc tenu à rechercher au maximum une maintenabilité aisée, une automatisation des procédés ainsi qu'une documentation la plus complète possible.

Je pense avoir réussi sur ce point car tous les composants sont générés l'un après l'autre en prenant comme seul point de départ la bibliothèque *C*.

On comprend donc bien tout l'intêret : le fait de changer quelque chose dans celle ci se répercutera automatiquement dans toute la chaîne (module *Python*, interprêteur de commandes, serveur *Xml-RPC*, documentation).

Ensuite, d'un point de vue personnel, j'ai pu découvrir le fonctionnement et l'ambiance de travail dans un laboratoire.

Même s'il ne s'agissait pas d'un travail de recherche, je me suis intéressé et ai pu comprendre la nature des travaux de l'équipe.

Ceci m'a d'ailleurs permis de faire mon choix pour mon orientation en Master 2.

3.2 Ouverture

Même si le travail réalisé est immédiatement exploitable, voici ce qui aurait pû être réalisé avec plus de temps :

- La gestion des tourelles des Khepera (Grippers, etc). Programmer ceci sous forme de greffons pourrait être une bonne idée ;

- Un logiciel de calibration du robot (ainsi que le support des profils dans la bibliothèque) permettant d'avoir une étalonnage fin selon ses préférences ;
- L'amélioration de la stabilité à l'aide de nombreux tests afin de stresser encore plus le robot ;
- Des bindings vers d'autres langages (*Objective-Caml* par exemple). *SWIG* supportant de nombreux langages, le travail le plus long serait sûrement l'intégration aux *Autotools*.
- La possibilité de réserver un robot sur le serveur réseau selon ses capacités. Ceci avait été proposé par un membre de l'équipe.

Enfin, suite à la clarification de la licence (tous les logiciels sont couverts sous la licence *GNU GPL*[1]), des paquets *Debian*[2] sont en cours de création afin de faciliter l'installation des logiciels sur les machines de développement.

[1] http ://www.gnu.org/licenses/licenses.html#GPL
[2] Système d'exploitation GNU/Linux, http ://www.debian.org

Annexe A

Configuration sous GNU/Linux

Afin d'utiliser la bibliothèque d'accès, la première étape consiste à configurer le Khepera II sur le système. Voici comment réaliser l'installation sous Debian GNU/Linux (cependant, ceci est facilement adaptable pour les autres distributions).

Paquets nécessaires :

- bluetooth
- bluez-utils
- libbluetooth2

Ils sont installables via la commande "*apt-get install NOM_PAQUET*".

Une fois ceux-ci installés, il faut relever le code MAC du robot. Pour ceci, une fois le robot sous tension, exécuter "*hcitool scan*".

Enfin, il est nécessaire de renseigner le fichier "/etc/bluetooth/rfcomm.conf" avec les informations suivantes :

```
rfcomm0 {
       # Automatically bind the device at startup
       bind yes;

       # Bluetooth address of the device
       device 00:04:XX:XX:XX:XX;

       # RFCOMM channel for the connection
       channel 1;

       # Description of the connection
       comment "KheperaII robot#1";
}
```

S'il y a plusieurs robots, il suffira de répéter l'opération autant de fois que

nécessaire sans oublier d'incrémenter "rfcommX".

Une fois le fichier enregistré, un redémarrage de la pile bluetooth est nécessaire. Ceci s'effectue à l'aide de "*/etc/init.d/bluetooth restart*".