UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Génie électrique et génie informatique

ARCHITECTURE DE COOPÉRATION MULTI-ROBOTS

Mémoire de maîtrise es sciences appliquées
Spécialité : génie électrique

Jasmin LETENDRE

Sherbrooke (Québec) Canada

Avril 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-05929-X
Our file Notre référence
ISBN: 0-494-05929-X

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

RÉSUMÉ

Les robots sont appelés à jouer un rôle de plus en plus important dans notre société. Leur nombre toujours croissant augmentera la quantité d'interactions entre eux et ils devront arriver à travailler en collaboration. Il est donc important de développer les bases logicielles qui permettront de faciliter ces interactions.

Ce mémoire présente *AIDER*, une Architecture pour l'Interaction Dynamique Entre Robots. Le but de cette architecture est de permettre le développement rapide de systèmes multi-robots robustes pouvant accomplir un grand éventail de tâches. L'approche proposée est un hybride entre l'approche comportementale et l'approche multicouche et elle permet une grande flexibilité dans les interactions à l'intérieur d'un groupe de robots. Pour ce faire, *AIDER* permet l'échange de capacités, le partage transparent de ressources entre les robots ainsi que la possibilité d'exécuter des arbres de tâches distribués dont les branches se situent sur différents robots. Ces différents mécanismes permettent aux robots d'interagir entre eux à tous les niveaux, du partage des ressources à la prise de décision.

Un scénario d'exploration planétaire a été développé afin de valider l'architecture. Son implémentation a permis de confirmer que *AIDER* permet bel et bien un développement plus facile et plus rapide de tâches multi-robots, tout en ayant des performances lui permettant d'être utilisée dans des systèmes où la puissance de calcul ou les capacités de communication sont restreintes.

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur de maîtrise, François Michaud, professeur au Département de génie électrique et de génie informatique, pour m'avoir soutenu et aidé tout au long du projet par ses commentaires judicieux.

Je remercie aussi mon co-directeur, Erick Dupuis, de l'Agence spatiale canadienne, pour m'avoir assuré un soutien technique tout au long de mes travaux et pour ses questions précises m'obligeant à avoir une idée claire du chemin que j'allais prendre. J'inclus dans ces remerciements Pierre Allard et Régent L'Archevêque, du Département de robotique de l'Agence spatiale canadienne, qui m'ont aidé à de nombreuses reprises et avec qui j'ai eu plusieurs conversations enrichissantes sur le développement d'un système de contrôle de robots.

Enfin, je tiens à remercier Sonia, mon épouse, pour son soutien sans faille dans les meilleurs moments comme dans les moments plus difficiles de ma maîtrise.

Je termine en soulignant le support du Fonds de recherche sur la nature et les technologies et de LABORIUS, le Laboratoire de robotique mobile et de systèmes intelligents de l'Université de Sherbrooke, qui ont contribué financièrement à ce projet.

TABLE DES MATIÈRES

1 INTRODUCTION **1**

2 APPROCHES POUR LE CONTRÔLE DE GROUPES DE ROBOTS **3**

2.1 Approches avec peu ou pas de communication 4

2.2 Approches avec communication explicite 5

2.3 Architectures de coopération à plusieurs niveaux 7

2.4 Concepts applicables à une architecture multi-robots 10

3 ARCHITECTURE *AIDER* **13**

3.1 Objectifs 13

3.2 Lignes directrices 14

3.3 Description de l'architecture 15

3.3.1 Gestion des capacités 16

3.3.2 Gestion des tâches 18

3.3.3 Gestion des ressources 23

3.3.4 Gestion de la communication 27

3.3.5 Configuration d'un système 30

4 SCÉNARIO D'ÉVALUATION **35**

4.1 Objectifs du scénario 35

4.2 Description 37

4.3 Environnement d'implémentation 40

4.3.1 Ressources matérielles 40

4.3.2 Ressources logicielles 43

4.4 Implémentation avec *AIDER* 43

4.4.1 Ressources 44
4.4.2 Tâches 46

5 ANALYSE ET DISCUSSION **54**

5.1 Temps de développement 55
5.2 Performances 63
5.2.1 Utilisation du processeur 63
5.2.2 Utilisation de la bande passante 67
5.2.3 Temps de réaction 69
5.3 Travaux futurs 70

6 CONCLUSION **72**

ANNEXE I - FICHIER DE CONFIGURATION DU ROBOT RÉEL **74**

ANNEXE II - RÉSULTATS DES TESTS DU SCÉNARIO **77**

ANNEXE III - RÉSULTATS DES TESTS DE TEMPS DE RÉACTION **82**

BIBLIOGRAPHIE **83**

LISTE DES FIGURES

Figure 3.1 Structure de base d'*AIDER* 16
Figure 3.2 Contenu d'une description de capacité 17
Figure 3.3 Arbre de tâches 18
Figure 3.4 arbre de tâches distribuées 19
Figure 3.5 Transition entre les états d'une tâche 20
Figure 3.6 Étapes du lancement d'une tâche distante 22
Figure 3.7 Étapes de la requête d'accès à une ressource distante 25
Figure 3.8 Exemple d'utilisation d'une interface générique 26
Figure 3.9 Exemple d'utilisation d'une interface spécifique 26
Figure 3.10 Traitement de l'adresse de destination d'un message 29
Figure 3.11 Contenu d'un message 31
Figure 3.12 Structure de base d'un fichier de configuration 34

Figure 4.1 P2-AT modifié utilisé dans le scénario 40
Figure 4.2 Terrain utilisé pour le scénario 42
Figure 4.3 Arbre de tâches pour la partie déplacement du scénario 47
Figure 4.4 Exemple de carte d'exploration 50
Figure 4.5 Arbre de tâches du scénario 53

Figure 5.1 Exemple de la fonction principale d'une tâche 58
Figure 5.2 Exemple de la fonction d'initialisation d'une tâche 59
Figure 5.3 Exemple de la fonction de mise en pause d'une tâche 59

Figure II.1 Arbre de tâches à l'étape 1 79
Figure II.2 Arbre de tâches à l'étape 2 79
Figure II.3 Arbre de tâches à l'étape 3 80

Figure II.4 Arbre de tâches à l'étape 4 80
Figure II.5 Arbre de tâches à l'étape 5 81

LISTE DES TABLEAUX

Tableau 3.1 Exemples de tâches 18

Tableau 4.1 Rôles des robots 40
Tableau 4.2 Ressources utilisées dans le scénario 51
Tableau 4.3 Tâches utilisées dans le scénario 52

Tableau 5.1 Caractéristiques des ordinateurs utilisés 63
Tableau 5.2 Performance pour la situation # 1 64
Tableau 5.3 Performance pour la situation # 2 64
Tableau 5.4 Performance pour la situation # 3 64
Tableau 5.5 Performance pour la situation # 4 65
Tableau 5.6 Principaux modules utilisateurs de mémoire 67

GLOSSAIRE

Arbre de tâches. Ensemble des tâches actives sur un robot, représentées en fonction des relations parent-enfant liant chacune des tâches.

arbre de tâches distribuées. Arbre de tâches qui contient des tâches s'exécutant sur un robot différent de la tâche racine de l'arbre. Un arbre de tâches distribuées possède donc des branches qui s'étendent sur d'autres robots.

Capacité. Tâche qu'un robot peut accomplir ou ressource que celui-ci peut partager.

Numéro d'identification. Numéro qui identifie de façon unique chacun des éléments présents dans le système. Ces éléments peuvent être des modules internes de l'architecture, des tâches, des ressources ou des messages.

Priorité de tâche. Priorité qui est assignée à une tâche lors de son démarrage. Cette priorité reste fixe tout au long de l'exécution de la tâche.

Priorité dynamique. Priorité associée à une tâche qui reflète le niveau d'urgence de la tâche à agir. Cette valeur peut donc être modifiée tout au long de l'exécution d'une tâche en fonction des conditions courantes.

Ressource. Unité fonctionnelle du robot. Une ressource peut être purement logicielle (un regroupement d'algorithmes de vision) ou elle peut représenter une partie matérielle du robot, soit un capteur ou actionneur (ex. caméra, contrôle des moteurs).

Service. Travail qu'un robot peut accomplir sur demande. Un service est rendu par l'exécution de la tâche y étant associée.

Tâche. Ensemble d'actions exécutées par un robot dans le but de fournir un service. Une tâche peut être complète en soi ou encore composée d'un ensemble de sous-tâches pouvant s'exécuter de façon simultanée ou consécutive.

Tâche-parent. Tâche qui a créé la tâche courante et qui est responsable de sa gestion.

Tâche distante. Tâche dont la tâche-parent s'exécute sur un autre robot.

Tâche locale. Tâche dont la tâche-parent s'exécute sur le même robot.

CHAPITRE 1

INTRODUCTION

La robotique est un domaine qui est présentement en pleine effervescence. En ce début de millénaire, un nombre toujours croissant de chercheurs s'attaquent aux problèmes liés au développement de machines de plus en plus intelligentes pouvant non seulement survivre dans notre monde changeant et imprévisible, mais aussi y être utiles. Conséquemment, la quantité de robots en activité augmente constamment et le travail en équipe devient nécessaire. Certains problèmes ne peuvent tout simplement pas être résolus par un robot seul. Le travail d'équipe permet théoriquement aux robots d'accomplir leurs tâches plus rapidement. En outre, le développement de plusieurs robots plus simples est souvent moins coûteux que si on avait voulu développer un seul robot omnipotent. Il est donc important de s'attarder au problème de la coopération entre les robots afin qu'ils puissent interagir de la façon la plus efficace possible dans un monde où ils seront de plus en plus présents.

Tout dépendant de la tâche à effectuer, le degré de coopération nécessaire peut varier. Certaines tâches, comme l'exploration d'une région, ne demandent qu'une interaction faible entre les robots, alors que d'autres tâches, comme le déplacement d'une poutre, demandent une collaboration étroite entre les parties. Jusqu'à tout récemment, la plupart des robots développés l'étaient dans le but de résoudre des tâches bien précises. Par conséquent, l'architecture du logiciel de contrôle était pensée en fonction du type de tâches à accomplir. Cependant, les robots des générations futures auront les capacités nécessaires pour réaliser un nombre élevé de tâches qui, fort probablement, demanderont différents niveaux de coopération. C'est pourquoi on voit ces dernières années des projets de recherche visant à développer dès maintenant une architecture logicielle permettant la coopération à divers niveaux au sein d'un groupe de robots hétérogènes.

C'est précisément cette problématique qui est étudiée dans le cadre de ce projet. Ce travail propose une architecture permettant à des groupes de robots hétérogènes de coopérer afin de résoudre des tâches demandant différents niveaux d'interaction. L'architecture développée, que l'on nommera *AIDER*, pour Architecture pour l'Interaction Dynamique Entre Robots, se veut une architecture flexible qui permet le développement rapide de nouvelles tâches multi-robots tout en laissant au développeur le plus de liberté possible quant à sa façon d'aborder sa tâche. Elle offre des mécanismes permettant la coopération entre robots à tous les niveaux, du partage des ressources aux prises de décision, en passant par le suivi des tâches actives. Le terme architecture est utilisé dans ce travail dans un sens large, incluant à la fois les méthodes d'interactions entre les robots et tout l'environnement logiciel de développement qui s'y rattache.

Afin de valider l'architecture proposée, un scénario d'exploration planétaire est utilisé. Le développement de ce scénario permet de vérifier l'utilité réelle de l'architecture et de tester ses différentes fonctionnalités.

Ce document présente d'abord au chapitre 2 une revue des travaux connexes déjà effectués par d'autres équipes de chercheurs. Le chapitre 3 énonce ensuite les objectifs du projet, puis présente une description de l'architecture développée. Les détails du scénario d'évaluation sont donnés au chapitre 4, suivis au chapitre 5 d'une discussion portant sur les résultats obtenus. Le chapitre 6 présente la conclusion découlant du travail réalisé.

CHAPITRE 2

APPROCHES POUR LE CONTRÔLE DE GROUPES DE ROBOTS

La recherche sur les groupes de robots n'est pas aussi récente qu'on pourrait le croire. En effet, déjà dans la première moitié du 20e siècle, Grey Walter, en compagnie de Wiener et Shannon, étudiait un groupe de robots démontrant des comportements sociaux complexes [Dorf, 1990]. Ce n'est cependant pas avant les années 80 que l'on a vu ce domaine attirer plus d'attention. Cette section présente les principaux travaux qui ont marqué la recherche sur la coopération entre robots.

Voyons tout d'abord les travaux qui sont considérés comme les pionniers de la coopération multi-robots. L'équipe de Fukuda a été l'une des premières à s'attaquer aux groupes de robots avec CEBOT [Fukuda et Kaga, 1997], un système robotique cellulaire formé de robots simples pouvant se reconfigurer en fonction de la tâche à accomplir. Ce système utilisait une communication simple entre les modules. À la même époque, l'équipe de Beni ouvrait la voie aux colonies de robots démontrant une intelligence de groupe sans avoir à communiquer entre eux [Beni, 1988]. La première architecture dédiée à la coopération dans un groupe de robots mobiles et qui a eu un impact majeur sur la recherche fut ACTRESS [Asama et coll., 1989]. Dans ce système, chaque robot était doté d'un potentiel chiffré pour accomplir une tâche, et lorsqu'une tâche à accomplir demandait plus de points que le potentiel d'un robot, d'autres robots s'ajoutaient jusqu'à ce que leur potentiel total soit suffisant pour accomplir la tâche.

Depuis, de nombreuses équipes de chercheurs ont proposé des approches qui varient, entre autres, au niveau du degré d'interaction entre agents et au niveau de la communication. Les sections suivantes résument les travaux accomplis par ces équipes.

2.1 Approches avec peu ou pas de communication

Un premier groupe a choisi de travailler sur la coopération dans un groupe de robots où il n'y a pas de communication explicite entre eux. Werger [1999] propose un modèle basé sur l'approche comportementale minimaliste. Les efforts de conception de son système sont concentrés en grande partie sur la recherche de l'ensemble minimal de comportements qui permet d'accomplir les objectifs. Il veut, par une approche de développement rigoureuse, démontrer qu'un système purement comportemental peut accomplir beaucoup plus que le croient la plupart des chercheurs qui utilisent majoritairement des approches hybrides. La coopération entre les robots se fait de façon totalement implicite, les comportements d'équipes émergeant de l'assemblage des comportements de base. Werger a utilisé cette approche afin de développer une équipe de robots pour participer à la compétition RoboCup [Kitano et coll., 1997].

Pagello et coll. [1999] présentent également une architecture sans raisonnement de haut niveau pour un groupe de robots participant à RoboCup. L'activation des différents comportements se fait en fonction de la position des équipiers et des adversaires par rapport au ballon et au but. Par exemple, une passe est effectuée lorsque le porteur du ballon voit un coéquipier plus près du but et sans adversaire à ses côtés.

Des mécanismes de communication implicite sont aussi utilisés par Saito et Yamada [1999] afin de solutionner le problème du déplacement d'une boîte par plusieurs robots. Différents groupes de comportements sont créés afin d'évoluer vers la solution à partir de situations diverses. Un analyseur de situations détermine ensuite quel est l'état actuel et active le groupe de comportements associés à cette situation. Kube et Zhang [1993] utilisent une approche similaire. Ils définissent un comportement de groupe comme étant un ensemble de comportements, chacun représentant une étape de la solution. Des changements dans l'environnement sont utilisés pour déclencher la transition d'un comportement à un autre, éliminant ainsi le besoin de communication explicite. Il est intéressant de noter qu'afin de faire l'arbitrage des comportements pour une même étape, les auteurs utilisent un réseau logique adaptatif (*Adaptive Logic Network*) qu'ils entraînent à l'aide d'un simulateur ou directement sous la supervision d'un humain. Voilà un aspect qui diffère notablement de

la plupart des autres réalisations de l'approche comportementale.

Plusieurs autres groupes de chercheurs utilisent une approche dite stigmergique [Cao et coll., 1997], où un robot observe l'environnement afin de prendre ses décisions.

2.2 Approches avec communication explicite

Toutes les approches précédentes ont l'avantage d'éliminer les nombreux problèmes liés à la communication entre les robots. Elles démontrent qu'il est possible pour un groupe de robots d'accomplir des tâches demandant une certaine coordination sans communiquer directement entre eux. Cependant, ces systèmes ne s'appliquent que dans les cas où il n'y a qu'un seul but commun à tous les robots. Dans des environnements plus dynamiques, où les buts et les regroupements de robots nécessaires peuvent changer continuellement, une approche sans communication explicite ne serait pas suffisante.

Dans [Chaimowicz et coll., 2002], des robots coopèrent grâce à une assignation dynamique des rôles nécessaires pour accomplir une tâche. Lorsque de nouveaux rôles sont annoncés, les robots pour lesquels l'utilité du nouveau rôle est plus grande que l'utilité de leur rôle actuel postulent sur ces rôles. Les agents ayant la plus grande utilité pour le rôle sont sélectionnés. Trois mécanismes permettent à un robot de changer de rôle : l'*allocation*, où un robot inactif assume un nouveau rôle, la *réallocation*, lorsqu'un robot change de rôle à cause d'une plus grande utilité ou encore d'un manque de progression, et l'*échange*, où deux robots s'échangent leur rôle respectif. N'importe quel robot peut offrir de nouveaux rôles. Ainsi, un robot trouvant un objet à déplacer qui nécessite plus d'un robot diffusera un message annonçant les rôles disponibles.

ALLIANCE [Parker, 1998] est une autre architecture qui permet la coopération dans un groupe de robots hétérogènes. Cette architecture complètement distribuée a pour objectif de permettre l'accomplissement de missions composées de sous-tâches pouvant avoir des dépendances entre elles, mais ne demandant qu'une interaction large entre les robots. À la base, le système correspond à l'approche comportementale avec des comportements actifs qui s'inhibent en fonction de l'environnement détecté. Contrairement à l'approche

classique, les comportements peuvent être regroupés dans des ensembles qui sont activés ou mis en hibernation en fonction d'un comportement motivationel qui lui est associé. Deux types de motivations permettent de séparer la tâche entre les robots sans avoir besoin d'un contrôle centralisé. La première motivation, l'*impatience*, pousse un robot à prendre en charge une tâche lorsqu'il ne détecte aucune évolution dans l'accomplissement de cette tâche. D'un autre côté, le *consentement* amène un robot à abandonner sa tâche s'il s'aperçoit qu'il ne fait pas de progrès. Ces deux motivations suffisent pour s'assurer que toutes les tâches seront éventuellement accomplies, même dans le cas où certains robots seraient rendus inaptes par l'échec d'un ou plusieurs de leurs modules.

Un autre système d'allocation de tâches est proposé par [Matarić et Gerkey, 2002]. Ce système, nommé MURDOCH, utilise le principe de l'encan pour diviser les tâches entre les différents robots. Lorsqu'une entité crée une nouvelle tâche, elle émet un avis indiquant les paramètres de la tâche à accomplir. Les robots répondent ensuite en faisant une offre en fonction de leur capacité à effectuer le travail. Après un certain délai, l'initiateur de l'encan choisit celui qui a fait la meilleure offre et lui donne le contrat pour effectuer la tâche pendant un certain temps. Il surveille ensuite l'avancement du travail afin de renouveler le contrat, ou encore de l'assigner à un autre. Le fait d'accorder les contrats pour une période déterminée permet au système de réagir rapidement en cas de panne. Un des aspects intéressants dans cette architecture est le mécanisme de communication. Les messages, au lieu d'être adressés à des robots en particulier, sont associés à des capacités et à des états. Ainsi, un message peut être envoyé à tous ceux qui ont une caméra et qui sont inactifs par exemple. Cette technique assure une plus grande robustesse à l'ensemble, les robots n'ayant pas besoin de connaître tous les robots qui les entourent et n'ayant pas à maintenir à jour de l'information sur ceux-ci. MURDOCH a été implémenté sur un groupe de robots Pioneer 2-DX pour solutionner des problèmes nécessitant des tâches à interaction faible (nettoyage de zone, sentinelle) et à interaction plus forte (déplacement d'une boîte). Notons cependant que dans le cas du déplacement d'une boîte, un robot supplémentaire doit être utilisé pour coordonner le travail fait par les deux robots qui poussent la boîte.

Il est aussi important de noter que dans ses travaux, Matarić [Matarić, 1998; Matarić,

1994] préconise toujours fortement l'utilisation d'une approche comportementale plutôt qu'une approche hiérarchique ou hybride. [Matarić, 1994] présente d'ailleurs les résultats de ses travaux sur la coopération dans un groupe de robots en utilisant des comportements simples.

2.3 Architectures de coopération à plusieurs niveaux

Les approches décrites à la section 2.2 assurent une bonne coopération dans un groupe de robots principalement en répartissant les tâches de façons dynamique et robuste. Bien qu'efficaces pour régler plusieurs problèmes, ces systèmes n'offrent qu'une possibilité d'interaction au niveau des tâches, limitant le type de coopération qui peut s'exercer entre différents agents physiques. En effet, les mécanismes offerts ont pour but la répartition des tâches et ne permettent pas à deux robots d'interagir directement au niveau de leurs comportements de bas niveaux ou de leurs ressources. Au cours des cinq dernières années, des architectures permettant une interaction à plusieurs niveaux ont commencé à faire leur apparition.

Une première étape menant à une coopération complète multi-niveaux est le partage de l'information sensorielle de chacun des robots. Khoo et Horswill [2002] présentent HIVEMind, une architecture de coopération basée sur un modèle hybride de l'approche comportementale. HIVEMind permet aux robots de s'échanger leurs états délibératifs ainsi que leur information sensorielle en diffusant ces données sous forme compacte à chaque seconde. Chaque agent faisant partie du groupe est donc considéré comme un capteur virtuel amenant une information supplémentaire à l'engin d'inférence de ses coéquipiers. Ainsi, les robots prennent une décision en fonction de la perception de tout le groupe. En modifiant la fonction qui effectue la fusion des informations provenant des différents agents, différents types de comportements de groupe peuvent être créés.

Sellem et coll. [2000] proposent aussi une architecture permettant de partager des informations sensorielles. Cette architecture, qui est une extension de travaux antérieurs sur le contrôle d'un seul robot, permet à tous les membres d'un groupe de rendre accessible aux

autres leur représentation du monde. Lorsque deux robots utilisent un même format pour représenter une information, l'architecture leur permet d'accéder directement à la représentation de l'autre afin de suppléer à une incapacité ou encore pour compléter leur propre information. Afin de pouvoir aller chercher l'information dont ils ont besoin, chaque robot doit garder à jour une table de compétences, table qui indique les capacités de chacun des participants.

Au Jet Propulsion Laboratory, en Californie, une équipe travaille sur une architecture de coopération dans le but d'utiliser un groupe de robots pour l'exploration planétaire [Pirjanian et coll., 2000]. Cette architecture nommée CAMPOUT utilise une approche comportementale adaptée à un environnement multi-agents. Elle permet une grande flexibilité au niveau de la coordination des comportements grâce à l'utilisation de *Behaviour Coordination Mechanisms* qui peuvent facilement être interchangés en fonction du type de coordination nécessaire. Ces mécanismes de coordination peuvent être divisés en deux classes principales : l'arbitrage, basé sur des états ou des niveaux de priorités, et la fusion de commandes, pouvant utiliser des techniques de votes ou de fusion floue. CAMPOUT permet aussi le partage de ressources entre les différents robots. Ainsi, les comportements, les capteurs et les actionneurs d'un agent peuvent être utilisés par tous les robots du groupe.

Ces dernières approches augmentent les possibilités de coopération en permettant l'échange d'information au niveau des ressources physiques des robots. Certaines architectures vont encore plus loin en permettant une interaction à plusieurs niveaux différents du système de contrôle d'un robot. Sans vraiment définir une architecture de coopération pour un groupe de robots, Nakamura et Arai [2002] posent tout de même les fondements de l'interaction à plusieurs niveaux en proposant une approche où une requête peut être envoyée à quatre niveaux d'abstraction différents. Il est donc possible de choisir de contrôler directement un robot en accédant à son module de mouvement (niveau 1), de contrôler tout un groupe (niveau 2), de définir les manipulations à être appliquées sur un objet (niveau 3) ou bien de demander au système d'accomplir une tâche complexe de façon autonome (niveau 4). Plus le niveau augmente, plus le degré d'autonomie nécessaire sur les robots est grand. Par

exemple, demander à un robot d'avancer d'un mètre ne demande que très peu d'autonomie de sa part alors que celui-ci devra être beaucoup plus évolué pour faire partie d'un système auquel on demanderait de construire un abri. Nakamura et Arai [2002] proposent ces différents niveaux d'interaction pour l'interface humain-machine, mais il précise aussi que l'instigateur des requêtes pourrait tout aussi bien être un autre agent, physique ou non.

KAMARA, présentée dans [Laengle et coll., 1998], est une architecture de contrôle distribué conçue pour contrôler les différentes parties d'un robot évolué. Chaque élément d'un robot qui est appelé à travailler en équipe avec d'autres est représenté par un agent comprenant trois modules. Ces modules s'occupent de la communication, de la planification et de l'exécution des tâches. Un agent peut aussi contenir d'autres agents. En choisissant quels agents coopéreront pour effectuer une tâche, il est possible d'obtenir une interaction plus ou moins forte entre les parties. Cette architecture pourrait aussi s'étendre à un groupe de robots, mais certaines modifications seraient nécessaires puisque dans l'état actuel, il n'est pas possible pour un agent d'effectuer plus d'une tâche à la fois.

Simmons et coll. [2000] présentent une architecture de contrôle à trois couches (comportements, exécution, planification) permettant une interaction à chaque niveau. On peut ainsi choisir le niveau de granularité avec lequel les robots interagissent et se synchronisent. Au niveau des comportements, qui font habituellement un lien entre capteurs et actionneurs, il est possible de connecter les entrées et les sorties de ces comportements à n'importe quelle ressource, que celle-ci soit locale ou sur un autre robot. Cela permet de créer des boucles de contrôle distribué efficaces pour les tâches demandant une très forte interaction entre les robots. De plus, cette architecture n'impose pas de restriction sur le type d'information qui peut être transmis d'un comportement à l'autre. Au niveau exécution, l'architecture permet de contrôler et de se synchroniser avec des tâches étant exécutées sur un autre robot. De plus, il est possible pour tous les membres du groupe de faire une requête de tâche directement à un autre robot. L'interface qui permet de contrôler la couche d'exécution est basée sur le PRL (*Plan Representation Language*) [Simmons et coll., 2000], alors que l'implémentation de la couche elle-même est faite à partir du TDL (*Task Description Language*) [Simmons et Apfelbaum, 1998]. Ces deux langages permettent d'ajouter très

facilement de nouveaux comportements au robot. Finalement, la couche de planification permet aux robots de se répartir les tâches en utilisant une approche de marché [Zlot et coll., 2002]. Lorsqu'un agent désire faire exécuter une tâche, il fait une offre d'achat sur le marché. Les robots pouvant fournir le service répondent alors en indiquant le prix qu'ils demandent pour effectuer le travail. Le demandeur choisit ensuite le coût le plus bas, à condition que ce prix soit inférieur au coût s'il effectuait lui-même la tâche. Il est aussi possible d'avoir des leaders qui effectuent des transactions plus complexes impliquant plusieurs agents et plusieurs tâches. Cette dernière possibilité permet d'obtenir une répartition plus efficace des tâches que dans le cas où le marché est limité aux échanges simples.

Afin de faire une planification efficace, il est utile de savoir quel type de robots nous entourent et quelles sont leurs capacités. Pour ce faire, Simmons et coll. [2000] ont développé la notion de serveurs de capacités. Chaque robot garde en mémoire les capacités de tous les autres robots à proximité et fait appel à cette base de données lorsque nécessaire. Afin d'avoir la certitude d'avoir une information à jour, même dans le cas de problèmes intermittents de communication, chaque robot diffuse régulièrement ses capacités afin que tous ses coéquipiers mettent à jour leur base de données. Cela permet aussi de détecter l'échec d'un robot si les capacités de ce dernier n'ont pas été reçues depuis un certain temps.

En examinant ces travaux, nous pouvons constater une évolution vers des architectures qui permettent une flexibité de plus en plus grande dans l'interaction entre robots. Comme les archictectures de coopération sont reliées de très près à l'architecture d'un robot seul, la prochaine section présente certains travaux sur l'autonomie d'un robot qui apportent des fonctionnalités qui sont intéressantes dans un contexte de coopération multi-agents.

2.4 Concepts applicables à une architecture multi-robots

Mackenzie et coll. [1997] proposent la notion d'*assemblage*, un ensemble de comportements de base et de mécanismes de coordination regroupés afin de pouvoir être traités comme une primitive de haut niveau par le développeur qui veut définir une mission pour un robot. Une mission consiste alors en un ordonnancement séquentiel d'*assemblages.*

L'architecture d'autonomie présentée par Alami et al. [Alami et coll., 1998] propose une méthode intéressante pour l'accès aux ressources du robot. Elle décentralise le contrôle d'accès en associant à chaque ressource un module qui est responsable de traiter les requêtes d'accès et de fournir les rapports d'exécution ainsi que les données pertinentes à cette ressource. Le module n'a aucune connaissance de ses clients au départ puisque la relation client-serveur est créée de façon dynamique lorsqu'une tâche doit utiliser une ressource. L'architecture n'offre cependant pas beaucoup de flexibilité pour gérer les conflits lors d'accès simultanés par plusieurs tâches. En effet, dans ce cas, les conflits sont toujours résolus en donnant priorité à la dernière requête reçue, ce qui n'est pas souhaitable pour tous les types de ressources.

CLARAty [Volpe et coll., 2001] est un exemple d'architecture permettant une grande flexibilité pour le développeur. C'est une architecture à deux couches où la planification et l'exécution, deux niveaux distincts dans les modèles en couches standards, sont fusionnés. La couche fonctionnelle est entièrement conçue avec une approche orienté-objets. Chacun des objets dérive d'une même classe qui contient des fonctionalités de base permettant entre autres d'interagir avec d'autres objets ou avec la couche de décision. Les objets forment un arbre hiéarchique représentant le robot (par exemple, un robot muni de deux bras et d'une caméra, chaque bras étant lui-même muni de trois moteurs) et il est possible d'accéder directement à n'importe quel niveau dans la hiérarchie. Cette conception permet d'effectuer des simulations à différents niveaux en remplaçant la branche de l'arbre à simuler par un objet de simulation.

La couche de décision est, pour sa part, constituée au sommet d'un arbre de buts représentant l'objectif de haut niveau. Ces buts sont ensuite subdivisés en tâches, puis en commandes. C'est au niveau des noeuds de commandes que s'effectue l'interaction avec la couche fonctionnelle. CLARAty permet que ces commandes soient de haut niveau, comme demander d'aller à un endroit particulier, ou de bas niveau, comme demander de tourner le segment 2 du bras droit de 15 °. Cela laisse toute la flexibilité au développeur quant à la répartition de la tâche entre le niveau décisionnel et le niveau fonctionnel.

L'augmentation de la puissance de calcul disponible et des capacités de communication

est une des raisons qui rendent aujourd'hui possible plusieurs des fonctionnalités proposées dans les différentes architectures qui ont été présentées. Ces nouvelles capacités retirent certaines contraintes qui limitaient les choix possibles pour le développeur. Cela a mené au cours des dernières années à des systèmes robotiques de plus en plus flexibles. Cependant, ce type d'architecture de coopération n'en est encore qu'à ses débuts et plusieurs concepts intéressants se retrouvent éparpillés dans différentes architectures isolées. La recherche dans le domaine de coopération dans un groupe de robots est donc rendue à un point où il est possible et souhaitable de s'attaquer à la question énoncée par Parker [Parker, 2000], à savoir s'il est possible de développer une architecture plus générale pouvant être facilement adaptée pour supporter un plus grand éventail de systèmes multi-robots.

CHAPITRE 3

ARCHITECTURE *AIDER*

3.1 Objectifs

Le présent projet de recherche a pour but de répondre aux problèmes du développement d'une architecture d'interactions multi-robots générique permettant à un groupe de robots d'accomplir des tâches de nature variée. L'objectif est donc de développer une architecture qui apporte flexibilité et robustesse aux systèmes de contrôle de groupes de robots. Le but premier n'est pas d'implémenter de nouveaux algorithmes de répartition des tâches ou de nouveaux types de comportements, mais bien d'offrir une architecture logicielle offrant des mécanismes innovateurs qui permettront de développer rapidement de nouveaux systèmes plus polyvalents. Autant les méthodes d'interactions entre les robots que l'environnement logiciel de développement sont inclus dans ce qui est désigné tout au long de ce document par le terme *architecture*.

L'architecture développée doit :

- offrir un engin qui permet le contrôle de tâches réparties sur plusieurs robots ;
- permettre le développement et la réutilisation de plusieurs types de contrôleurs de tâches ;
- permettre l'interaction entre robots autant au niveau de la planification que de l'exécution de bas niveau ;
- permettre le partage des ressources (capteurs, actionneurs, algorithmes) entre robots ;
- offrir un mécanisme de communication inter-robots ;
- permettre le développement et la réutilisation de plusieurs types de contrôleur d'accès aux ressources ;
- permettre au robot de connaître les capacités des autres membres de son groupe.

Les possibilités de coopération dans un groupe de robots hétérogènes sont liées de près au système de contrôle de chacun des robots pris de façon isolée et c'est pourquoi le système développé propose une solution complète qui touche autant l'autonomie d'un robot seul que ses mécanismes de coopération.

3.2 Lignes directrices

Il y a de nombreuses façons de s'attaquer à la problématique proposée. Il faut tout d'abord faire un choix au niveau de l'approche de contrôle pour un robot. L'approche comportementale permet le développement de systèmes simples et robustes. D'un autre côté, l'approche multicouches permet généralement une plus grande flexibilité dans les tâches qu'elle permet d'accomplir. Comme l'objectif est de bâtir une architecture qui s'adapte à de nombreuses utilisations, une approche hybride est préconisée afin d'allier simplicité et flexibilité en permettant la création de comportements simples, respectant l'idéologie comportementale pure, ainsi que des arbres de tâches hiérarchiques plus complexes.

Dans toute approche utilisant des comportements, il est nécessaire d'avoir un mécanisme d'arbitrage des commandes provenant des divers comportements s'exécutant simultanément. Différents schèmes peuvent être utilisés pour déterminer comment combiner les différentes commandes, comme par exemple l'inhibition ou l'addition. Encore une fois, afin de permettre le plus de liberté possible dans les systèmes qui sont développés, l'architecture favorise l'utilisation de contrôleurs d'accès indépendants qui sont associés directement aux ressources. Il est ainsi possible de changer le type d'arbitrage en assignant à une ressource un contrôleur d'accès différent. De plus, un nouveau mécanisme de priorités dynamiques est développé afin de permettre au système de s'adapter rapidement à de nouvelles situations.

Un autre des choix importants qui doivent être faits lorsqu'on développe une architecture d'interactions entre robots est le type de communication inter-robots qui est permis. Certaines architectures permettent une communication entre robots au niveau de la planification uniquement, d'autres aux niveaux des comportements et d'autres encore permettent de communiquer directement avec les ressources d'un autre robot. Dans le cas qui

nous intéresse, le système permet l'ensemble de ces communications, mais oblige toutes les communications à passer par un même point d'entrée. Ce dernier détail a été choisi afin d'éviter de se retrouver avec un système de communication où l'information circule de plusieurs façons différentes, ce qui serait difficile à gérer. Une attention particulière est portée sur ce point puisqu'en enlevant le plus de restrictions possibles quant aux communications, on augmente aussi les risques d'une utilisation non optimale, voire mauvaise, de ces mécanismes.

Les trois choix architecturaux qui viennent d'être présentés forment la base sur laquelle l'architecture proposée est construite.

3.3 Description de l'architecture

L'architecture proposée est nommée *AIDER*, pour Architecture pour l'Interaction Dynamique Entre Robots. À la base, un robot développé avec *AIDER* n'exporte que trois services : requête de capacités, requête de tâche et requête de ressource.

La *requête de capacités* permet à tout agent externe d'obtenir les informations concernant les différentes capacités du robot. Les *requêtes de tâche* et *requêtes de ressource* permettent ensuite de faire appel à ces capacités, que ce soit l'exécution d'une tâche ou l'accès à une ressource tel un capteur, un actionneur ou encore un algorithme. Ensemble, ces trois services permettent d'implémenter la grande majorité des fonctionnalités offertes par les architectures de coopération à plusieurs niveaux présentées à la section 2.3, que l'on pense par exemple au partage des comportements, capteurs et actionneurs de CAMPOUT [Pirjanian et coll., 2000] ou encore à l'échange de capacités tel que proposé par Simmons et coll. [2000] ou Sellem et coll. [2000]. *AIDER* permet donc d'intégrer les fonctionnalités qui correspondent à l'état de l'art dans le domaine de la coopération multi-robots.

Trois modules sont utilisés afin de faire la gestion des services offerts : le gestionnaire de capacités, le gestionnaire de tâches et le gestionnaire de ressources. Ces trois gestionnaires principaux utilisent les services d'un quatrième module, le gestionnaire de messages, qui est responsable de la communication entre les différents éléments actifs sur un ou plusieurs

robots. La figure 3.1 présente cette structure de base. Les possibilités qu'offrent ces modules sont décrites dans les prochaines sous-sections.

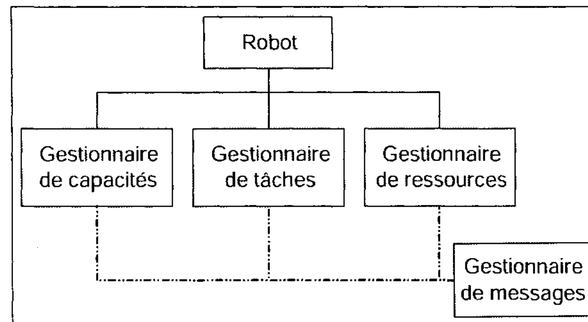

Figure 3.1 – Structure de base d'*AIDER*

3.3.1 Gestion des capacités

Le gestionnaire de capacités est responsable de tenir à jour les capacités d'un robot et de tous ses coéquipiers. Au démarrage d'un robot, la liste de ses capacités est créée en fonction des tâches qu'il peut exécuter et des ressources qu'il peut partager. La section 3.3.5 décrit comment sont spécifiées les différentes capacités d'un robot.

Pour chaque tâche présente dans la liste des capacités, les paramètres pouvant lui être passés sont précisés sous forme de plages de valeurs possibles. Afin de garder la configuration d'un robot le plus simple possible, seulement trois types de paramètres sont acceptés : les nombres entiers (*integer*), les nombres à point flottant (*float*) et les chaînes de caractères (*string*). Chacun des paramètres délimités par ces plages peut être désigné comme optionnel ou obligatoire. Notons qu'à travers toute l'architecture, les paramètres sont désignés par un nom, et non par un ordre précis dans une liste. Cela permet d'ajouter facilement de nouveaux paramètres, tout en gardant une compatibilité avec le code antérieur qui ne supporte pas ces nouveaux paramètres.

En plus de ces plages de paramètres, une capacité est aussi liée à une liste de ressources qui lui sont nécessaires. Ainsi, lorsqu'une ressource n'est pas disponible, toutes les tâches nécessitant cette ressource sont retirées de la liste des capacités. Cette fonctionnalité permet d'avoir une liste de capacités qui se met à jour dynamiquement lorsque survient un

changement dans la disponibilité des ressources. Il est donc possible grâce à ce mécanisme de réagir de façon élégante aux pannes matérielles pouvant survenir sur le robot. En effet, dans un tel cas, le robot, ainsi que tous ses coéquipiers, connaît automatiquement les tâches qu'il ne peut plus accomplir et n'entreprendra pas d'activités vouées à l'échec.

Nous venons tout juste de préciser que les coéquipiers d'un robot sont automatiquement mis au courant lorsque ce dernier ne peut plus effectuer certaines tâches. Cela est possible grâce aux règles suivantes, qui définissent les échanges de capacités à travers le réseau de robots. À sa création, et lors de tout changement dans ses capacités, un robot a le devoir de diffuser sa liste de capacités afin que tous ses coéquipiers puissent tenir à jour continuellement l'information dans leur base de données. De plus, afin d'assurer une plus grande robustesse dans le cas où le médium de communication ne serait pas fiable, il est possible de configurer le gestionnaire pour qu'il diffuse ses capacités à intervalles réguliers.

La diffusion des capacités se fait sous la forme d'un message contenant la liste de toutes les tâches pouvant être accomplies par le robot. Ce message précise aussi l'étendue des paramètres acceptés par chacune des tâches. Le contenu complet d'une capacité est présenté à la figure 3.2. On y retrouve, entre parenthèses, un exemple pour chacun des paramètres.

Capacité

- Type de service (MoveInLine)
- Niveau de confiance (50)
- Classe d'implémentation (robot.tasks.MoveInLine)
- Liste des paramètres
  - Nom du paramètre (Distance)
  - Type du paramètre (Float)
  - Valeur maximale (200)
  - Valeur minimale (-200)
  - Paramètre obligatoire (Oui)

Figure 3.2 – Contenu d'une description de capacité

3.3.2 Gestion des tâches

Le noyau du fonctionnement d'un robot développé selon *AIDER* est un arbre hiérarchique de tâches, comme le montre la figure 3.3. Le terme *tâche* est ici utilisé dans un sens générique puisqu'un noeud de l'arbre peut aussi bien être un simple comportement qu'un ensemble de comportements ou un coordonnateur de comportements. Le tableau 3.1 présente différents exemples de tâches.

L'arbre de tâches de *AIDER* a été bâti à partir de *AARVARC* [Dupuis et L'Archevêque, 2003], un arbre de tâches développé à l'Agence spatiale canadienne. Cet arbre de tâches simple a cependant été complètement rebâti au cours du développement. Une des caractéristiques majeures de *AIDER* est de permettre à l'arbre de tâches de développer des branches autant localement que sur des robots distants, tel que le présente la figure 3.4.

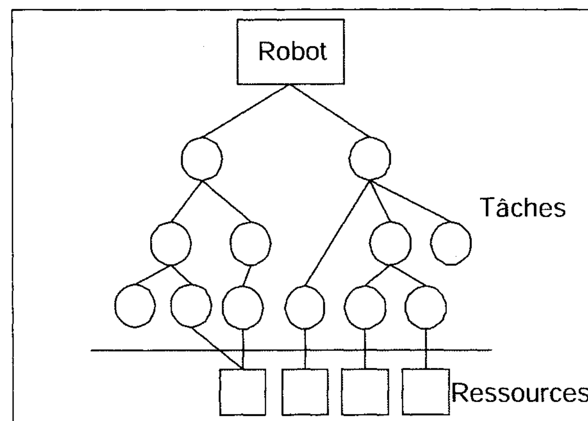

Figure 3.3 – Arbre de tâches

TABLEAU 3.1 – Exemples de tâches

Tâche de haut niveau	Exploration d'une zone
Tâche de bas niveau	Avance en ligne droite
Coordonnateur	Séparation des tâches par encan
Tâche de sécurité	Détection d'inclinaison dangereuse

Un arbre de tâches est contrôlé par un gestionnaire de tâches, et un tel gestionnaire est

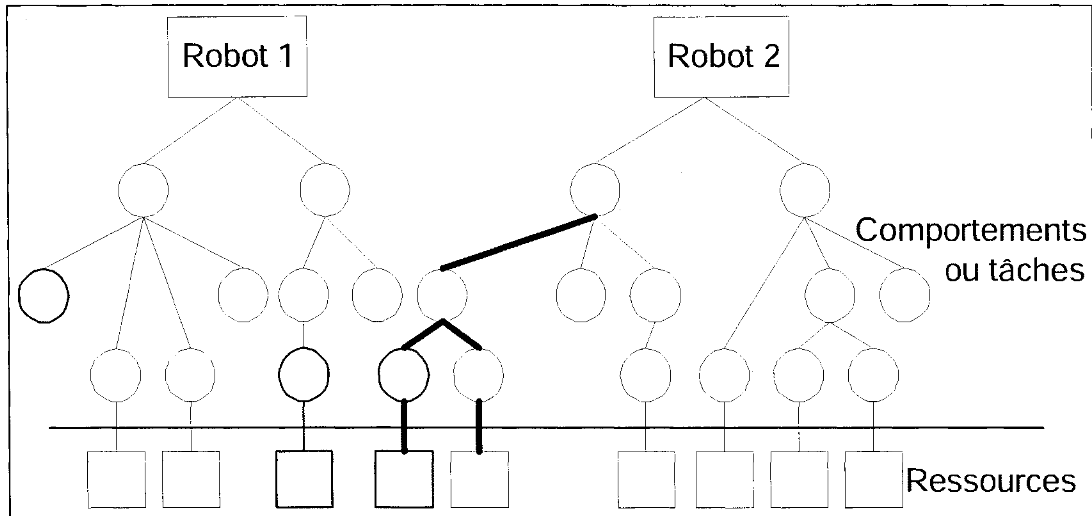

Figure 3.4 – arbre de tâches distribuées

présent sur chacun des robots. Le rôle premier du gestionnaire de tâches est de fournir le service de création de tâches. Toute nouvelle tâche est automatiquement ajoutée à l'arbre de tâches actives que ce gestionnaire a la responsabilité de gérer. Ce service peut être utilisé autant par une tâche active localement que par une tâche active sur un autre robot par le biais d'une *requête de tâche.* C'est ce qui permet d'avoir un arbre de tâches réparti sur plusieurs robots. Les détails sur le démarrage d'une tâche sont présentés plus loin dans cette section.

Une tâche peut être dans six états différents : initialisé, actif, en pause, terminé avec succès, terminé avec échec ou annulé, selon les transitions montrées à la figure 3.5. *AIDER* fournit six points d'entrée, sous forme de fonctions, qui peuvent être utilisés pour modifier le comportement de la tâche lorsqu'il y a changement d'état. Les six fonctions qui peuvent être implémentées à cet effet sont :

- myInit() : Appelée à l'initialisation de la tâche.
- myPausing() : Appelée lors de la réception d'une requête de pause.
- myPaused() : Appelée lorsque le tâche entre dans l'état *en pause.*
- myResume() : Appelée lors de la réception d'une requête de remise en marche.
- myStopping() : Appelée lors de la réception d'une requête d'arrêt.
- myTerminate() : Appelée lorsque la tâche est complètement arrêtée.

Seul le créateur d'une tâche peut contrôler l'état de cette dernière, à l'exception du cas où le parent d'une tâche n'est plus accessible (par exemple un parent sur un autre robot avec lequel on a perdu la communication pendant une trop longue période). Lors de la création d'une tâche, le parent lui assigne une priorité qui pourra par la suite être utilisée par le système pour déterminer s'il accepte une autre tâche ou encore s'il accorde l'accès à une ressource.

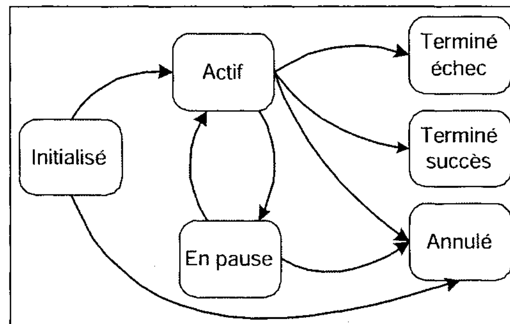

Figure 3.5 – Transition entre les états d'une tâche

L'interface de contrôle d'une tâche offre trois types de fonctions : les fonctions de contrôle, les fonctions d'information et les fonctions d'inscription aux événements. Les fonctions de contrôle de tâches sont les suivantes :

- start() : Démarre la tâche.
- stop() : Arrête la tâche de façon permanente.
- pause() : Arrête la tâche temporairement.
- resume() : Redémarre une tâche qui avait été mise en pause.
- waitForChild() : Attend que la tâche soit complétée avant de poursuivre l'exécution.

Ces fonctions permettent de contrôler l'état de la tâche et d'attendre que ses sous-tâches se terminent. On retrouve ensuite les fonctions d'information qui permettent d'obtenir les principales informations sur une tâche, soit son état, sa valeur de retour, son type, ses paramètres, etc. Ces fonctions sont les suivantes :

- getCompletionCode() : Obtient l'état de succès de la tâche (réussie ou échoué).
- getState() : Obtient l'état courant de la tâche.

- getReturnValue() : Obtient la valeur de retour de la tâche (spécifique à chaque tâche).
- getId() : Obtient l'identificateur unique de la tâche.
- getBehaviourName() : Obtient le nom de la tâche.
- getServiceType() : Obtient le type de service effectué par la tâche.
- getParameters() : Obtient les paramètres de la tâche.
- getChildList() : Obtient la liste des sous-tâches de la tâche.
- getLastTimeRun() : Obtient le dernier moment auquel la tâche a été active.
- getPriority() : Obtient la priorité de la tâche.

Le troisième type de fonctions fournies par l'interface d'une tâche est le groupe de fonctions d'inscription aux événements. Ces fonctions permettent de s'inscrire auprès d'une tâche afin d'être informé lorsque son état ou sa valeur de retour change. Ces fonctions sont les suivantes :

- addStateChangeListener() : Permet de s'inscrire pour être avisé lorsque l'état de la tâche change.
- removeStateChangeListener() : Retire une inscription qui avait été faite à l'aide de la fonction addStateChangeListener.
- addCompletionCodeListener() : Permet de s'inscrire pour être avisé lorsque l'état de succès de la tâche change.
- removeCompletionCodeListener() : Retire une inscription qui avait été faite à l'aide de la fonction addCompletionCodeListener.

Lorsque qu'une tâche active ou un opérateur désire démarrer une nouvelle tâche, la première étape consiste à effectuer une *requête de tâche* auprès du gestionnaire de tâches en spécifiant le type de service demandé, les paramètres pour ce service, la priorité, ainsi que, optionnellement, le numéro d'identification du robot sur lequel on veut que la tâche s'exécute.

Dans le cas d'une requête pour une tâche locale, le gestionnaire de tâches vérifie alors que le service demandé est bien dans la liste des capacités du robot et que les paramètres demandés sont valides, i.e. s'ils sont à l'intérieur des bornes définies par la capacité. Ensuite, le gestionnaire de tâches initialise la nouvelle tâche, puis il remet une interface à

cette tâche à l'appelant. Cette interface est en fait une référence directe à l'objet *tâche* nouvellement créé qui permet à la tâche-parent de contrôler cette tâche via les fonctions décrites précédemment.

Dans le cas d'une requête pour une tâche distante, le gestionnaire de tâches local transmet la requête au robot spécifié dans celle-ci par le biais d'un message. Le gestionnaire de tâches distant procède alors aux mêmes étapes de vérification et d'initialisation que lors d'une tâche locale. Cependant, à ce moment, le gestionnaire distant ne peut remettre l'interface de la nouvelle tâche à l'appelant puisqu'ils ne sont pas sur le même robot. Il remet alors l'interface à une tâche spéciale de contrôle des tâches distantes et renvoie le numéro d'identification de la nouvelle tâche au gestionnaire de tâches local (celui qui avait reçu la requête initiale de l'appelant), encore une fois par le biais d'un message. Ce dernier crée alors un représentant local de la tâche distante, dont le seul rôle est de transmettre de façon transparente les commandes et leurs résultats entre l'appelant local et la tâche distante. La figure 3.6 résume les principales étapes du lancement d'une tâche distante.

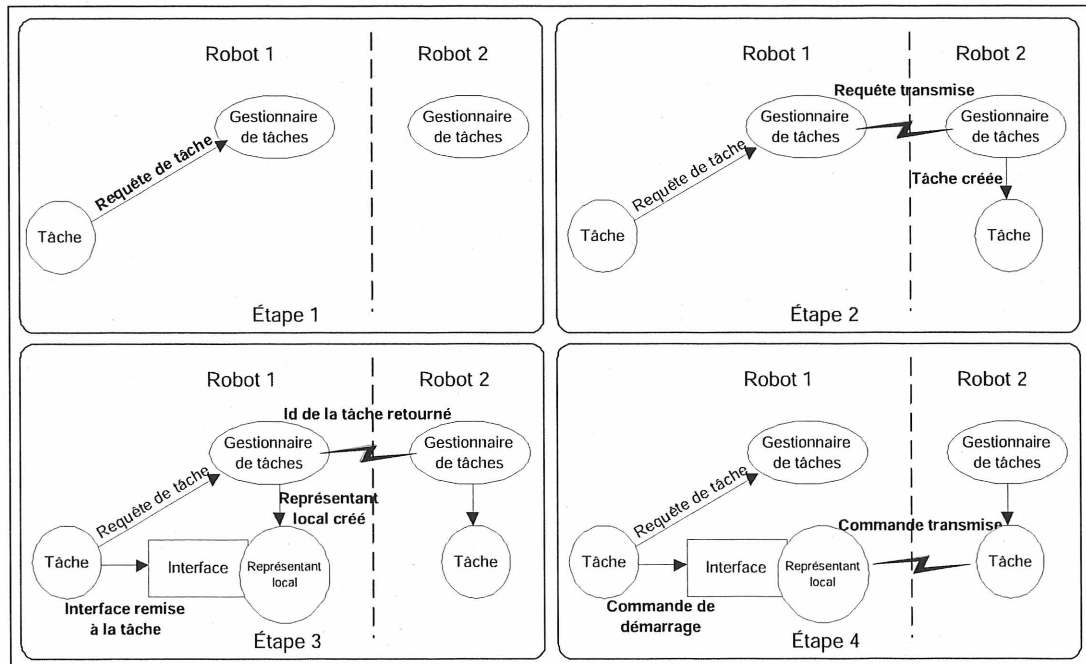

Figure 3.6 – Étapes du lancement d'une tâche distante

L'un des objectifs de *AIDER* est de permettre le développement rapide et facile de nouvelles

tâches. Une attention particulière a donc été portée à la quantité de travail à effectuer lors de la création d'une tâche. Le développeur d'une tâche crée d'abord une nouvelle classe qui hérite d'une tâche de base qui n'effectue aucun travail. Il redéfinit ensuite la fonction qui représente le corps de la tâche, y programmant le travail de la tâche qu'il crée. Pour plus de contrôle, le développeur peut aussi décider de redéfinir les fonctions qui sont appelées lors de l'initialisation de la tâche (*myInit*), lorsqu'elle est mise en pause (*myPausing*, *myPaused* et *myResume*) ou arrêtée (*myStopping*), ou lorsque celle-ci se termine (*myTerminate*).

L'exécution d'une tâche demande, dans la plupart des cas, l'utilisation de plusieurs comportements en parallèle ou encore un ordonnancement de comportements. *AIDER* préconise dans ces cas la création d'une tâche de coordination. C'est à cette tâche de coordination que revient la responsabilité de démarrer tous les comportements nécessaires à l'atteinte de l'objectif. Cette façon de faire permet en même temps de faciliter l'activation ou la désactivation d'ensembles de comportements en envoyant une seule commande à la tâche de coordination. Ces tâches de coordination sont en général les tâches de plus haut niveau dans le système.

3.3.3 Gestion des ressources

Les ressources présentes sur un robot, comme par exemple ses capteurs ou ses moteurs, sont à la base de toutes ses capacités. En rendant ces ressources accessibles à tous les autres robots d'un groupe, *AIDER* permet une interaction très forte entre les robots. Il est même possible pour un robot d'être entièrement contrôlé par le logiciel s'exécutant sur un autre robot.

Il existe deux principaux types de ressources : les ressources associées à des éléments matériels du robot, et les ressources algorithmiques. On retrouve dans le premier groupe des ressources telles que le contrôle des moteurs, la télémétrie ou encore un système de vision. Le deuxième groupe est quant à lui composé de ressources uniquement logicielles comme par exemple un générateur de trajectoire ou un algorithme de reconnaissance de formes dans une image.

L'accès aux ressources se fait en premier lieu à travers le gestionnaire de ressources. Ce

gestionnaire est responsable de la gestion de toutes les ressources actives sur un robot. Il détient l'ensemble des interfaces vers les différentes ressources. Chacune des ressources est créée au démarrage du robot en fonction des entrées présentes dans le fichier de configuration du robot (voir la section 3.3.5 pour plus de détails sur la configuration d'un robot). Le gestionnaire de ressources initialise dès cet instant chacune des ressources, ce qui permet par exemple de vérifier que le dispositif matériel est bel et bien présent et en état de fonctionner. Cependant, la ressource sera uniquement démarrée lorsque, pour la première fois, une tâche effectuera une requête pour obtenir une interface à cette ressource.

Lorsqu'une tâche, locale ou non, désire accéder à une ressource, elle doit faire une requête au gestionnaire de ressources afin d'obtenir une interface. Elle précise dans cette requête le type de la ressource désirée, son index, ainsi que, optionnellement, le numéro d'identification du robot cible. L'index de la ressource permet de différencier entre elles plusieurs ressources d'un même type. Une valeur de «0» pour cet index indique que l'on désire une interface à la première ressource du bon type qui est trouvée. Dans le cas où un numéro d'identification du robot cible est donné, celui-ci indique quel robot devra fournir la ressource.

Dans le cas où une ressource locale est demandée, le gestionnaire de ressources vérifie d'abord la présence de la ressource, puis son état. Si celle-ci n'est pas active, il procède à son démarrage. Si à ce stade, la tâche est bien à l'état «actif», le gestionnaire remet alors à la tâche appelante une interface vers la ressource demandée. Une erreur est retournée lorsque la ressource n'a pas été activée avec succès.

Dans le cas où la ressource demandée est située sur un robot distant, le gestionnaire de ressources local communique avec le gestionnaire de ressources du robot cible afin de vérifier l'état de la ressource demandée. Après avoir démarré la ressource (si nécessaire), le gestionnaire distant retourne au gestionnaire local le numéro d'identification de la ressource. Le gestionnaire local utilisera ensuite ce numéro d'identification afin de créer un représentant local de la ressource distante. La tâche de ce représentant local est de communiquer avec la ressource distante afin de lui transmettre les commandes ou de récupérer les données qu'elle fournit. Finalement, le gestionnaire de ressources local remet à la tâche appelante

l'interface vers la ressource. Une fois la requête de ressource terminée avec succès, la tâche utilise les méthodes de l'interface qui lui a été retournée pour contrôler la ressource ou accéder à ses données. La figure 3.7 résume les principales étapes de la requête d'accès à une ressource distante.

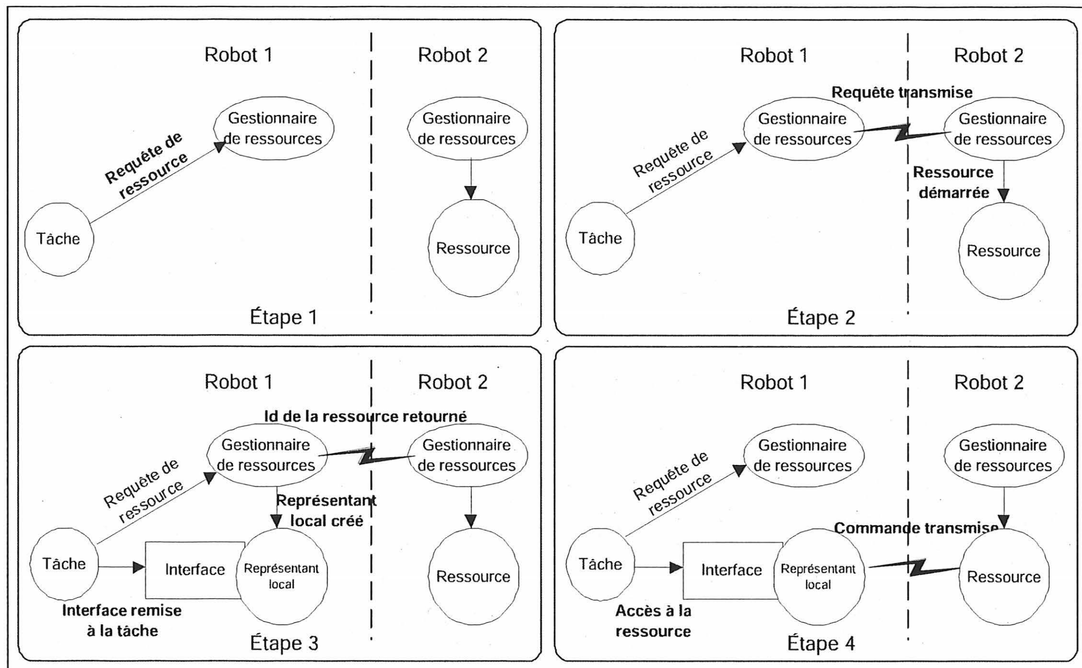

Figure 3.7 – Étapes de la requête d'accès à une ressource distante

Lorsqu'un gestionnaire de ressources remet une interface à une tâche, il peut remettre différents types d'interface. Il doit en effet vérifier si une interface spécifique est indiquée dans la configuration de la ressource. Si c'est le cas, c'est une instance de cette interface spécifique qui est remise à la tâche appelante. Dans le cas où aucune interface n'est spécifiée, le gestionnaire retourne alors à la tâche une interface générique. L'interface générique n'offre qu'une seule fonction qui permet d'envoyer des commandes à la ressource. Cette fonction, *executeRequest()*, permet de spécifier le nom de la commande à effectuer ainsi que les paramètres de la commande sous la forme d'un tableau. La figure 3.8 présente un exemple d'utilisation d'une interface générique.

De l'autre côté, l'interface spécifique permet d'écrire le code de façon plus claire et plus concise en fournissant un ensemble de fonctions propres au type de ressource qu'elle re-

```
Object param[] = new Object[2];
param[0] = new Integer(3);
param[1] = new Float(34.2);
Float result = res.executeRequest( "test", param );
```

Figure 3.8 – Exemple d'utilisation d'une interface générique

```
Float result = res.test( new Integer(3), new Float(34.2) );
```

Figure 3.9 – Exemple d'utilisation d'une interface spécifique

présente. Par exemple, une interface de type `RobotControl` pourrait fournir les fonctions *setSpeed()* et *stop()*. Cela permet d'écrire l'équivalent du code de la figure 3.8 d'une façon plus conviviale, tel que montré à la figure 3.9. Ces deux méthodes sont totalement équivalentes au niveau de l'exécution, la différence étant uniquement au niveau de la lisibilité du code.

L'utilisation de ces interfaces, qu'elles soient génériques ou spécifiques à un type de ressource, permet de facilement changer une ressource pour une autre semblable sans affecter le reste du système, le code utilisé dans les tâches restant exactement le même. Par exemple, si différentes tâches sont programmées afin de permettre à un type de robot de se rendre à un point spécifié, il est très facile de permettre à un autre type de robot d'avoir le même comportement. En effet, il n'est aucunement nécessaire de modifier les tâches existantes. La seule chose à faire est de créer deux nouvelles classes ressources sur le nouveau robot qui fourniraient les fonctions définies dans les interfaces de ressource `RobotControl` et `RobotTelemetry`. Ce nouveau robot peut alors utiliser exactement les mêmes tâches que le robot précédent puisque ces tâches ne font référence qu'à un type de ressource, et non à une ressource directement. C'est le fichier de configuration du nouveau robot qui indique au système quelles classes représentent les ressources demandées.

Un robot actif a souvent plusieurs comportements qui s'exécutent simultanément, en particulier lorsqu'une approche comportementale classique est utilisée. En effet, avec cette

approche, l'arbre de tâches contiendrait plusieurs tâches simples s'exécutant en parallèle et exerçant chacune leur influence sur le robot, favorisant ainsi l'émergence d'une action plus complexe. Dans *AIDER*, l'arbitrage des commandes issues de comportements actifs en parallèle n'est pas fait au niveau des comportements eux-mêmes, mais bien au niveau de la ressource à laquelle ils accèdent. En effet, si plusieurs comportements accèdent à une même ressource en même temps, c'est le contrôleur d'accès associé à la ressource qui fait l'arbitrage des requêtes. Un contrôleur d'accès est un objet indépendant de la ressource à laquelle il est associé. Un même contrôleur d'accès peut ainsi être utilisé pour différentes ressources. Ainsi, une fois que les principaux types de contrôleur d'accès ont été développés, le développeur d'une nouvelle ressource n'a plus à se soucier de cet aspect ; il n'a qu'à choisir le type d'arbitrage qu'il désire parmi les contrôleurs existants. Divers types de contrôleurs d'accès peuvent être développés afin de satisfaire différents schèmes de partage de ressources. On peut par exemple désirer faire une fusion des différentes requêtes, ou encore accepter seulement la requête du comportement ayant le plus haut niveau de priorité.

Dans ce dernier cas, le contrôleur peut tenir compte de deux variables afin de faire l'arbitrage. La première est la *priorité de tâche* qui est assignée au moment de la création du comportement et qui est fixe. La deuxième est une *priorité dynamique* qui varie en fonction de l'état interne du comportement. Par exemple, la priorité dynamique d'une tâche de détection d'obstacles augmenterait à l'inverse de la distance de l'obstacle le plus proche. Le produit de ces deux priorités est la valeur qui est utilisée pour effectuer l'arbitrage.

$$priorite_{totale} = priorite_{tache} \times priorite_{dynamique} \tag{3.1}$$

Un tel contrôleur peut facilement être utilisé afin de simuler des sentiments tel l'impatience, en utilisant le délai de non-activité comme priorité dynamique.

3.3.4 Gestion de la communication

AIDER offre aux tâches, ainsi qu'à ses différents modules internes, un canal de communication par le biais du gestionnaire de messages. Ce dernier fournit des services d'envoi et

de réception de messages. Afin de pouvoir recevoir des messages, une entité doit préalablement s'enregistrer auprès du gestionnaire en spécifiant son numéro d'identification, qui correspond à son adresse, ainsi que l'objet qui est responsable du traitement des messages reçus à cette adresse. Notons toutefois qu'un développeur utilisant *AIDER* n'a pas à se soucier de ce détail puisque l'enregistrement des divers éléments est effectué par le système au moment de leur création.

Chaque élément de *AIDER*, que ce soit un gestionnaire, une tâche ou une ressource, détient son propre numéro d'identification, et ce numéro est unique à travers tout le système, ce qui permet de l'utiliser comme adresse dans le système de messagerie. Cet identificateur unique est composé de trois parties : le numéro du robot, le type de l'élément et un identificateur local pour différencier les divers éléments du même type sur un même robot.

Le gestionnaire de messages permet deux modes d'acheminement des messages : l'envoi direct et la diffusion de messages. C'est le champ de destination du message qui détermine le type d'envoi. Ce sont plus particulièrement les deux premières parties de l'adresse, soit le numéro du robot et le type de l'élément qui sont analysées afin de déterminer comment traiter le message. Si dans l'adresse de destination, le type de l'élément est TOUS_LES_ÉLÉMENTS, cela indique de diffuser le message à l'interne, c'est-à-dire sur le robot d'où provient le message. Si en plus, le numéro du robot indique TOUS_LES_ROBOTS, le message est alors diffusé globalement, c'est-à-dire sur tous les robots du système. Notons qu'il n'est pas permis d'effectuer une diffusion globale tout en spécifiant un type d'élément spécifique, puisque la méthode par inscription à un type de message permet d'obtenir le même résultat de façon plus flexible. Lorsque le numéro de robot et le type d'élément sont tous deux spécifiques, on parle alors d'un envoi direct. La figure 3.10 résume les différentes possibilités pour l'adresse de destination.

Lors d'un envoi direct, le message est dirigé vers l'unique élément du système dont le numéro d'identification correspond au champ *adresse de destination* du message. Lorsque cet élément se trouve sur un autre robot, le gestionnaire de messages envoie automatiquement ce message à l'externe. Si l'élément destinataire se trouve sur le robot local, le message est transmis à l'objet que ce dernier a spécifié lors de son enregistrement pour le traitement

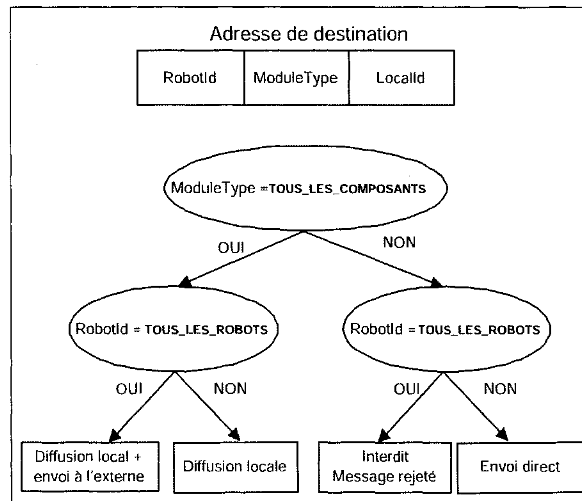

Figure 3.10 – Traitement de l'adresse de destination d'un message

des messages.

La diffusion de messages utilise quant à elle un mécanisme différent. Elle fonctionne par la méthode des abonnements. À cette fin, un autre champ de l'en-tête du message, le *type de message*, est utilisé. Tout élément a la possibilité de s'inscrire auprès du gestionnaire de messages pour recevoir les messages d'un certain type. Il peut à tout moment s'inscrire à de nouveaux types de message ou annuler des inscriptions antérieures. Un élément a aussi la possibilité de s'inscrire afin de recevoir tous les messages diffusés, peu importe leur type. C'est au moment de cette inscription que l'élément précise l'objet qui effectuera le traitement des messages, tout comme il l'avait fait lors de l'enregistrement de son numéro d'identification. Lorsque le gestionnaire de messages reçoit un message à diffuser, il envoie d'abord celui-ci à tous les éléments inscrits pour recevoir tous les messages, puis il envoie le message à tous ceux inscrits pour recevoir ce type de message en particulier. Si la diffusion est globale, le gestionnaire envoie aussi le message à l'externe afin que tous les robots du système le reçoivent.

La conception du mécanisme d'envoi de messages vers le monde extérieur a été faite de façon à permettre à *AIDER* d'être utilisée dans différents environnements. En effet, lorsque le gestionnaire de messages reçoit un message qu'il doit diriger vers le monde extérieur, il transmet ce message à tous les objets qui se sont préalablement enregistrés comme repré-

sentant du monde extérieur. Ce sont ces objets qui sont alors responsables de transmettre le message aux autres robots. Un développeur désirant changer la façon de transmettre les messages entre les robots n'aurait qu'à développer son module de communication et à l'enregistrer comme représentant externe auprès du gestionnaire de messages. Au cours du développement de *AIDER*, le représentant externe qui a été utilisé était un module de communication qui diffusait les messages sur un réseau IP sans fil en utilisant le protocole UDP et la sérialisation standard du langage Java. Ce module pourrait facilement être remplacé par un autre qui transmettrait les messages sous un autre format, XML par exemple, ou encore en utilisant un protocole différent (TCP/IP, Bluetooth, transmission sans-fil propriétaire, etc.). Ce mécanisme a aussi l'avantage de faciliter l'interaction de *AIDER* avec d'autres systèmes, peu importe le langage utilisé par ceux-ci. Il suffit de développer le module de communication qui traduit les messages dans le format utilisé par le système avec lequel interagir. Il n'existe donc pas de restriction quant à la façon de transmettre les messages vers le monde extérieur.

En restant toujours dans cette optique de flexibilité, aucune restriction n'est appliquée sur le contenu des messages envoyés, en faisant abstraction de l'en-tête bien sûr. C'est le rôle du développeur de s'assurer que le contenu du message qu'il envoie sera compris par son destinataire. Ce système de communication peut ainsi être utilisé autant à l'interne par les éléments principaux de *AIDER*, pour envoyer des messages de contrôle, que par tout type de tâche développée par la suite et qui nécessite un échange d'information.

Nous venons donc de voir les principales caractéristiques du système de communications de *AIDER*. Les messages transmis contiennent évidemment d'autres champs, mais ils sont utilisés de façon standard et il n'est donc pas nécessaire d'en faire ici une description exhaustive. Le contenu complet de l'en-tête d'un message est présenté à la figure 3.11.

3.3.5 Configuration d'un système

Le développement de systèmes multi-robots se fait habituellement par une série d'étapes successives. On développe généralement les fonctionnalités de base de chacun des robots, pour ensuite s'attaquer à leurs fonctionnalités plus complexes, et enfin, on termine en

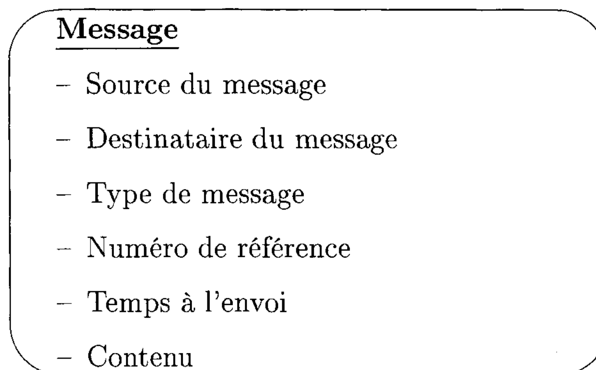

Figure 3.11 – Contenu d'un message

développant les fonctionnalités de groupe. Ces différentes étapes, ainsi que le déverminage, nécessitent de nombreux changements dans la configuration des robots. C'est pourquoi il est important de pouvoir faire facilement ces changements de configuration. *AIDER* utilise à cette fin un fichier de configuration de format XML qui permet de contrôler de façon très simple tous les paramètres d'un robot.

Ce fichier de configuration est la seule chose qui diffère entre les différents robots d'un groupe, tout le reste du code étant exactement le même. Ainsi, pour déployer *AIDER*, il n'y a que deux choses à fournir : l'archive de l'application (le fichier .JAR contenant tout le code) et un fichier de configuration pour chaque robot. Le lancement de *AIDER* se fait en évoquant la classe de base présente dans le fichier .JAR et en lui passant en paramètres le répertoire contenant les fichiers de configuration des robots à démarrer. Un robot est démarré pour chaque fichier de configuration présent dans le répertoire.

La structure de base d'un fichier de configuration est présentée à la figure 3.12. On y retrouve quatre principales sections : l'identification du robot, la définition des ressources, la définition des capacités et les tâches à lancer au démarrage.

La section d'identification permet de différencier entre eux les robots. Elle comprend :
- un nom de robot (*robot-name*),
- un numéro d'identification (*robot-id*), qui doit être unique à travers le système,
- et le numéro du port qui doit être utilisé pour la communication (*inet-port*).

Notons que ce dernier paramètre n'est utilisé qui si l'on emploie un module de communi-

cation IP. Il permet notamment d'avoir deux groupes de robots totalement indépendants sur un même réseau, ce qui s'avère parfois essentiel, comme par exemple dans le cas où plusieurs chercheurs dans un même laboratoire travailleraient sur des projets différents. Afin de prévenir toute erreur liée à un conflit de numéro d'identification, une vérification est faite lors du démarrage de chaque robot afin de s'assurer qu'aucun autre robot sur le réseau n'a le même numéro d'identification.

La deuxième section du fichier de configuration contient une énumération des ressources disponibles sur le robot. Chaque entrée comprend :

- le nom de la ressource (*resource-name*),
- l'index de la ressource (*resource-id*),
- les paramètres d'initialisation(*parameter*),
- le nom de la classe d'implémentation (*implementing-class-name*),
- et le nom de la classe d'interface (*interface-class-name*).

Le nom de la ressource indique le type de travail effectué par cette ressource. L'index de la ressource est facultatif lorsqu'il n'y a qu'une seule ressource d'un même type puisque cet index est utilisé uniquement pour distinguer des ressources d'un même type entre elles. Les paramètres d'initialisation inscrits dans le fichier de configuration sont transmis directement à la ressource lors de son initialisation. Cette ressource en question est chargée de façon dynamique par *AIDER* en utilisant le nom de la classe d'implémentation. Le nom de la classe d'interface permet quant à lui de préciser le type d'interface qui est remis à une classe désirant accéder à la ressource. Ce paramètre est facultatif et lorsqu'il est absent, l'interface remise aux tâches est une interface générique.

La troisième section du fichier de configuration permet de définir les différents services que le robot peut fournir. Elle comprend, pour chaque tâche que le robot peut accomplir :

- le nom du service fourni (*service-type*),
- le niveau de confiance (*confidence-level*),
- la liste des paramètres (*param-range*),
- le nom de la classe d'implémentation (*implementing-class-name*),
- et les ressources nécessaires (*resource-dependency*).

Afin d'utiliser à son maximum les possibilités de réutilisation de code qu'offre *AIDER*, il

est recommandé de standardiser les *nom de service* de façon à ce qu'un même service ait le même nom à travers tout le système. Par exemple, un service permettant à un robot de se rendre à un endroit spécifié pourrait s'appeler `GotoLocation`. Ainsi, une tâche de plus haut niveau faisant appel à ce service fonctionnerait avec tous les robots ayant cette capacité.

Le deuxième paramètre, le niveau de confiance, permet d'indiquer à quel point le robot est capable de rendre le service avec succès et avec une efficacité maximale. Par exemple, dans le cas du service `GotoLocation`, on peut assigner un niveau de confiance plus élevé à un robot ayant une meilleure mobilité. Grâce à ce paramètre, un planificateur de tâches peut prendre une décision plus éclairée lorsque plus d'un robot sont en mesure d'effectuer la tâche demandée.

La définition d'une capacité comprend aussi la liste des paramètres qui peuvent ou doivent être fournis lorsque l'on désire utiliser le service. Pour chacun de ces paramètres, on indique dans le fichier de configuration si le paramètre est requis ou optionnel. On y indique aussi les plages de valeurs possibles, comme par exemple les vitesses minimales et maximales permises pour un service de déplacement du robot.

On trouve ensuite, dans la définition d'un service, le nom de la classe qui contient l'implémentation du service. C'est cette classe qui est chargée de façon dynamique par *AIDER* lorsqu'une requête de service valide est reçue. Une description du contenu d'une telle classe est donnée à la section 3.3.2.

La définition d'un service se termine par la liste des ressources qui sont nécessaires afin de pouvoir effectuer la tâche demandée. On y indique le type de la ressource nécessaire, ainsi que son index si une instance précise de ce type de ressource est requise.

```
<robot-definition xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <robot-identification robot-id="1" inet-port="10050">
    <robot-name>Nom du robot</robot-name>
  </robot-identification>
  <resource-definitions xsi:type="resource-definition" resource-id="1">
    <resource-name>Ressource1</resource-name>
    <parameter name="Param1" value="Valeur du parametre 1"
      xsi:type="string-parameter"/>
    <parameter name="Param2" value="10" xsi:type="integer-parameter"/>
    <parameter name="Param3" value="7.0" xsi:type="float-parameter"/>
    <implementing-class-name>moncode.ressources.Ressource1
      </implementing-class-name>
    <interface-class-name>moncode.ressources.interfaces.Type1
      </interface-class-name>
  </resource-definitions>
  <capability-definitions confidence-level="90" xsi:type="capability">
    <service-type>Service1</service-type>
    <param-range min-value="0" max-value="100"  required="true"
      xsi:type="integer-range">
      <name>Param1</name>
    </param-range>
    <param-range min-value="0" max-value="100" required="true"
      xsi:type="float-range">
      <name>Param2</name>
    </param-range>
    <param-range required="false" xsi:type="string-range">
      <name>Param3</name>
    </param-range>
    <implementing-class-name>moncode.services.Service1
      </implementing-class-name>
    <resource-dependency resource-id="0" robot-id="1"
      xsi:type="resource-dependency">
      <resource-name>Ressource1</resource-name>
    </resource-dependency>
  </capability-definitions>
  <startup-tasks xsi:type="task-description">
    <service-type>Service1</service-type>
    <parameter name="Param1" value="1" xsi:type="integer-parameter">
      </parameter>
    <parameter name="Param2" value="2.4" xsi:type="float-parameter">
      </parameter>
  </startup-tasks>
</robot-definition>
```

Figure 3.12 – Structure de base d'un fichier de configuration

CHAPITRE 4

SCÉNARIO D'ÉVALUATION

Une architecture d'une certaine ampleur, telle que *AIDER*, ne peut être validée et évaluée à partir de simples tests unitaires. En effet, les nombreuses interactions entre les différents modules, de façon asynchrone dans bien des cas, ne peuvent à toute fin pratique pas être reproduites fidèlement par des tests de bas niveau. Une architecture d'interactions entre robots ne peut, par sa nature, qu'être jugée en fonction de ses capacités et performances telles que démontrées dans un réel contexte de collaboration entre plusieurs robots. C'est pourquoi un scénario a été développé afin d'effectuer la validation et l'évaluation de *AIDER*. Ce projet ayant été développé en collaboration avec l'Agence Spatiale Canadienne, le choix d'un scénario lié à l'exploration spatiale s'est imposé de lui-même. *AIDER* a donc été utilisée pour implémenter le système de contrôle d'un groupe de robots simulant un scénario de recherche planétaire. Les sections suivantes présentent plus en détails les différents aspects de ce scénario.

4.1 Objectifs du scénario

L'objectif premier du scénario est de recréer une situation proche de la réalité dans laquelle plusieurs robots doivent coopérer afin d'accomplir une tâche donnée. Cela permet d'évaluer l'architecture dans son ensemble et de vérifier son utilité. Il est aussi important que ce scénario permette d'évaluer les différentes fonctionnalités de *AIDER*.

Voici donc une liste des différentes fonctionnalités qui sont validées au cours du déroulement du scénario.

Gestion des tâches (chaque item doit être vérifié pour une tâche locale et distante)

- Démarrage de tâches.
- Pause et redémarrage de tâches.
- Arrêt de tâches.
- Attente pour la terminaison d'une tâche.
- Attente pour la terminaison d'un groupe de tâches.
- Récupération de l'état d'une tâche.
- Récupération de la valeur de retour d'une tâche.
- Inscription aux événements de changement d'état d'une tâche.

Gestion des ressources (chaque item doit être vérifié pour une ressource locale et distante)

- Acquisition d'une interface à une ressource.
- Utilisation d'un index pour différencier plusieurs ressources d'un même type.
- Démarrage d'une ressource lors de la première requête d'accès.
- Utilisation d'une ressource associée à du matériel.
- Utilisation d'une ressource logicielle seulement.
- Utilisation d'une interface générique pour accéder à une ressource.
- Utilisation d'une interface spécifique pour accéder à une ressource.
- Contrôle d'accès à une ressource.

Communications

- Transmission directe de messages.
- Diffusion de messages.
- Fiabilité du système de communication lors de trafic important.
- Utilisation de la bande passante.
- Délai de transmission.

Configuration et démarrage

- Configuration de l'identité des robots.
- Création des listes de capacités.
- Création et initialisation des ressources.
- Vérification de la disponibilité des ressources nécessaires aux capacités.

– Lancement automatique de tâches au démarrage.

Autres mécanismes

– Adaptation facile à différents robots.
– Mise à jour des listes de capacités lors de changements d'état d'une ressource.
– Mise à jour des listes de capacités périodiquement.
– Fonctionnement du Visionneur de Tâches (*TaskViewer*).

Un scénario de base a donc été développé, puis enrichi afin d'utiliser toutes les fonctionnalités énumérées. La prochaine section présente en détail le résultat de cette démarche.

4.2 Description

Le scénario développé correspond à une des tâches les plus souvent mentionnées lorsque l'on parle d'exploration planétaire, c'est-à-dire l'exploration robotisée d'un terrain afin de mieux comprendre l'histoire d'une planète. De nombreux regroupements actifs dans le domaine spatial étudient depuis déjà plusieurs années la possibilité d'envoyer une équipe de robots, plutôt qu'un seul, afin d'effectuer cette tâche. En partant de cette base, le scénario élaboré impliquant un groupe de robots consiste à explorer une zone encombrée de rochers avec comme objectif la recherche de points d'intérêt. Une fois repérés, les points d'intérêt doivent être examinés à l'aide d'un instrument présent sur un ou plusieurs robots du groupe.

Ainsi, on prétend qu'un atterrisseur portant à son bord deux robots rouleurs a récemment atterri sur la planète à étudier. Le scénario commence alors que les deux robots rouleurs viennent tout juste de quitter l'atterrisseur en roulant sur une courte distance pour s'en dégager. Comme l'objectif de ce travail n'est pas de développer un algorithme évolué de navigation, le groupe de robot commence sa mission avec en sa possession une carte du terrain indiquant l'emplacement des rochers assez gros pour empêcher un robot de passer. On se retrouve donc avec un groupe de trois robots (un atterrisseur et deux robots rouleurs) prêts à répondre aux commandes d'un contrôleur. Dans notre cas, le contrôleur est un humain, sur Terre, utilisant une console de contrôle pour envoyer les tâches à

effectuer. Afin de vérifier que l'architecture peut s'adapter à différents types de robots, et puisqu'un seul robot rouleur est disponible pour nos expérimentations pratiques, le scénario est réalisé en utilisant un robot rouleur, un robot fixe représentant la plate-forme d'atterrissage (fournit des services de vision et gestion des tâches) ainsi qu'un troisième robot virtuel représenté par un simulateur qui utilise une version numérique du terrain dans lequel évoluent les deux robots réels. Les deux robots rouleurs ont les capacités pour se déplacer, détecter des obstacles, trouver des cibles et les analyser. La station fixe, de son côté, peut seulement trouver des cibles à l'aide d'une caméra montée sur une unité motorisée.

On désire que l'exploration se fasse de façon entièrement autonome, sans aucune intervention humaine. Par conséquent, la seule commande qui est envoyée à partir de la console de contrôle est d'explorer le terrain en entier. Il faut donc qu'il y ait par la suite répartition des tâches entre les différents robots. Il a ainsi fallu faire un choix entre les deux principales techniques de répartition des tâches, soit la prise de décision centralisée et la prise de décision distribuée. Dans un contexte d'exploration spatiale, le principal avantage de l'approche distribuée est sa plus grande robustesse face aux fautes d'un ou plusieurs robots. Par contre, une approche centralisée permet d'optimiser l'exécution en rassemblant toute l'information en un seul endroit. En outre, le résultat est plus facile à prévoir avec ce type d'approche, ce qui est un avantage majeur dans un contexte où les gestionnaires de projets spatiaux sont encore réticents à l'utilisation de robots intelligents. Ces raisons ont mené à la décision d'utiliser l'approche de prise de décision centralisée pour le scénario d'évaluation. La commande d'exploration est donc envoyée à un des robots, qui lui se charge de redistribuer les tâches entre les différents robots du groupe en fonction de leurs capacités.

La tâche d'exploration du terrain est divisée en trois fonctions principales : l'exploration d'une zone par un robot rouleur parcourant toute la zone à la recherche de cibles intéressantes, la recherche visuelle de cibles et l'analyse des cibles trouvées. Le tableau 4.1 présente les rôles que peuvent prendre chacun des robots du scénario ainsi que leurs particularités. Ainsi, lorsque débute l'exploration, les deux robots rouleurs commenceront à

parcourir chacun leur moitié de la zone tout en cherchant des cibles à analyser. Pendant ce temps, la station fixe utilise son système de vision mobile afin de rechercher elle aussi des cibles. Lorsqu'un robot mobile trouve une cible, il tente immédiatement de l'analyser. Lorsqu'une cible est trouvée par la station fixe, ou qu'un robot mobile échoue dans sa tentative d'analyser une cible qu'il vient de trouver, cette cible est insérée dans une liste de cibles à analyser plus tard. Chaque fois qu'un robot termine une tâche, le robot chargé de la distribution des tâches vérifie s'il reste d'autres tâches que ce robot est capable d'effectuer, comme par exemple l'analyse de cibles trouvées plus tôt. L'exploration se termine lorsque toutes les recherches de cibles sont terminées et que le terrain en entier a été exploré, à l'exception des zones ou les robots n'ont pu se rendre. Une carte d'exploration, mise à jour tout au long du processus, indique alors les cibles analysées, les cibles trouvées mais non analysées ainsi que les zones ou les robots mobiles n'ont pu se rendre.

Afin de vérifier la mise à jour des capacités lors de changements d'état d'une ressource, l'analyseur d'un des robots mobiles est programmé afin de cesser de fonctionner à partir de sa deuxième tentative d'analyse. Cela permet de vérifier que sa capacité (*AnalyzeTarget*) est bel et bien retirée et que le distributeur de tâches ne lui assigne plus de tâches d'analyse. De plus, cela permet de vérifier que la cible en cours d'analyse lors de la panne est bien réassignée à un autre robot pour une analyse ultérieure.

La vérification du fonctionnement des schèmes de contrôle d'accès aux ressources étant un des objectifs du scénario, ce dernier utilise deux contrôleurs d'accès différents. Le premier, utilisé par la plupart des ressources, est un contrôleur de type «premier arrivé-premier servi» avec mise en queue des requêtes. Le deuxième contrôleur utilise le schème de priorité dynamique décrit à la section 3.3.3 et est utilisé pour les ressources de contrôle des moteurs des robots. Afin de vérifier le fonctionnement de ce contrôleur d'accès par priorité dynamique, une tâche de détection d'obstacles est ajoutée au scénario. Cette tâche est active durant tout déplacement d'un robot et sa priorité dynamique varie de façon inversement proportionnelle à la distance de l'obstacle le plus près. Lorsque sa priorité totale dépasse celle de la tâche qui contrôle à ce moment le robot, la tâche de détection d'obstacles prend le contrôle et arrête les moteurs afin de prévenir une collision.

TABLEAU 4.1 – Rôles des robots

	Robots rouleurs 1 & 2	Station fixe
Exploration	Supporté	Non supporté
Recherche visuelle de cible	Courte portée	Longue portée
Analyse	Supporté	Non supporté

4.3 Environnement d'implémentation

Le développement du scénario ainsi que tous les tests ont été effectués dans les locaux de l'Agence spatiale canadienne, à Saint-Hubert (Qc), Canada. Les sections 4.3.1 et 4.3.2 présentent les différentes ressources matérielles et logicielles utilisées pour exécuter le scénario d'évaluation.

4.3.1 Ressources matérielles

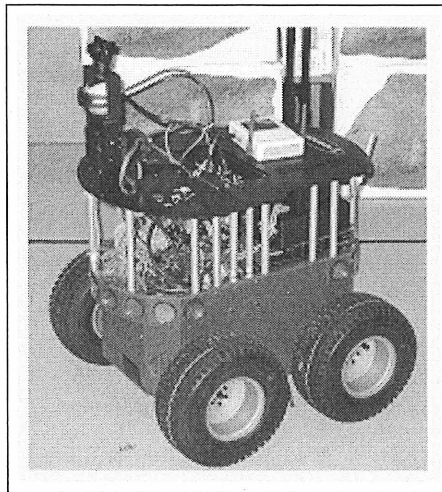

Figure 4.1 – P2-AT modifié utilisé dans le scénario

Les principales ressources matérielles utilisées sont les deux plates-formes robotiques réelles, l'ordinateur simulant le troisième robot et le terrain simulant le sol martien. Le premier robot est un Pioneer P2-AT de la compagnie ActivMedia. Ce robot rouleur à quatre roues est muni de deux séries de huit sonars, l'une à l'avant et l'autre à l'arrière. Il possède aussi

une unité de mesure des déplacements par inertie ainsi que quatre capteurs de proximité à l'avant. Le P2-AT original a été modifié afin de le rendre étanche à l'eau et d'augmenter le volume du boîtier qui contient l'électronique. Il est relié à un réseau IP par un adapteur Ethernet sans-fil. La photo présentée à la figure 4.1 montre le robot sans le boîtier étanche. Notons que la caméra présente sur la photo n'était plus disponible à cause de modifications requises par un autre projet utilisant le même robot. L'ordinateur à bord du robot est muni d'un processeur Pentium III de Intel cadencé à 850MHz et de 96MB de mémoire vive. Le système d'exploitation utilisé est Linux.

La station fixe est formée d'un ordinateur portable muni d'un processeur Pentium III de Intel cadencé à 750MHz et de 512MB de mémoire vive. Le système d'exploitation utilisé est Windows 2000. Cet ordinateur est relié à une caméra USB Logitech QuickCam Pro 3000 montée sur une unité de positionnement motorisée. Cette dernière est contrôlée par l'ordinateur portable par l'entremise du port série.

Le troisième robot étant entièrement simulé, il ne nécessite qu'un ordinateur. Un des serveurs du laboratoire a donc été utilisé à cette fin. Il s'agit d'un processeur Intel Xeon cadencé à 2.4GHz et possédant 1GB de mémoire vive. Ce serveur utilise le système d'exploitation Linux.

Le terrain dans lequel se déroule le scénario constitue un autre élément important du matériel utilisé. Une zone rectangulaire de 4 mètres par 8 mètres a été réservée dans les Hautes Baies à l'Agence spatiale canadienne. Des boîtes de carton peintes ont été utilisées pour représenter les rochers martiens assez gros pour empêcher les robots de passer. La figure 4.2 présente deux photographies du terrain.

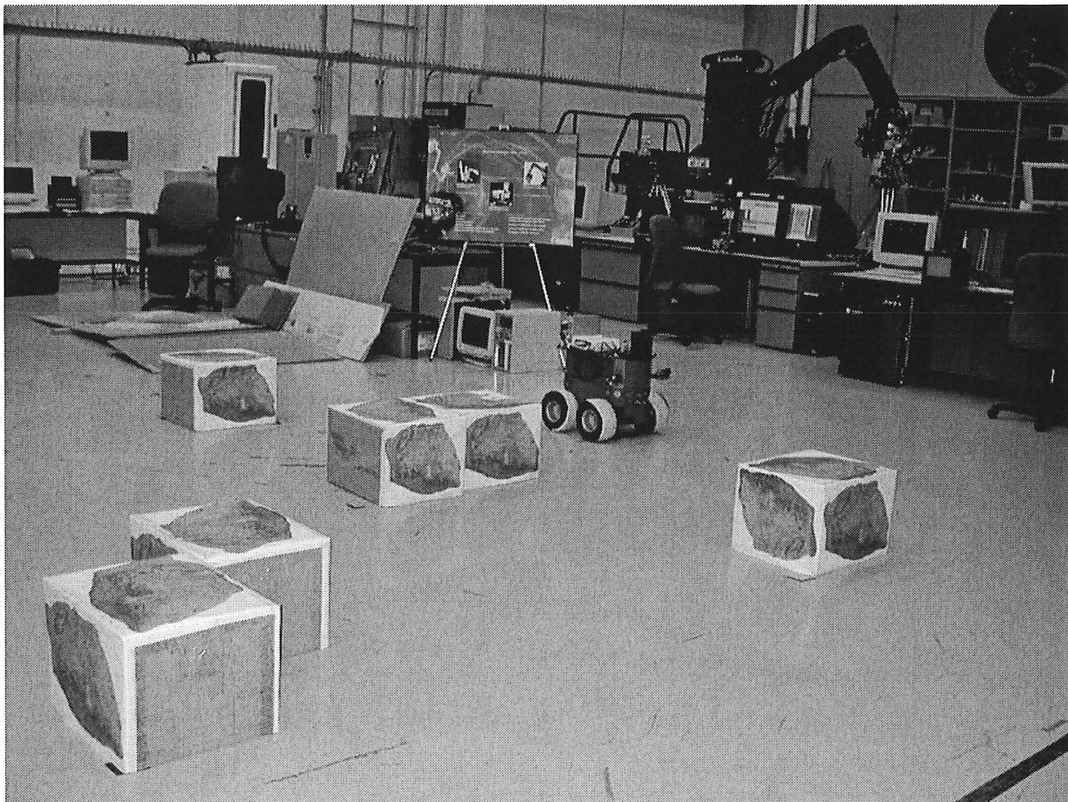

Figure 4.2 – Terrain utilisé pour le scénario

4.3.2 Ressources logicielles

Cette section présente les outils logiciels déjà existants qui ont été utilisés pour développer le scénario d'utilisation.

Premièrement, une librairie nommée P2OS est fournie avec le robot P2-AT afin d'accéder à sa télémétrie et de contrôler ses moteurs et ses sonars. Une classe nommée `P2osInterface` avait déjà été programmée à l'Agence spatiale canadienne afin de pouvoir accéder à ces fonctionnalités à partir de Java. Cette classe a donc été réutilisée lors de la création des ressources associées au contrôle et à la télémétrie du robot.

Pour ce qui est du robot simulé, l'environnement de simulation utilisé est Player/Stage, développé à USC (*University of Southern California*) [Vaughan et coll., 2003]. *Stage* étant un simulateur supportant plusieurs robots, il ne restait plus qu'à développer le client Java permettant de s'y connecter.

Deux autres classes développées à l'Agence spatiale canadienne ont aussi pu être utilisées afin d'accélérer le développement du scénario. La première, `PathPlanner`, fournit un service de navigation en terrain connu. Elle permet, à partir d'une carte, de donner le trajet qu'un robot peut emprunter entre deux points situés sur cette carte. La deuxième classe, `VisionSystem`, contient un algorithme d'analyse d'images permettant la détection d'objets colorés, ainsi que le service de capture d'images pour une caméra Logitech. Elle a aussi la capacité de contrôler une unité de positionnement de caméra motorisée (*pan-tilt unit*).

La librairie XML *Castor* de *Exolab Group*, distribuée librement, a été utilisée afin de faciliter la création d'une description de robot à partir du fichier de configuration en format XML.

4.4 Implémentation avec *AIDER*

Le développement de tout projet à l'aide de *AIDER* peut se décomposer en deux grandes phases : le développement des ressources et de leurs interfaces, et le développement des tâches.

4.4.1 Ressources

Il faut tout d'abord identifier les différentes ressources requises. Dans le cas de la station fixe, le seul matériel à contrôler est la caméra montée sur l'unité de positionnement motorisée. Par contre, au niveau logiciel, il faut aussi avoir accès à un algorithme d'analyse d'image. Comme la classe `VisionSystem` présenté à la section 4.3.2 fournissait déjà tous ces services, il suffisait de développer la classe ressource faisant l'interface entre *AIDER* et la classe existante. Cette nouvelle classe, `VisionControl`, exporte les fonctions *findTarget()* et *setPanTilt()* et utilise un contrôleur d'accès de type premier arrivé-premier servi.

Les deux robots mobiles utilisent tous deux les mêmes ressources, même si un des deux robots est entièrement simulé. En premier lieu, ces robots ont besoin de pouvoir contrôler leurs moteurs et d'obtenir leur télémétrie afin de pouvoir se déplacer. Le développeur doit donc créer une ressource qui permet de contrôler le robot et une autre qui permet d'obtenir l'information sur ses capteurs et sa position. Le choix d'utiliser deux ressources séparées plutôt qu'une seule est dicté par le désir d'utiliser une règle d'arbitrage différente pour la ressource de contrôle et celle de télémétrie. En effet, dans le premier cas, la ressource `RobotControl` ne doit être utilisée que par une tâche à la fois. On y associe donc un contrôleur d'accès par priorité dynamique. Dans le cas de la ressource `RobotTelemetry`, il n'est pas nécessaire de contrôler l'accès simultané alors un contrôleur de type premier arrivé-premier servi est utilisé.

Dans le cas du robot réel, nous avons déjà une classe Java (présentée à la section 4.3.2) qui permet d'accéder aux ressources matérielles du robot. Les deux ressources développées ne sont donc que des interfaces qui permettent aux tâches dans *AIDER* d'accéder au robot à travers cette classe. Dans le cas du robot simulé, les commandes équivalentes doivent être envoyées au simulateur *Stage*. Il a donc fallu développer un client en Java pour communiquer avec le simulateur. Une interface identique à la classe `P2osInterface` fut développée afin de permettre d'accéder au vrai robot. De cette manière, les mêmes classes ressources peuvent être utilisées pour un robot réel et un robot simulé. La seule différence se situe au moment de la création de la ressource. Dans un cas, la ressource

instantie un objet de la classe `P2osInterface` alors que dans l'autre, elle instantie le client vers *Stage*. Un paramètre de la ressource permet d'indiquer si celle-ci doit se connecter sur le robot réel ou sur un simulateur. Lorsqu'un simulateur est utilisé, un autre paramètre de la ressource permet d'indiquer l'adresse IP de ce dernier.

Une fois les robots capables de se déplacer, ils doivent alors pouvoir planifier les déplacements qu'ils effectueront. Comme une classe offrant ce service a été développée préalablement, il suffit de créer la classe ressource qui permet d'y accéder. Cette ressource, nommée **`PathPlanner`**, reçoit en paramètres le nom du fichier contenant la carte du terrain. Elle crée ensuite le générateur de trajectoire en l'initialisant avec la carte.

Les deux robots mobiles doivent aussi être capables de détecter des cibles à analyser. Cependant, comme il n'y avait pas de caméra sur le robot (voir section 4.3.1), le développeur a décidé de simuler la détection des cibles en créant une ressource qui connait à l'avance la position des cibles. Les positions de ces cibles sont passées en paramètres à la ressource par le biais du fichier de configuration. Ainsi, lorsque le robot se rapproche d'une cible, cette ressource indique la position de la cible, comme l'aurait fait une ressource utilisant un vrai système de vision.

Finalement, afin de pouvoir analyser les cibles trouvées, les robots mobiles doivent avoir un analyseur. Il n'est pas utile dans le cadre de notre scénario d'avoir de réels analyseurs. La ressource **`Analyzer`** se contente donc de simuler l'analyse en imposant un délai, puis en retournant les résultats.

Cela termine la description des ressources nécessaires pour les différents robots du scénario. Notons que toutes ces ressources, à l'exception de la ressource de contrôle du robot, utilisent un contrôleur d'accès du type premier arrivé-premier servi. De plus, pour chacune des ces ressources, une interface spécifique a été générée afin de faciliter l'écriture et la lecture du code. La génération de ces interfaces a été complètement automatisée grâce à l'outil *InterfaceGenerator* développé à cet effet.

4.4.2 Tâches

La deuxième étape dans le développement du scénario d'évaluation est de définir et de programmer les différentes tâches. Cette étape implique aussi de choisir le type de relation que l'on désire entre les différentes tâches. Les adeptes de l'approche comportementale pourraient alors concevoir un ensemble de tâches simples indépendantes alors qu'à l'opposé, on pourrait concevoir le système pour toujours avoir une tâche qui contrôle toutes les autres. L'approche choisie pour le développement de notre scénario se situe à mi-chemin entre ces deux techniques. Elle consiste en un arbre de tâches dans lequel on retrouve des tâches simples indépendantes s'exécutant en parallèle avec des tâches plus complexes, elles-mêmes décomposées en sous-tâches plus simples s'exécutant simultanément ou séquentiellement.

En partant des services que les robots doivent fournir, le développeur définit d'abord les tâches de plus haut niveau pour chacun des robots.

Dans le cas des robots mobiles, ils doivent être capables de :
- se rendre de façon sécuritaire à un endroit désigné,
- trouver des cibles à analyser et
- analyser des cibles.

Pour ce qui est de la station fixe, elle doit uniquement être capable de trouver des cibles à analyser. On désire aussi, pour chacun des robots, avoir une tâche de sécurité qui vérifie de façon continue la santé du robot (niveau de batterie, température, etc.).

Il faut ensuite décider, pour chacune de ces fonctions principales, comment les décomposer en tâches plus simples. Nous créons tout d'abord une tâche `GotoLocation` qui permet à un robot de se rendre à un point donné, n'importe où sur la carte. Afin de simplifier la programmation et de favoriser la réutilisation de tâches, nous décomposons la partie déplacement de cette tâche en une séquence de déplacements en ligne droite qui sont exécutés par une sous-tâche nommée `FollowSegment`. La ressource `PathPlanner` fournira les coordonnées de chacun des points de navigation qui sont passés en paramètres à cette tâche.

Nous voulons que tout au long du déplacement, le robot s'assure de ne pas entrer en

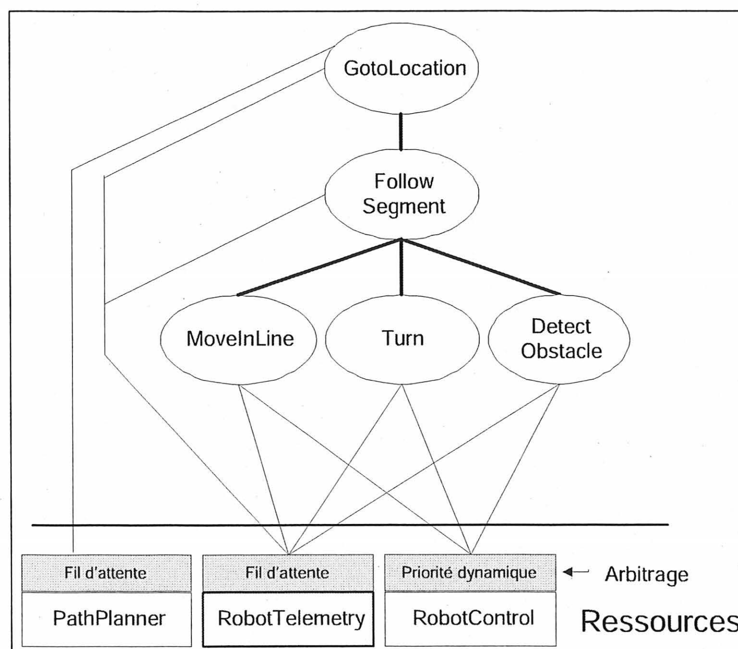

Figure 4.3 – Arbre de tâches pour la partie déplacement du scénario

collision avec un obstacle. Nous utilisons donc une sous-tâche nommée `DetectObstacle` qui s'exécute en parallèle tant que la tâche `FollowSegment` est active. Cette tâche est responsable d'arrêter les moteurs lorsque le robot est trop près d'un obstacle. En plus de cette tâche de détection d'obstacles, la tâche `FollowSegment` est aussi décomposée séquentiellement en une tâche permettant au robot de s'aligner dans la bonne direction (`Turn`) et une autre permettant d'avancer en ligne droite (`MoveInLine`). L'arbre de tâches pour la partie déplacement est présenté à la figure 4.3.

Il faut maintenant définir les tâches nécessaires pour la détection de cible. Cette tâche peut être exécutée autant par les robots mobiles que par la station fixe. Dans le premier cas, il s'agit d'une caméra fixe qui examine le terrain à l'avant du robot (cette partie est simulée, tel qu'indiqué à la section 4.3.1). Dans le cas de la station fixe, la recherche se fait à l'aide de la caméra motorisée qui peut balayer une très grande surface à cause de sa position en hauteur. Nous avons donc un même service fourni de deux façons différentes. Ainsi, nous développons une classe pour chacun de ces cas, et le fichier de configuration de chacun des robots présente leur capacité d'offrir le même service de repérage de cibles, nommé `FindTarget`, mais en précisant une classe d'implémentation différente. Ces tâches étant assez simples, il n'est pas nécessaire de les décomposer.

Il reste à définir la tâche pour l'analyse des cibles, que l'on nomme `AnalyzeTarget`. L'analyse en soi est très simple puisqu'elle ne demande que l'utilisation de la ressource représentant l'analyseur. Cependant, nous permettons à l'utilisateur de cette tâche de passer en paramètres la position de la cible à analyser. La tâche d'analyse utilise donc comme sous-tâche la tâche de déplacement `GotoLocation` afin de se rendre à l'emplacement de la cible. Elle utilise ensuite la ressource `Analyzer` afin d'effectuer l'analyse.

Une fois que sont définies toutes ces tâches pouvant être accomplies par les différents robots, il reste à concevoir la tâche de plus haut niveau qui effectue l'exploration du terrain en utilisant les capacités des trois membres du groupe. Nous définissons pour cette fin la tâche `ExploreArea`. Cette tâche reçoit deux paramètres : les coordonnées du terrain à explorer et les membres de l'équipe d'exploration, i.e. une liste des numéros d'identification des robots pouvant participer à l'exploration. Pour ce deuxième paramètre, il est aussi possible de spécifier que tous les robots connus peuvent participer.

Le paramètre d'équipe indique si l'opération d'exploration se fait en solo ou en équipe, la version solo n'étant disponible que pour les robots pouvant se déplacer et trouver des cibles. Lorsqu'une exploration solo est spécifiée, cette tâche utilise une série de sous-tâches `GotoLocation` afin que le robot parcourt toutes les zones accessibles du terrain à explorer. Une carte d'exploration est mise à jour continuellement afin d'indiquer les zones visitées et les cibles trouvées. Tout au long des déplacements, une sous-tâche de recherche de cibles

(`FindTarget`) est active. Lorsqu'une cible est trouvée, les tâches de déplacement et de recherche de cibles sont arrêtées et une tâche d'analyse (`AnalyzeTarget`) est démarrée. La carte d'exploration est ensuite mise à jour pour indiquer une cible analysée avec succès ou une cible non analysée. Tout au long de son déroulement, la tâche d'exploration envoie des messages de mise à jour de l'exploration qui peuvent être captés par la tâche mère afin de mettre à jour sa propre carte d'exploration. Cela est bien sûr utile lorsque l'exploration se fait en équipe.

Lorsque la requête de tâche précise plus d'un robot dans l'équipe, le robot exécutant cette requête est celui qui effectue la répartition des tâches entre les coéquipiers. Dans un tel cas, la première chose que cette tâche-parent accomplit est de vérifier, pour chaque membre de l'équipe, quelles sont ses capacités. Cela permet de savoir quels robots peuvent explorer en se déplaçant, lesquels peuvent trouver des cibles et lesquels peuvent les analyser. Il suffit ensuite de démarrer une sous-tâche sur chacun des robots en fonction de leurs capacités.

Au départ, puisque aucun emplacement de cible n'est connu, les tâches démarrées consistent uniquement en des tâches d'exploration solo (`ExploreArea`) et des tâches de recherche de cibles (`FindTarget`). La tâche-parent divise le terrain à explorer en sections rectangulaires de grandeurs égales, créant une section pour chacun des robots pouvant se déplacer. Chacun de ces robots se voit assigner la section étant la plus près de sa propre position sur le terrain. Avant de démarrer les tâches d'exploration, la tâche-parent s'inscrit auprès du gestionnaire de messages afin de recevoir les messages de mise à jour de l'exploration. Cela lui permet de mettre à jour la carte maîtresse d'exploration à mesure que les robots accomplissent leurs sous-tâches.

Lorsqu'un robot termine une sous-tâche, la tâche-parent vérifie s'il reste des tâches à effectuer que ce robot est capable d'accomplir. Par exemple, si une cible est trouvée par un robot qui n'a pu l'analyser, son analyse est réassignée au robot libre en démarrant une nouvelle tâche `AnalyzeTarget`, en autant que celui-ci ait la capacité d'analyser des cibles. Le scénario d'évaluation comporte deux situations qui entraînent ce type de réassignation : le cas où l'analyseur du premier robot tombe en panne au cours d'une analyse, et le cas où une cible est trouvée par la station fixe.

Tout au long de l'exploration, la tâche-parent met à jour une image en format jpeg représentant la carte d'exploration, ce qui permet de suivre visuellement le déroulement des recherches. La figure 4.4 présente un exemple de carte. Les zones rectangulaires indiquent des obstacles connus d'avance. Les zones explorées sont représentées par des ronds blancs et les zones où le robot n'a pu se rendre sont de couleur gris pâle. Les petits ronds gris foncé représentent des cibles trouvées.

Les tableaux 4.2 et 4.3 présentent un résumé des différentes tâches et ressources utilisées dans le scénario. La figure 4.5 présente quant à elle une vue globale de l'arbre de tâches du scénario.

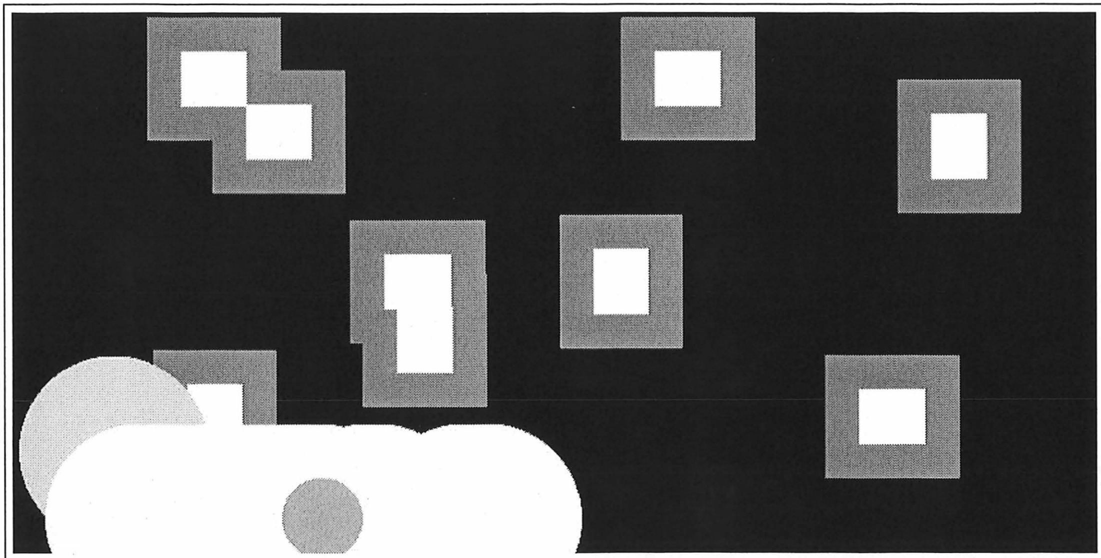

Figure 4.4 – Exemple de carte d'exploration

Une fois l'ensemble des ressources et des tâches conçues, il ne reste plus qu'à créer le fichier de configuration pour chacun des robots, en incluant dans ceux-ci uniquement les ressources et les services appropriés. Le fichier de configuration complet du robot réel est présenté en exemple à l'annexe I.

TABLEAU 4.2 – Ressources utilisées dans le scénario

Ressource	Description	Paramètres possibles
RobotTelemetry	Permet d'obtenir la position du robot et les données de ses capteurs.	UseSimulator RobotName
RobotControl	Permet de contrôler les moteurs du robot et d'activer ses autres capteurs.	UseSimulator RobotName ResetSimulation ResetX ResetY ResetTheta
VisionControl	Permet de contrôler l'unité de positionnement motorisée de la caméra, de prendre des images et de les analyser pour y trouver des cibles.	PTUModel PTUPort CameraType
PathPlanner	Permet de planifier une trajectoire évitant tous les obstacles pour se rendre d'un point A à un point B.	MapFilename
Analyzer	Permet d'analyser le sol.	

TABLEAU 4.3 – Tâches utilisées dans le scénario

Tâche	Description	Paramètres possibles	Sous-tâches utilisées	Ressources utilisées
ExploreArea	Exploration d'une zone déterminée dans le but de trouver des cibles et de les analyser.	XOrigin YOrigin Width Height Team	ExploreArea GotoLocation FindTarget AnalyzeTarget	RobotTelemetry
AnalyzeTarget	Analyse d'une cible.	XLocation YLocation	GotoLocation	Analyzer
FindTarget	Recherche visuelle de cibles.			VisionControl
GotoLocation	Déplacement du robot pour se rendre à un endroit donné en évitant les obstacles.	X Y Theta	FollowSegment	RobotTelemetry PathPlanner
FollowSegment	Déplacement du robot en ligne droite vers un endroit spécifié.	X Y Theta	MoveInLine Turn DetectObstacle	RobotTelemetry
MoveInLine	Déplacement du robot en ligne droite d'une distance spécifée.	Distance		RobotControl RobotTelemetry
Turn	Rotation du robot d'un angle spécifié	Angle		RobotControl RobotTelemetry
DetectObstacle	Détection d'obstacle à l'aide des sonars.			RobotControl RobotTelemetry

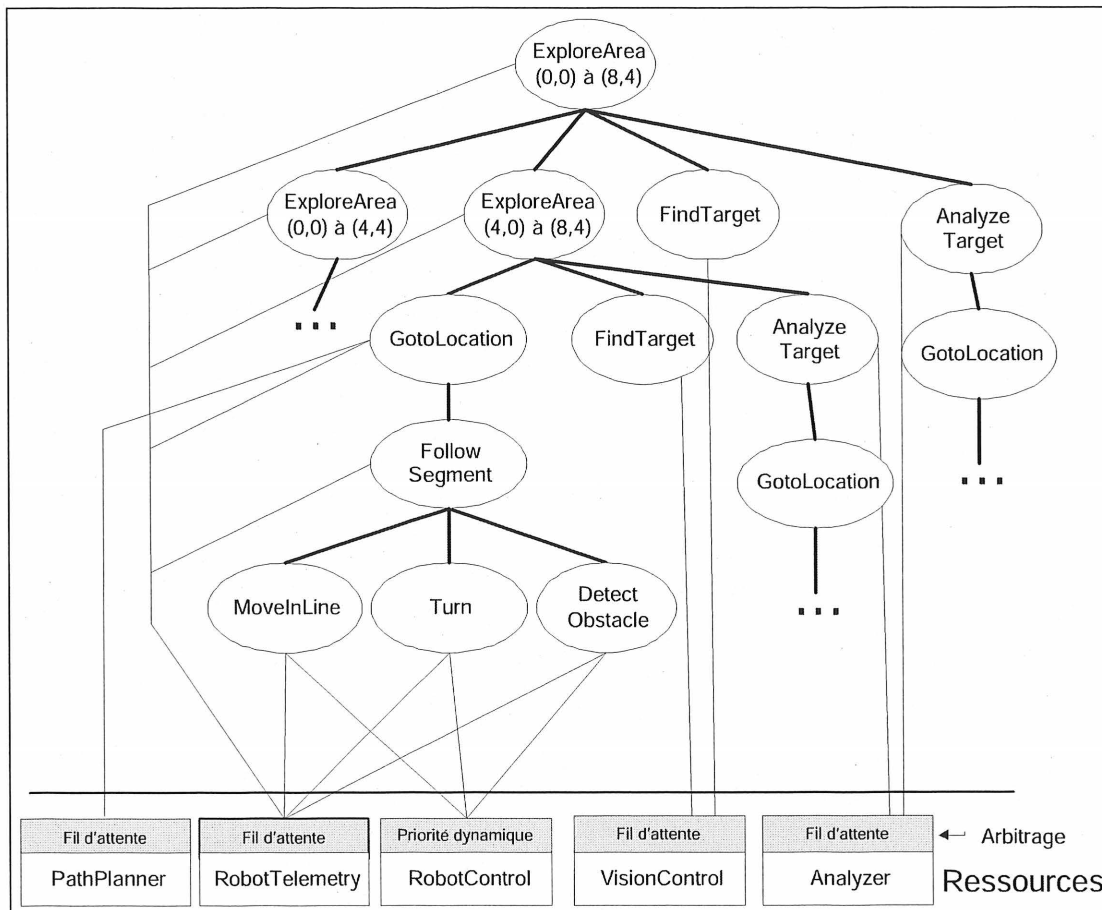

Figure 4.5 – Arbre de tâches du scénario

CHAPITRE 5

ANALYSE ET DISCUSSION

La conception de l'architecture a permis de regrouper dans un système flexible les fonctionnalités requises par tous les objectifs initiaux. Une fois cette étape de conception accomplie avec succès, il était important de vérifier le fonctionnement réel de l'architecture et de s'assurer qu'elle offrait véritablement un environnement souple, facile à utiliser et fiable pour le développement de tâches pour des groupes de robots. Le scénario développé est l'outil principal qui a permis d'effectuer ces vérifications et de tester *AIDER* dans son ensemble. Cette section présente une analyse des différents résultats obtenus afin de définir le degré de réussite de l'architecture en fonction des objectifs de départ. Cela permet aussi de déterminer les points fort de *AIDER* ainsi que les points qui pourraient être améliorés. Le chapitre se termine par une discussion portant sur les travaux futurs, principalement sur les améliorations et les ajouts pouvant être apportés à *AIDER* ainsi que ses utilisations possibles.

Mais avant de commencer à traiter de différents aspects spécifiques de l'architecture, il importe de discuter en premier lieu de la capacité de *AIDER* à assumer son rôle principal, c'est-à-dire de permettre le développement d'une tâche multi-robots. Le scénario présenté au chapitre 4 a été implémenté en totalité avec succès. Cette implémentation a permis tout d'abord de vérifier que l'architecture ne comporte pas de faiblesse majeure qui aurait pu rendre difficile, voir impossible, le développement de la tâche désirée. L'implémentation s'est en effet déroulée sans anicroche et a permis de faire explorer par un groupe de robots, de façon autonome, une zone déterminée en y analysant les points d'intérêts trouvés.

Le scénario dans son entier, tel que décrit au chapitre précédent, a été exécuté à vingt reprises afin de vérifier la stabilité du système et la répétitivité de son action. Chaque

essai dure environ trois minutes et 15 secondes. Les résultats de ces essais sont présentés à l'annexe II, accompagnés d'une description détaillée des événements observés pendant le déroulement du scénario. On y remarque un taux de succès de 95%, puisqu'un seul des essais effectués s'est soldé par un échec. Cet échec est dû à l'erreur insérée dans le positionnement du robot lorsque celui-ci a accroché le coin d'un rocher. Comme le robot estime sa position uniquement en fonction de la rotation de ses roues (*dead reckoning*), le changement de direction du robot dû à son contact avec le coin du rocher n'a pas été enregistré par l'estimateur de position en raison du glissement des roues lors de la rotation engendrée. On voit dans le tableau que le robot réel a aussi accroché le coin d'un rocher dans deux autres cas. Ces deux essais se sont tout de même soldés par un succès puisque le robot n'a pas dévié de sa trajectoire suffisamment pour l'empêcher de terminer sa tâche.

Ces tests ont donc permis de valider le fonctionnement de l'architecture dans son ensemble. Les prochaines sections vont maintenant présenter une analyse se concentrant sur différents aspects plus spécifiques de l'architecture.

5.1 Temps de développement

Cette section vise à vérifier si l'architecture a bien permis un développement plus rapide du système de contrôle des robots, et ce, sans limiter le développeur quant à sa façon d'implémenter la tâche à accomplir. Pour ce faire, les points suivants sont traités :

- le nombre de fichiers et de lignes de code nécessaires ;
- le temps de développement ;
- les connaissances requises pour le programmeur ;
- la liberté de choix offerte au programmeur.

Premièrement, regardons comment *AIDER* permet de minimiser la quantité de code à écrire lors du développement d'une tâche multi-robots. Ce paramètre est important puisqu'il a un impact direct sur le temps de développement et, encore plus important, il influence directement la probabilité d'insérer des erreurs dans le code. Il y a seulement deux endroits où un programmeur doit écrire du code lorsqu'il développe son système : à

l'intérieur d'une classe définissant une tâche, et à l'intérieur d'une classe définissant une ressource. Un troisième endroit peut s'ajouter dans les cas particuliers où l'on veut définir un type de message pouvant être échangé entre les tâches. Ce nombre limité d'endroits où l'on peut trouver le code est déjà un avantage pour le développeur puisqu'il permet rapidement de se retrouver, même pour un programmeur qui désire aller chercher de l'information ou modifier le code d'un autre.

De plus, les classes de base fournies par *AIDER* pour une tâche, une ressource ou un message contiennent déjà tous les mécanismes nécessaires pour gérer ces différents éléments. Pour le cas d'une tâche, la classe de base `Task` contient le code nécessaire à l'initialisation, la synchronisation, le contrôle et les changements d'état de la tâche. Tout ce code, caché au programmeur, permet à ce dernier de se concentrer uniquement sur le travail que doit accomplir sa tâche. Une seule fonction, *body()*, doit obligatoirement être écrite par le développeur lorsqu'il hérite de la classe de base. Dans cette fonction, il n'a qu'un nombre très restreint de lignes de code à écrire afin d'accéder aux paramètres de la tâche. En effet, l'accès aux paramètres par leur nom et, surtout, la vérification faite avant même l'initialisation de la tâche de la validité des paramètres en fonction des capacités du robot, permettent au programmeur de récupérer les valeurs voulues sans avoir à ajouter d'autre code de vérification ou de validation. De plus, lors de changements des plages de paramètres acceptés, il n'est pas nécessaire d'effectuer des modifications au fichier de code puisque seule une modification au fichier de configuration permet de tenir compte des nouvelles valeurs acceptées.

Une fois les paramètres de la tâche connus, le reste des lignes écrites dans la fonction *body()* servent directement à effectuer la tâche. À ce niveau, les fonctions de démarrage et de contrôle de sous-tâches permettent de réduire encore une fois la quantité de code à écrire en permettant de façon très simple de séparer la tâche en sous-tâches plus simples et d'obtenir facilement l'état final de ces sous-tâches lorsqu'elles se terminent. De plus, cette stratégie permet une bonne réutilisation de code à travers le développement de sous-tâches de base pouvant être réutilisées pour différentes tâches plus complexes. Par exemple, dans le scénario développé, la tâche `GotoLocation` est utilisée par la tâche `AnalyzeTarget`

ainsi que par la tâche `ExploreArea` (voir figure 4.5). La figure 5.1 donne un exemple de la quantité de code nécessaire et de sa simplicité pour permettre à un robot de se rendre à un point spécifié et d'y analyser le sol. Cette figure présente la fonction *body()* de la tâche `AnalyzeTarget`.

Il est courant de vouloir effectuer des actions particulières lorsqu'une tâche démarre, arrête ou est mise en pause. Le développement du scénario a permis de vérifier que l'ajout de ces différentes actions à une tâche se fait de façon très rapide. En effet, *AIDER* fournit déjà un point d'entrée pour chacune de ces situations en offrant les fonctions *myInit*, *myPausing*, *myPaused*, *myResume*, *myStopping* et *myTerminate*. La classe de base `Task` contient une version vide de ces fonctions ; le programmeur n'a donc qu'à les redéfinir dans sa propre tâche en y mettant le code qu'il désire faire exécuter lors de chacun des événements. Encore une fois, de nombreuses lignes de code sont épargnées puisque les mécanismes d'appel de ces fonctions ainsi que la synchronisation lors d'événements concurrents sont pris en charge par *AIDER*. Les figures 5.2 et 5.3 montrent des exemples d'utilisation de ces fonctions et démontrent leur simplicité.

La programmation des nouvelles ressources nécessaires au scénario a aussi permis de valider la capacité de *AIDER* à réduire la quantité de code devant être écrit par le développeur. En effet, comme dans le cas d'une tâche, l'architecture a permis de développer une ressource en ne se concentrant que sur les services fournis par cette ressource. Il suffit au développeur de définir une classe héritant de la classe de base `Resource` et d'y ajouter une fonction publique pour chacune des fonctionnalités qu'il désire offrir avec cette ressource. Il n'a donc qu'à programmer cette classe comme s'il s'agissait d'un simple objet qui sera utilisé à travers des appels de fonctions directs à partir d'un seul fil d'exécution. Nous savons que ces hypothèses ne sont pas réalistes dans un système robotique complexe, encore moins lorsqu'il s'agit d'un système multi-robots offrant le partage des ressources entre robots. Et c'est à ce niveau que *AIDER* est intéressante puisqu'elle permet au programmeur de développer sa ressource dans un contexte simplifié, puis elle utilise ses mécanismes internes afin de permettre à cette ressource d'être utilisée dans un contexte où des tâches multiples s'exécutant sur des robots différents peuvent tous accéder à cette ressource simultanément.

```
protected void body() throws BehaviourStoppedException
{
    Collection  gotoLocationParameters = new ArrayList(3);
    /* Gets the target position from parameters.*/
    Double paramX =
     (Double)Parameter.findByName( "X", parameters ).getValue();
    Double paramY =
     (Double)Parameter.findByName( "Y", parameters ).getValue();

    try {
        /*****************************************/
        /* Go to the location to analyse           */
        /*****************************************/
        gotoLocationParameters = new ArrayList(2);
        gotoLocationParameters.add( new FloatParameter( "X", paramX ) );
        gotoLocationParameters.add( new FloatParameter( "Y", paramY ) );
        gotoLocationBeh = spawnChild( "Goto",
         gotoLocationParameters,
         getPriority() );
        gotoLocationBeh.start();

        waitForChild( gotoLocationBeh );
        checkPoint();
        if( gotoLocationBeh.getCompletionCode() !=
         CompletionCode.SUCCESS ) {
            complete(CompletionCode.FAILED);
        }
        /*****************************************/
        /* Do the analysis                         */
        /*****************************************/
        analyser.analyse();
    }
    catch( Exception ex ) {
        logger.post( MessageType.SEVERE,
         component_name,
         "Error while analsing: " + ex );
    }
}
```

Figure 5.1 – Exemple de la fonction principale d'une tâche

```
/** Cette méthode est exécutée une seule fois lors de l'initialisation
    de la tâche
 */
protected void myInit() throws Exception
{
    try
    {
        // Get access to the robot motion control and telemetry
       analyser = (SoilAnalyserInterface)
        resourceManager.requestResource( "SoilAnalyser", 1, this );

    }
    catch( ResourceNotAvailableException ex )
    {
        logger.post( MessageType.SEVERE,
         component_name,
         "Cannot get interface to a required resource: " + ex );
        throw ex;
    }
}
```

Figure 5.2 – Exemple de la fonction d'initialisation d'une tâche

```
/** Cette méthode est exécutée lorsque la tâche est mise en pause
 */
protected void myPaused() throws Exception
{
    try
    {
        /* Command the Robot to stop.*/
        robot.setTranslationVelocity( new Integer(0) );
    }
    catch (ResourceAccessException ex)
    {
        logger.post( MessageType.SEVERE,
                     component_name,
                     getServiceType() +
                     "-Error while accessing a required resource: " +
                     ex );
    }
}
```

Figure 5.3 – Exemple de la fonction de mise en pause d'une tâche

Cela est rendu possible grâce aux mécanismes d'accès distant aux ressources et de contrôle d'accès présentés à la section 3.3.3. Les deux contrôleurs d'accès inclus avec *AIDER*, soit la mise en queue des requêtes et le contrôleur par priorité dynamique, permettent de tenir compte d'un grand nombre de situations et le développeur n'a qu'à sélectionner le contrôleur désiré lors de la création de la ressource.

Nous avons donc vu que *AIDER* a un impact important sur la quantité de code à écrire lors du développement d'une tâche multi-robots. Comme mentionné précédemment, cela permet d'accélérer le développement. D'autres caractéristiques de l'architecture permettent aussi un développement plus rapide du système.

Le fichier de configuration est une de ces caractéristiques qui permettent de sauver beaucoup de temps, et ce pour plusieurs raisons. Dans un premier temps, ce fichier permet d'utiliser exactement le même code pour tous les robots. On n'a donc qu'une seule archive Java (.jar) à distribuer sur les différents robots et c'est uniquement le fichier de configuration qui distingue les robots entre eux. Le développeur n'a donc qu'un seul projet ou script de compilation à gérer, et il n'a qu'une seule compilation à effectuer même lorsqu'il veut appliquer une modification à tous les robots. Dans un deuxième temps, ce même fichier de configuration admet le changement de la majorité des paramètres d'un robot, sans avoir à recompiler quoi que ce soit. Au cours des cycles de développement d'un robot, ou plus encore d'une tâche multi-robots, il est fréquent d'avoir à effectuer un grand nombre d'essais successifs visant à optimiser certains paramètres. Dans un tel contexte, la possibilité d'effectuer chaque essai en n'ayant qu'à modifier la valeur du paramètre dans le fichier de configuration amène une économie de temps substantielle. Le développement du scénario a de plus montré qu'il est possible de franchir aisément certaines étapes majeures du processus de développement grâce à l'utilisation de ce mode de configuration. En effet, après avoir terminé de tester la tâche d'exploration (`ExploreArea`) à l'aide du simulateur, le simple fait de mettre à «0» le paramètre *UseSimulator* des ressources `RobotControl` et `RobotTelemetry` a permis de passer de la simulation au monde réel et d'ainsi observer le robot Pioneer en pleine action dans son nouveau rôle d'explorateur.

Nous avons vu au début de cette section, en traitant des avantages de *AIDER* quant à la quantité de code à écrire, que le développeur n'a qu'à écrire son code à un nombre restreint d'endroits, et que dans chacun de ces cas, il n'a qu'à se concentrer sur le travail à accomplir. Cela a pour effet de limiter les connaissances qui sont requises du développeur pour accomplir sa tâche. Le fait que tous les mécanismes de gestion des tâches, ressources et capacités soient pris en charge par l'architecture autorise un plus grand éventail de personnes à développer des tâches multi-robots avec *AIDER* puisque des connaissances avancées en synchronisation, gestion de tâches ou réseautique ne sont pas requises. Le développeur n'a pas non plus à avoir de connaissances au niveau des communications puisque le système de messagerie ne demande au programmeur que de définir le contenu de son message.

AIDER a donc facilité de plusieurs façons la tâche du programmeur. Dans la majorité des cas, lorsqu'une librairie ou une architecture est développée pour alléger la tâche du programmeur, cela se fait au détriment de la liberté de ce dernier à faire les choses à sa façon. Cet aspect a été tenu en compte au cours du développement de *AIDER*, mais il est souvent impossible d'éliminer toutes contraintes et un équilibre doit être trouvé entre la facilité d'utilisation et les fonctionnalités d'une part, et la liberté permise d'autre part. Nous allons donc présenter les limitations que l'architecture impose au programmeur ainsi que des exemples de cas où celui-ci jouit d'une liberté de choix.

Les principales limitations imposées par *AIDER* se situent au niveau de ce que le programmeur peut, ou ne peut pas, faire à l'intérieur d'une tâche. La première obligation du programmeur est d'inclure des points de vérification de l'état de la tâche à travers son code de façon à ce que la tâche puisse être arrêtée ou mise en pause. Une tâche ne doit pas s'exécuter pendant une trop longue durée sans passer par un tel point de vérification, sous peine de ralentir considérablement le temps de réaction du système. Cela implique que tout appel à des fonctions bloquantes pouvant avoir un long temps d'exécution doit être protégé par un délai d'annulation (*timeout*). Le programmeur inclut un point de vérification dans son code en appelant la fonction *checkPoint()* de la classe de base `Task`. Il est bon de noter que cette limitation est nécessaire pour tout système désirant arrêter une

tâche en cours d'exécution puisque seul le programmeur de la tâche sait à quel moment il est sécuritaire que celle-ci soit interrompue et que certains systèmes d'exploitation (notamment Windows) n'offrent tout simplement pas la possibilité de forcer l'arrêt d'un fil d'exécution de façon sécuritaire.

Une autre limitation vient du fait que le programmeur doit respecter le principe général de l'architecture, c'est-à-dire utiliser le concept des tâches et des ressources. Il doit de plus programmer chaque élément en héritant de la tâche de base correspondante ; la classe `Task` pour les tâches, la classe `Resource` pour les ressources et la classe `Message` pour les messages à être échangés entre les tâches. Le programmeur pourrait cependant décider d'utiliser une tâche ou une ressource gigantesque qui accomplirait tout le travail, mais ce faisant, il perdrait de nombreux avantages liés à l'utilisation de *AIDER*.

Malgré ces limitations, l'utilisateur de *AIDER* a tout de même une grande latitude quant à l'utilisation qu'il peut faire de l'architecture. Par exemple, en ce qui a trait au mécanisme d'allocation des tâches dans le groupe de robots, de nombreuses techniques existent et de nombreuses recherches sont d'ailleurs en cours dans ce domaine. On peut principalement regrouper ces techniques en deux groupes : les approches centralisées et les approches distribuées. Une solution élégante peut être développée dans les deux cas avec *AIDER* puisque celle-ci fournit d'une part les mécanismes permettant de connaître les capacités des membres du groupe nécessaire aux approches centralisées, et d'autre part une approche distribuée peut facilement être implémentée en développant une tâche de coordination utilisant le service de messagerie pour négocier la distribution des tâches avec les tâches de coordination équivalentes situées sur les autres robots.

Nous avons vu qu'à travers ses nombreux mécanismes, *AIDER* permet d'optimiser de façon notable le processus de développement d'une tâche pour un groupe de robots en réduisant la charge de travail du programmeur et en accélérant chacun des cycles de développement. *AIDER* a de plus réussi à atteindre cet objectif tout en gardant au minimum les contraintes imposées au programmeur et en lui offrant un large éventail de possibilités.

5.2 Performances

Il est important pour une architecture comme *AIDER* d'offrir un bon niveau de performance. Trop souvent, les systèmes offrant un grand nombre de fonctionnalités présentent aussi une certaine lourdeur et sont très exigeants au niveau des ressources du système. Évidemment, lorsque l'on parle d'un système robotique, ces ressources sont souvent limitées et il est donc important d'avoir une architecture efficace.

Dans cette optique, différents tests ont été conçus et réalisés afin de vérifier les performances de *AIDER* en ce qui a trait à l'utilisation du processeur et de la mémoire, à l'utilisation de la bande passante du réseau et au temps de réaction du système. Les prochaines sous-sections présentent ces différents tests ainsi que les résultats obtenus.

5.2.1 Utilisation du processeur

Le premier point qui a été évalué est l'utilisation du processeur et de la mémoire lorsque *AIDER* est en fonction. L'évaluation de cet aspect étant directement liée aux caractéristiques de l'ordinateur sur lequel s'exécute les tests, le lecteur peut se référer au tableau 5.1 qui présente les caractéristiques des trois ordinateurs utilisés, soit l'ordinateur embarqué du robot Pioneer (Robot 1), l'ordinateur simulant un robot mobile (Robot 2) et l'ordinateur de la station fixe (Robot 3). Dans le cas de l'ordinateurs fonctionnant sous le système d'exploitation Windows 2000, les données ont été recueillies à l'aide de l'application *Gestionnaire des tâches Windows*. Dans le cas des ordinateurs fonctionnant sous Linux, c'est l'utilitaire *top* qui a permis d'effectuer les mesures.

TABLEAU 5.1 – Caractéristiques des ordinateurs utilisés

	Description	Processeur	Mémoire vive	Système d'exploitation
Robot 1	Robot Pioneer	Intel Pentium III à 850 MHz	96Mb	Linux 2.4.19
Robot 2	Robot mobile simulé	Intel Xeon à 2,4 GHz	1Gb	Linux 1.4.1814
Robot 3	Station fixe	Intel Pentium III Mobile à 750MHz	512Mb	Windows 2000

Les taux d'utilisation des processeurs et de la mémoire ont été mesurés dans quatre situations différentes. Dans chacune de ces quatre situations, les trois robots du scénario sont présents et s'échangent leur identification et leurs capacités tel qu'indiqué à la section 3.3.1. Les tableaux 5.2 à 5.5 présentent le résumé des résultats obtenus pour chaque situation.

TABLEAU 5.2 – Performance pour la situation # 1

	% CPU moyen	% CPU max.	Mémoire (MB)
Robot 1	0,5	2,0	35
Robot 2	0	1,5	34
Robot 3	2	4	30

TABLEAU 5.3 – Performance pour la situation # 2

	% CPU moyen	% CPU max.	Mémoire (MB)
Robot 1	0,5	2,0	36
Robot 2	0	1,5	44
Robot 3	2	4	36

TABLEAU 5.4 – Performance pour la situation # 3

	% CPU moyen	% CPU max.	Mémoire (MB)
Robot 1	1	2,5	28
Robot 2	1	2	37
Robot 3	2	4	22

Dans la situation #1, les robots n'accomplissent aucune autre tâche que l'échange de capacités. Ces mesures représentent donc la charge que *AIDER* impose au processeur lorsque le système est au repos. Comme il est souhaitable dans ce genre de situation, l'utilisation du processeur mesurée est restée très minime. En effet, dans le cas du robot 1, l'utilisation moyenne du processeur est de 0,5% avec de très brefs sommets à 2,0%. Dans le cas du robot 2, l'utilisation du processeur était négligeable avec un taux moyen affiché de 0%. Comme dans le cas du premier robot, il y avait de brefs sommets d'utilisation à 1,5%. Le robot 3 avait quant à lui une utilisation moyenne du processeur de 2% avec des sommets à 4%. Le processeur est presque uniquement utilisé par le mécanisme d'échange de capacités, et les sommets d'utilisation du processeur mesurés correspondent au moment où

TABLEAU 5.5 – Performance pour la situation # 4

	% CPU moyen	% CPU max.	Mémoire (MB)
Robot 1	13	28	48
Robot 2	12	25	54
Robot 3	2	100	52

les robots envoient ou reçoivent une liste de capacités. La mince différence mesurée entre les trois robots s'explique par la différence de puissance des processeurs de ces robots. Notons que l'intervalle entre les envois de capacités peut être configuré et que dans toutes les situations qui ont été analysées, cet intervalle était fixé à 15 secondes.

La situation #2 qui a été analysée correspond à un système actif en attente d'une commande. Il y a pour ce test trois nouvelles tâches actives, soit une sur chacun des robots. Dans le cas du robot réel, le robot 1, cette nouvelle tâche est sa tâche de mise à jour de sa position auprès du simulateur. Pour ce qui est des deux autres robots, chacun exécute une tâche `TaskViewer` qui affiche une interface graphique permettant de voir les arbres de tâches de tous les robots du système et de lancer de nouvelles tâches sur le robot désiré. Les données d'utilisation du processeur dans cette situation étaient les mêmes que lors de la situation précédente, ce qui montre que les tâches de gestion que nous avons ajoutées ont une influence négligeable sur la puissance de calcul utilisée.

Pour la situation #3, les mesures ont été prises alors qu'une tâche de déplacement (*MoveInLine*) a été lancée sur le robot 1 à partir de la tâche `TaskViewer` s'exécutant sur le robot 2. Cela permet de vérifier la charge qu'exerce sur le processeur le contrôle d'une tâche distante. Il en résulte une augmentation de l'utilisation moyenne du processeur du robot 2 de 1% et son utilisation maximale de 0,5%. L'exécution de la tâche de déplacement par le robot 1 a, quant à elle, augmenté l'utilisation du processeur de ce robot de 1%, portant son utilisation moyenne à 1% et son utilisation maximale à 2,5%. Les données pour le robot 3 sont restées inchangées.

La situation #4 étudiée est l'exécution complète du scénario. Cet essais permet de mesurer l'utilisation du processeur des ordinateurs du système durant une période d'activité intense durant laquelle des dizaines de tâches sont démarrées, arrêtées, mises en pause, etc. Dans

le cas du robot 1, l'utilisation moyenne du processeur au cours du scénario a été de 13% avec des sommets d'utilisation à 28%. Le processeur du robot 2 a quant à lui été utilisé en moyenne à 12% de sa capacité, avec quelques sommets à 25%. L'utilisation moyenne du processeur du robot 3 est restée sensiblement la même, à 2%, puisque ce robot est beaucoup moins actif que les deux autres pendant l'exploration, se contentant d'analyser le terrain avec sa caméra afin d'y repérer des cibles. L'utilisation du processeur atteint cependant des sommets de 100% chaque fois que l'algorithme de vision est mis en marche pour analyser une image.

En ce qui concerne l'utilisation de la mémoire, les résultats obtenus indiquent une utilisation de mémoire qui, bien qu'elle soit acceptable, est plus élevée que les attentes initiales. Une utilisation de mémoire inférieure à 20 MB était souhaitée. Une étude plus approfondie de l'utilisation de la mémoire a donc été entreprise afin de déterminer quelles sont les sections qui utilisent le plus de mémoire. Différentes sections de code (des tâches, des ressources, des modules internes de *AIDER*) ont été enlevées les unes après les autres afin de déterminer leur utilisation de mémoire. Il en est ressorti que la majorité de la mémoire était utilisée par des modules externes à *AIDER*, comme par exemple le générateur de trajectoires (*PathPlanner*), qui utilise à lui seul environ 12 MB de mémoire. Le tableau 5.6 présente les modules qui utilisent le plus de mémoire. Notons que tous les tests de répartition de mémoire ont été effectués sous le système d'exploitation Windows 2000 et qu'il peut y avoir des différences significatives dans la quantité de mémoire utilisée lorsqu'un autre système d'exploitation est employé. Par exemple, il a été noté que la tâche `TaskViewer`, qui fournit une interface graphique, utilise 6 MB de mémoire sur Windows alors qu'elle en utilise 10 MB sur Linux.

Il ressort donc qu'un nombre restreint d'éléments utilisent la majorité de la mémoire. De plus, aucun des éléments présents dans ce tableau ne fait partie directement de *AIDER*. En effet, ils sont tous soit des librairies développées par d'autres avant le développement de *AIDER* (Path Planner, Castor, Logger, Vision), soit une tâche utilisant une interface graphique facultative, ou bien ils sont imposés par le langage Java. Si l'on retranche ces éléments de la mémoire totale utilisée par le système, on se rend compte que l'architecture

TABLEAU 5.6 – Principaux modules utilisateurs de mémoire

Module	Mémoire utilisée (MB)	Notes
Ressource PathPlanner	12	Générateur de trajectoires.
Environnement JAVA	7	Mémoire utilisée par Java pour une application vide.
Tâche TaskViewer	6	La mémoire est surtout utilisée par l'interface graphique de la tâche.
Librairie Castor	4,5	Pour la lecture des fichiers de configuration en format XML.
Logger	1,5	Garde une trace de tous les messages.
Ressource Vision	1,5	Lors d'une analyse, la mémoire utilisée grimpe à 17MB.

AIDER en elle-même utilise moins de 2 MB de mémoire, 9 MB si on inclut l'environnement Java. Chaque nouvelle tâche ajoutée à un système en marche requiert 35 kB de plus lorsque la tâche est locale et 75 kB de plus lorsqu'il s'agit d'une tâche distante.

Ces données recueillies sur l'utilisation des processeurs et de la mémoire permettent de conclure en l'efficacité de *AIDER* à ce niveau. Le système n'impose en effet qu'une charge négligeable sur le processeur lorsque celui-ci est au repos complet ou même en attente d'une commande. De plus, l'utilisation du processeur lorsque le système est en pleine activité reste très faible, ce qui indique que *AIDER* pourrait être utilisée sur des robots possédant des puissances de calcul beaucoup plus faibles. L'utilisation de la mémoire restait aussi à des niveaux acceptables, et il a été démontré que la majeure partie de la mémoire utilisée l'était par des librairies déjà existantes qui ont été utilisées lors de l'implémentation du scénario.

5.2.2 Utilisation de la bande passante

AIDER est une architecture d'interactions multi-robots dans laquelle une communication est nécessaire entre les robots. La bande passante disponible étant bien souvent limitée, il est important de garder le débit de communication du système le plus bas possible. L'utilisation de la bande passante du réseau a donc été mesurée dans chacune des quatre

situations décrites à la section 5.2.1. Dans la situation #1, les seules communications qui ont lieu sont les échanges de capacités entre les robots. Ces échanges se tiennent à intervalles réguliers (configuré à 15 secondes dans notre cas) et il y a donc une utilisation de la bande passante de façon sporadique, à chaque intervalle. La quantité de données envoyées à chacun de ces intervalles est dépendante du nombre et de la nature des capacités du robot. Les trois robots du scénario donnent à cet effet un exemple qui se rapproche de la réalité. Pour les robots 1 et 2, 3,96 kB de données sont envoyées à chaque 15 secondes, ce qui donne une utilisation moyenne de 0,264 kB/s. Le robot 3 ayant moins de capacités à transmettre, il génère un trafic de 0,168 kB/s. Les trois robots combinés génèrent donc un total de 0,696 kB/s sur le réseau. Il faut aussi tenir compte des messages de mise à jour des données envoyés par le simulateur *Stage*. Celui-ci génère un trafic d'environ 2,10 kB/s, ce qui porte le total à 2,80 kB/s.

Dans la situation #2, deux nouvelles sources de trafic s'ajoutent, soit la tâche de mise à jour du simulateur s'exécutant sur le robot 1 et les deux tâches `TaskViewer` s'exécutant sur les robots 2 et 3. L'apport de la tâche de mise à jour du simulateur est négligeable, ne représentant que quelques octets par secondes. Les tâches `TaskViewer` génèrent quant à elles un trafic de 3,9 kB chacune à toutes les 20 secondes ce qui donne un taux moyen de 0,195 kB/s. Le total pour cette situation est donc de 3,00 kB/s.

La situation #3 est pratiquement identique à la deuxième au niveau de l'utilisation de la bande passante puisque la seule communication qui est ajoutée est une courte série de messages comprenant la requête de tâche, la réponse, le démarrage de la tâche, l'inscription aux changements d'état de la tâche et les messages de changements d'état eux-mêmes. Cela représente un envoi ponctuel de 4,72 kB ce qui, pris de façon isolée, est négligeable au niveau de l'utilisation moyenne de la bande passante. Cela démontre néanmoins que même lors de l'utilisation de tâches distantes, la quantité d'information à transférer sur le réseau est gardée au minimum.

Alors que les trois premières situations ont permis de mesurer l'utilisation de la bande passante par différents mécanismes de *AIDER*, la situation #4, celle du scénario complet, permet plutôt d'avoir un exemple concret de l'utilisation de la bande passante lorsque le

système est en pleine exécution d'une tâche complexe. Les mesures pour cette situation donnent un taux moyen d'utilisation de la bande passante de 5,0 kB/s.

L'utilisation du réseau par *AIDER* reste donc à un niveau faible, même lors d'une activité intense. Cet aspect ne représente donc pas une limitation pour l'architecture puisqu'une telle bande passante est généralement disponible sur un robot. De plus, l'usage que fait *AIDER* de la bande passante croît de façon linéaire, et non exponentielle, en fonction du nombre de robots. Il est donc possible d'avoir dans un même système un nombre élevé de robots collaborant entre eux.

5.2.3 Temps de réaction

Un autre aspect important d'une architecture d'interactions multi-robots est que celle-ci réagisse rapidement lorsqu'une nouvelle tâche est demandée. Cela est particulièrement important lorsque l'on sait qu'une tâche complexe, comme l'exploration d'une zone telle qu'exécutée dans le scénario, demande le démarrage de plus de 200 tâches sur chacun des robots mobiles. Des tests ont donc été effectués afin de déterminer le temps de réaction lors du démarrage d'une tâche. Il n'est pas utile d'effectuer des mesures de temps de réaction pour la mise en pause ou l'arrêt d'une tâche puisque ces temps varient d'une tâche à l'autre, étant directement liés à l'emplacement des points de vérification (*checkpoint*) dans le code.

Les temps de réaction ont d'abord été déterminés pour le démarrage d'une tâche locale. Le démarrage d'une tâche est en réalité constitué de deux étapes : la création de la tâche et son démarrage. Les temps mesurés correspondent au total de ces deux étapes, i.e. du moment où la fonction de requête de tâche est appelée, jusqu'au moment où la première ligne de code de la fonction *body()* de la tâche est exécutée. Vingt tests successifs ont été réalisés et ont donné des temps variant de 8,4 ms à 14,4 ms. La moyenne des vingt échantillons est de 10,6 ms.

Dans le cas du démarrage d'une tâche distante, les temps de réaction sont significativement plus grands à cause des messages de contrôle qui doivent être échangés entre les robots. Encore une fois, vingt mesures ont été prises, donnant des temps de réaction variant entre 60 ms et 132 ms, pour une moyenne de 95 ms. Une partie non négligeable de ces temps

est due au délai de transmission des paquets IP sur le réseau. Cette portion du temps de réaction peut être très variable en fonction du type de réseau utilisé et de l'achalandage de ce dernier. C'est pourquoi ces temps de transmission ont été isolés afin de connaître la partie constante des temps de réaction. Au cours de chacun des tests effectués, des mesures supplémentaires ont été prises afin de pouvoir déterminer ces temps de transmission. La moyenne des temps de transmission mesurés était de 27 ms. Sur le temps de réaction total moyen de 95 ms, 68 ms sont donc engendrées par le traitement du démarrage d'une tâche distante dans *AIDER*. Les résultats complets des tests de temps de réaction sont présentés à l'annexe III.

Les temps de réactions très courts mesurés démontrent que *AIDER* malgré toutes les fonctionnalités qu'elle offre, n'impose pas de lourdeur et permet au système de réagir très rapidement à de nouvelles situations.

5.3 Travaux futurs

Le développement du scénario d'évaluation a permis de vérifier l'atteinte des objectifs initiaux de *AIDER*. Cette première période de la vie de l'architecture a donc été complétée avec succès. Il faut dès lors profiter de cet outil qui a été développé et de l'expérience acquise afin de pousser plus loin les travaux sur les groupes de robots. Dans cette optique, cette section présente les travaux futurs liés à *AIDER*, que ce soit l'utilisation de celle-ci à diverses fins, ou encore une amélioration de l'architecture en fonction de l'expérience acquise au cours du développement du scénario.

Comme il a été démontré dans les sections précédentes, *AIDER* offre une structure de base solide qui permet de développer rapidement des tâches multi-robots. Une prochaine étape logique est donc le développement d'une telle tâche à l'aide de *AIDER* par un groupe de développeurs externes, i.e. n'ayant pas participé à la création de l'architecture. En effet, le développeur d'une architecture n'est pas la meilleure personne pour évaluer son fonctionnement puisque la conception a été faite en fonction de ses propres besoins et en suivant sa propre façon de penser. C'est pourquoi l'utilisation de l'architecture par

une autre personne ou un autre groupe est souhaitée. Tout en sauvant du temps à ces développeurs, cela permettra d'évaluer *AIDER* sous un autre angle et d'ainsi tester sa réelle utilité.

Outre le développement d'autres tâches multi-robots, *AIDER* peut aussi être utilisée afin d'effectuer des recherches portant sur des aspects précis de l'intelligence des robots. Comme *AIDER* fournit un système de contrôle complet et fonctionnel, les chercheurs peuvent se consacrer entièrement à l'aspect novateur de leurs recherches. L'architecture pourrait par exemple être utilisée afin de tester différentes techniques de répartition des tâches dans un groupe ou encore tester de nouveaux mécanismes d'arbitrage entre tâches actives simultanément. À ce sujet, le développement du scénario a déjà permis de valider le fonctionnement de base du mécanisme d'arbitrage par priorité dynamique présenté à la section 3.3.3. Cependant, l'utilisation qui a été faite de ce mécanisme dans le scénario était assez simple, cette technique n'étant pas le sujet principal de ce travail. Une étude plus poussée de ce mécanisme serait donc souhaitable et pourrait faire l'objet de travaux futurs.

Le développement du scénario a permis d'entrevoir quelques améliorations qui pourraient être apportées à l'architecture. La première amélioration concerne l'utilisation des fichiers de configuration. Ces derniers ont été très pratiques tout au long du développement et ont permis de sauver beaucoup de temps. Ces avantages auraient pu être encore plus grands s'il avait été possible de changer la configuration en temps réel, alors que le robot est en action. Une analyse rigoureuse serait nécessaire afin d'évaluer les impacts de ces changements de configuration pouvant survenir à tout moment, mais la fonctionnalité serait très utile.

Une autre amélioration possible de *AIDER* serait l'ajout de dépendances entre les tâches, comme cela existe déjà au niveau des ressources. De plus, pour l'instant les dépendances aux ressources ne tiennent compte que des ressources locales, alors que dans certains cas, il pourrait être souhaitable de tenir compte des ressources de tout le système.

Ce ne sont là que quelques exemples des avenues futures qu'il sera possible d'explorer afin de poursuivre le travail qui a mené au développement de *AIDER* et de permettre le développement de tâches pour groupe de robots de plus en plus complexes.

CHAPITRE 6

CONCLUSION

Ce mémoire présente une architecture logicielle permettant la coopération dans un groupe de robots. Ce projet vise à fournir des mécanismes rendus nécessaires par l'augmentation des interactions entre les robots découlant de leur présence en plus grand nombre dans notre environnement. L'architecture conçue, nommée *AIDER*, permet le développement rapide de nouvelles tâches multi-robots. Elle a été développée avec comme objectif de diminuer les temps de développement et de faciliter le travail du programmeur. Elle se veut aussi une architecture flexible qui offre au programmeur une grande liberté quant à sa façon d'aborder les problèmes liés au développement de sa tâche.

AIDER permet différents types d'interactions entre les robots grâce à ses quatre principaux modules : le gestionnaire de capacités, le gestionnaire de ressources, le gestionnaire de tâches et le gestionnaire de messages. Le gestionnaire de capacités permet aux robots de connaître en tout temps les capacités des autres membres du groupe, facilitant ainsi la séparation des tâches dans le groupe. Le gestionnaire de tâches permet de son côté la création et le démarrage de tâches, et ce autant sur le robot local que sur un robot distant, permettant ainsi de créer des arbres de tâches distribués sur plusieurs robots. Ce gestionnaire rend transparent l'interaction avec ces tâches, si bien qu'une tâche distante est contrôlée par sa tâche-parent exactement de la même façon qu'une tâche locale.

Le gestionnaire de ressources permet quant à lui le partage transparent des ressources entre les robots. Un robot peut donc utiliser la ressource d'un autre robot exactement comme si c'était une de ses propres ressources. Des contrôleurs d'accès sont associés à chaque tâche afin d'effectuer l'arbitrage lors de l'accès simultané par deux tâches différentes. Finalement, le gestionnaire de messages permet une interaction flexible entre les

tâches en offrant un canal de communication permettant deux modes d'envois : l'envoi direct à un destinataire et un mécanisme d'inscription permettant la diffusion de certains types de messages seulement aux abonnés. Ensemble, ces quatre composants de *AIDER* permettent une interaction entre les robots à tous les niveaux, du partage de leurs moteurs à l'allocation des tâches entre eux.

Un scénario a été développé afin de valider l'architecture développée et de tester ses fonctionnalités. Ce scénario a été implémenté avec succès à l'aide de *AIDER* et a pu confirmer les avantages liés à l'utilisation de l'architecture. Cette dernière a en effet permis d'accélérer le développement en diminuant significativement la quantité de code à écrire et en offrant une méthode de configuration permettant d'importantes économies de temps. Les tests effectués ont aussi permis d'établir la grande efficacité de l'architecture au niveau des temps de réaction et de l'utilisation du processeur, de la mémoire et de la bande passante.

Nous pouvons donc conclure que les objectifs ont été atteints puisque *AIDER* permet réellement un développement rapide de tâches multi-robots, tout en restant flexible et en ayant des caractéristiques de performance lui permettant d'être utilisée dans la grande majorité des contextes. Elle pourra dès lors être utilisée pour développer des groupes de robots ou pour poursuivre la recherche dans le domaine de l'intelligence des robots afin que ceux-ci puissent répondre aux attentes de notre société au cours des prochaines décennies.

ANNEXE I - FICHIER DE CONFIGURATION DU ROBOT RÉEL

(RobotReal.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<robot-definition xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <robot-identification robot-id="99" inet-port="10052">
        <robot-name>P2Rover</robot-name>
    </robot-identification>
    <resource-definitions xsi:type="resource-definition" resource-id="1">
        <resource-name>P2Controller</resource-name>
        <parameter name="UseSimulator" value="0" xsi:type="integer-parameter"/>
        <parameter name="RobotName" value="P2real" xsi:type="string-parameter"/>
        <parameter name="ResetX" value="3.5" xsi:type="float-parameter"/>
        <parameter name="ResetY" value="0.25" xsi:type="float-parameter"/>
        <parameter name="ResetTheta" value="180" xsi:type="integer-parameter"/>
        <implementing-class-name>ca.gc.space.mrt.resources.Res_P2Controller</implementing-class-name>
        <interface-class-name>ca.gc.space.mrt.resources.P2ControllerInterface</interface-class-name>
    </resource-definitions>
    <resource-definitions xsi:type="resource-definition" resource-id="1">
        <resource-name>P2Telemetry</resource-name>
        <parameter name="UseSimulator" value="0" xsi:type="integer-parameter"></parameter>
        <parameter name="RobotName" value="P2real" xsi:type="string-parameter"></parameter>
        <implementing-class-name>ca.gc.space.mrt.resources.Res_P2Telemetry</implementing-class-name>
        <interface-class-name>ca.gc.space.mrt.resources.P2TelemetryInterface</interface-class-name>
    </resource-definitions>
    <resource-definitions xsi:type="resource-definition" resource-id="2">
        <resource-name>P2Controller</resource-name>
        <parameter name="UseSimulator" value="2" xsi:type="integer-parameter"/>
        <parameter name="RobotName" value="P2Sim" xsi:type="string-parameter"/>
        <parameter name="IpAddr" value="ewok.speng.space.gc.ca" xsi:type="string-parameter"/>
        <parameter name="Port" value="6678" xsi:type="integer-parameter"/>
        <parameter name="ResetSimulation" value="1" xsi:type="integer-parameter"/>
        <implementing-class-name>ca.gc.space.mrt.resources.Res_P2Controller</implementing-class-name>
        <interface-class-name>ca.gc.space.mrt.resources.P2ControllerInterface</interface-class-name>
    </resource-definitions>
    <resource-definitions xsi:type="resource-definition" resource-id="1">
        <resource-name>PathPlanner</resource-name>
        <parameter name="MapFilename" value="config/hb_terrain.pnm" xsi:type="string-parameter"></parameter>
        <implementing-class-name>ca.gc.space.mrt.resources.Res_PathPlanner</implementing-class-name>
        <interface-class-name>ca.gc.space.mrt.resources.PathPlannerInterface</interface-class-name>
    </resource-definitions>
    <resource-definitions xsi:type="resource-definition" resource-id="1">
        <resource-name>SoilAnalyser</resource-name>
        <implementing-class-name>ca.gc.space.mrt.resources.Res_SoilAnalyser</implementing-class-name>
        <interface-class-name>ca.gc.space.mrt.resources.SoilAnalyserInterface</interface-class-name>
    </resource-definitions>
    <capability-definitions confidence-level="30" xsi:type="capability">
        <service-type>TaskViewer</service-type>
        <implementing-class-name>
            ca.gc.space.argo.ai.robotcontrol.behaviours.Beh_TaskTreeViewer
        </implementing-class-name>
    </capability-definitions>
    <capability-definitions confidence-level="60" xsi:type="capability">
        <service-type>MoveInLine</service-type>
        <param-range min-value="-200" max-value="200" required="true" xsi:type="float-range">
            <name>Distance</name>
        </param-range>
```

```
    &lt;implementing-class-name&gt;ca.gc.space.mrt.behaviour.Beh_MoveInLine&lt;/implementing-class-name&gt;
&lt;/capability-definitions&gt;
&lt;capability-definitions confidence-level="50" xsi:type="capability"&gt;
    &lt;service-type&gt;Turn&lt;/service-type&gt;
    &lt;param-range min-value="-180" max-value="180" required="true" xsi:type="float-range"&gt;
        &lt;name&gt;Angle&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;implementing-class-name&gt;ca.gc.space.mrt.behaviour.Beh_TurnOnTheSpot&lt;/implementing-class-name&gt;
&lt;/capability-definitions&gt;
&lt;capability-definitions confidence-level="50" xsi:type="capability"&gt;
    &lt;service-type&gt;FollowSegment&lt;/service-type&gt;
    &lt;param-range min-value="-200" max-value="200" required="true" xsi:type="float-range"&gt;
        &lt;name&gt;X&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;param-range min-value="-200" max-value="200" required="true" xsi:type="float-range"&gt;
        &lt;name&gt;Y&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;param-range min-value="-180" max-value="180" required="false" xsi:type="integer-range"&gt;
        &lt;name&gt;Theta&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;implementing-class-name&gt;ca.gc.space.mrt.behaviour.Beh_FollowSegment&lt;/implementing-class-name&gt;
&lt;/capability-definitions&gt;
&lt;capability-definitions confidence-level="50" xsi:type="capability"&gt;
    &lt;service-type&gt;Goto&lt;/service-type&gt;
    &lt;param-range min-value="-200" max-value="200" required="true" xsi:type="float-range"&gt;
        &lt;name&gt;X&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;param-range min-value="-200" max-value="200" required="true" xsi:type="float-range"&gt;
        &lt;name&gt;Y&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;param-range min-value="-179" max-value="180" required="false" xsi:type="integer-range"&gt;
        &lt;name&gt;Theta&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;implementing-class-name&gt;ca.gc.space.mrt.behaviour.Beh_GoToLocation&lt;/implementing-class-name&gt;
&lt;/capability-definitions&gt;
&lt;capability-definitions confidence-level="80" xsi:type="capability"&gt;
    &lt;service-type&gt;AnalyseSoil&lt;/service-type&gt;
    &lt;param-range min-value="-200" max-value="200" required="true" xsi:type="float-range"&gt;
        &lt;name&gt;X&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;param-range min-value="-200" max-value="200" required="true" xsi:type="float-range"&gt;
        &lt;name&gt;Y&lt;/name&gt;
    &lt;/param-range&gt;
    &lt;implementing-class-name&gt;ca.gc.space.mrt.behaviour.Beh_AnalyseSoil&lt;/implementing-class-name&gt;
    &lt;resource-dependency resource-id="0" xsi:type="resource-dependency"&gt;
        &lt;resource-name&gt;SoilAnalyser&lt;/resource-name&gt;
    &lt;/resource-dependency&gt;
&lt;/capability-definitions&gt;
&lt;capability-definitions confidence-level="80" xsi:type="capability"&gt;
    &lt;service-type&gt;FindTarget&lt;/service-type&gt;
    &lt;implementing-class-name&gt;ca.gc.space.mrt.behaviour.Beh_FindTargetDummy&lt;/implementing-class-name&gt;
&lt;/capability-definitions&gt;
&lt;capability-definitions confidence-level="50" xsi:type="capability"&gt;
    &lt;service-type&gt;DetectObstacles&lt;/service-type&gt;
    &lt;implementing-class-name&gt;ca.gc.space.mrt.behaviour.Beh_DetectObstacles&lt;/implementing-class-name&gt;
&lt;/capability-definitions&gt;
&lt;capability-definitions confidence-level="90" xsi:type="capability"&gt;
    &lt;service-type&gt;UpdateSim&lt;/service-type&gt;
    &lt;implementing-class-name&gt;
        ca.gc.space.mrt.behaviour.Beh_UpdateSimulatorPosition
    &lt;/implementing-class-name&gt;
&lt;/capability-definitions&gt;
&lt;capability-definitions confidence-level="50" xsi:type="capability"&gt;
    &lt;service-type&gt;ExploreArea&lt;/service-type&gt;
    &lt;implementing-class-name&gt;ca.gc.space.mrt.behaviour.Beh_ExploreArea&lt;/implementing-class-name&gt;
    &lt;param-range min-value="-200" max-value="200" required="true" xsi:type="float-range"&gt;
        &lt;name&gt;X&lt;/name&gt;
    &lt;/param-range&gt;
```

```
        <param-range min-value="-200" max-value="200" required="true" xsi:type="float-range">
            <name>Y</name>
        </param-range>
        <param-range min-value="0" max-value="200" required="true" xsi:type="float-range">
            <name>Width</name>
        </param-range>
        <param-range min-value="0" max-value="200" required="true" xsi:type="float-range">
            <name>Height</name>
        </param-range>
        <param-range required="false" xsi:type="string-range">
            <name>Team</name>
        </param-range>
    </capability-definitions>
    <capability-definitions confidence-level="99" xsi:type="capability">
        <service-type>DisplayTelemetry</service-type>
        <implementing-class-name>ca.gc.space.mrt.behaviour.Beh_DisplayTelemetry</implementing-class-name>
    </capability-definitions>
    <startup-tasks xsi:type="task-description" resource-id="1">
        <service-type>UpdateSim</service-type>
    </startup-tasks>
</robot-definition>
```

ANNEXE II - RÉSULTATS DES TESTS DU SCÉNARIO

Déroulement du scénario

Voici, étape par étape, les différents événements observés au cours du déroulement des essais d'exécution du scénario, s'exécutant avec succès.

- Le robot 1 démarre la tâche `UpdateSim` dès qu'il est initialisé (tel qu'indiqué dans son fichier de configuration). Le robot 2 démarre la tâche `TaskViewer` dès qu'il est initialisé (tel qu'indiqué dans son fichier de configuration). (La figure II.1 présente les arbres de tâches des trois robots à ce moment.)
- La requête d'exploration est envoyée au robot 2 à l'aide de l'interface graphique de `TaskViewer`. Le robot 2 démarre la tâche `ExploreArea`. Cette tâche, que nous appellerons tâche-parent d'exploration, examine les capacités des trois robots puis sépare la zone à explorer en deux parties. La tâche-parent d'exploration démarre `ExploreArea` sur le robot 1 en lui assignant la moitié du terrain et démarre `ExploreArea` sur le robot 2 en lui assignant l'autre moitié du terrain. Les deux robots se mettent en marche afin d'explorer leur portion du terrain. Pour ce faire, ils lancent eux mêmes une série de sous-tâches `GotoLocation` et `FindTarget`. Les tâches `GotoLocation` sont décomposées en une série de sous-tâche `FollowSegement`, qui sont elles-mêmes exécutées en utilisant une combinaison des tâches `MoveInLine`, `Turn` et `DetectObstacle`.
- La tâche-parent d'exploration démarre `FindTarget` sur le robot 3. Ce robot commence à analyser le terrain à l'aide de sa caméra. (La figure II.2 présente les arbres de tâches des trois robots à ce moment.)
- Pendant ce temps, la tâche-parent d'exploration reçoit les messages de mise à jour de l'exploration de ses différentes sous-tâches, met à jour sa carte d'exploration, et attend que les sous-tâches se terminent. Le robot 1 trouve une cible et démarre son analyse. (La figure II.3 présente les arbres de tâches des trois robots à ce moment.)
- Le robot 1 complète l'analyse de la cible et poursuit son exploration. Le robot 3 trouve une cible et l'indique à la tâche-parent. Il poursuit sa recherche de cible.
- Le robot 3 termine sa recherche de cible. Comme celui-ci n'offre pas le service d'analyse de cibles (la station fixe n'a pas cette capacité), la tâche-parent n'a plus de tâche à lui assigner. Le robot 1 trouve une deuxième cible et démarre son analyse. L'analyseur du robot 1 tombe en panne au cours de l'analyse et il est retiré des ressources disponibles. Les capacités du robot 1 sont mises à jour et celui-ci ne peut plus offrir le service `AnalyzeTarget`. Le robot 1 poursuit ensuite son exploration.
- Le robot 2 termine son exploration. Voyant qu'il est libre et qu'il est capable d'analyser des cibles, la tâche-parent lui assigne l'analyse de la cible trouvée par le robot 3. Il se déplace vers l'emplacement de la cible pour l'analyser. (La figure II.4 présente les arbres de tâches des trois robots à ce moment.)

- Le robot 1 termine son exploration. Comme celui-ci n'offre plus le service d'analyse de cibles à cause de la panne de son analyseur, la tâche-parent n'a plus de tâche à lui assigner. Le robot 2 termine son analyse. Voyant qu'il est libre et qu'il est capable d'analyser des cibles, la tâche-parent lui assigne l'analyse de la cible que le robot 1 n'avait pas été capable d'analyser. Il se déplace vers l'emplacement de la cible pour l'analyser. (La figure II.5 présente les arbres de tâches des trois robots à ce moment.)
- Le robot 2 termine son analyse. Voyant que tout le terrain a été exploré et que toutes les cibles trouvées ont été analysées, la tâche-parent se termine.

Compilation des résultats

#	Succès	Notes
1	Oui	
2	Oui	
3	Oui	
4	Oui	
5	Oui	
6	Oui	
7	Oui	Boîte accrochée. Déviation minime.
8	Oui	
9	Oui	
10	Oui	
11	Non	Boîte accrochée. Déviation de plus de 20 degrés.
12	Oui	
13	Oui	
14	Oui	
15	Oui	
16	Oui	
17	Oui	Boîte accrochée. Déviation minime.
18	Oui	
19	Oui	
20	Oui	

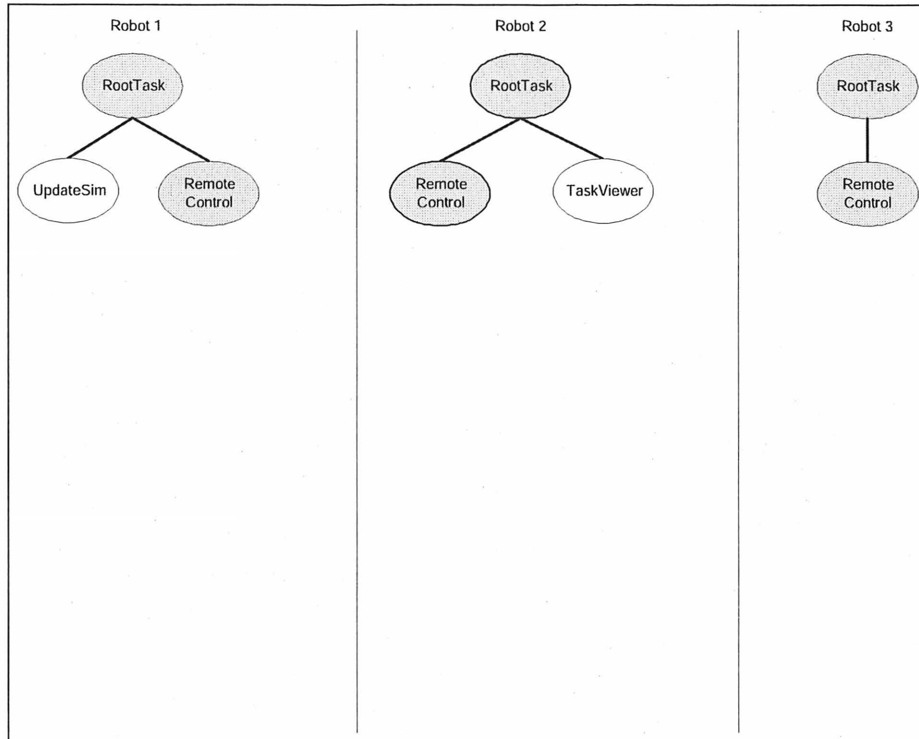

Figure II.1 – Arbre de tâches à l'étape 1

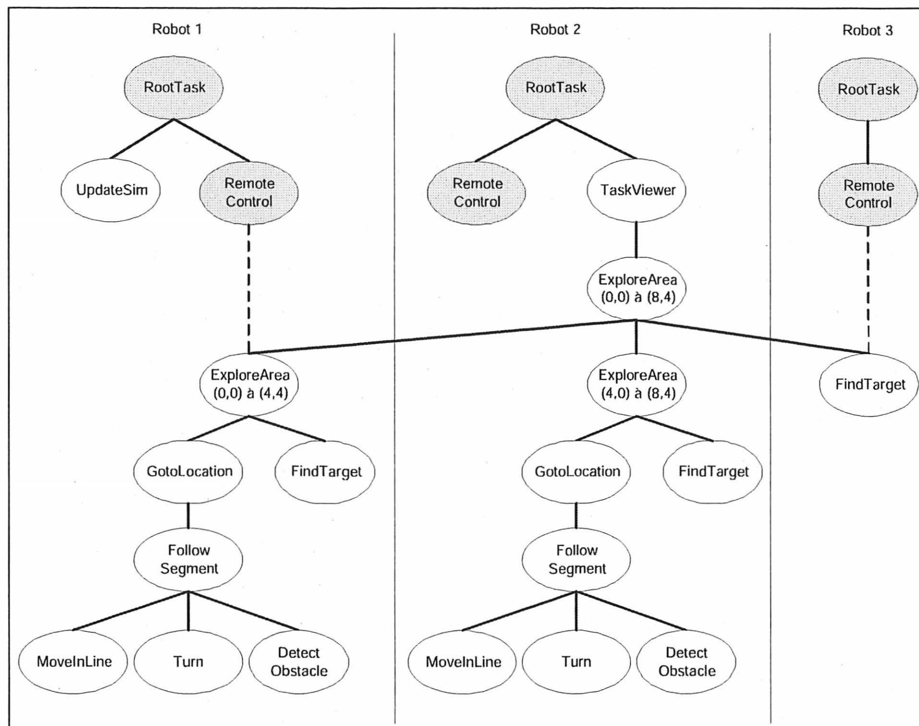

Figure II.2 – Arbre de tâches à l'étape 2

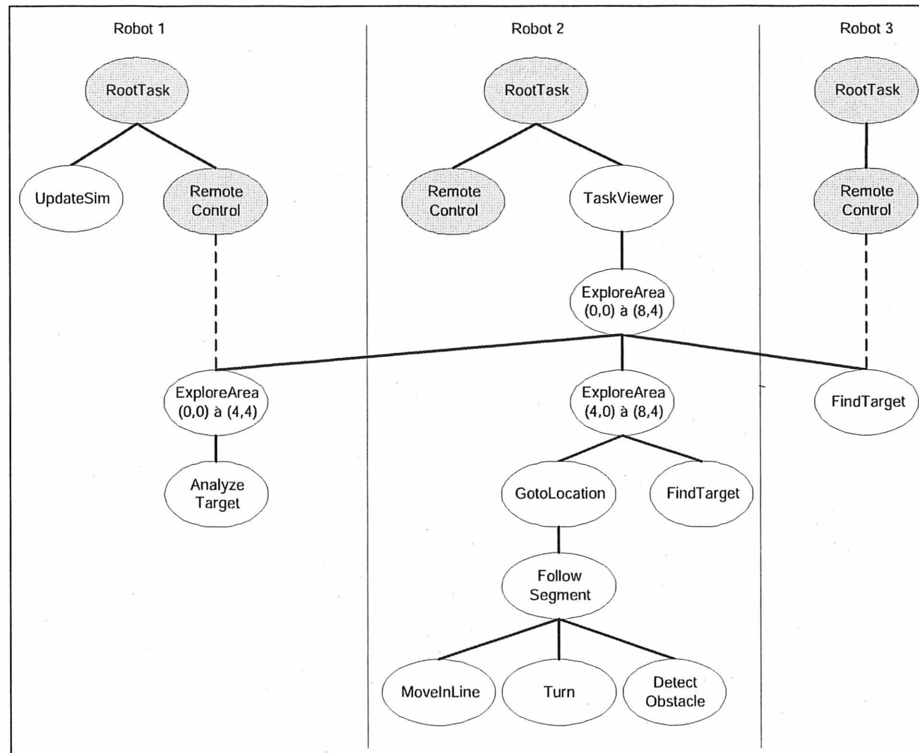

Figure II.3 – Arbre de tâches à l'étape 3

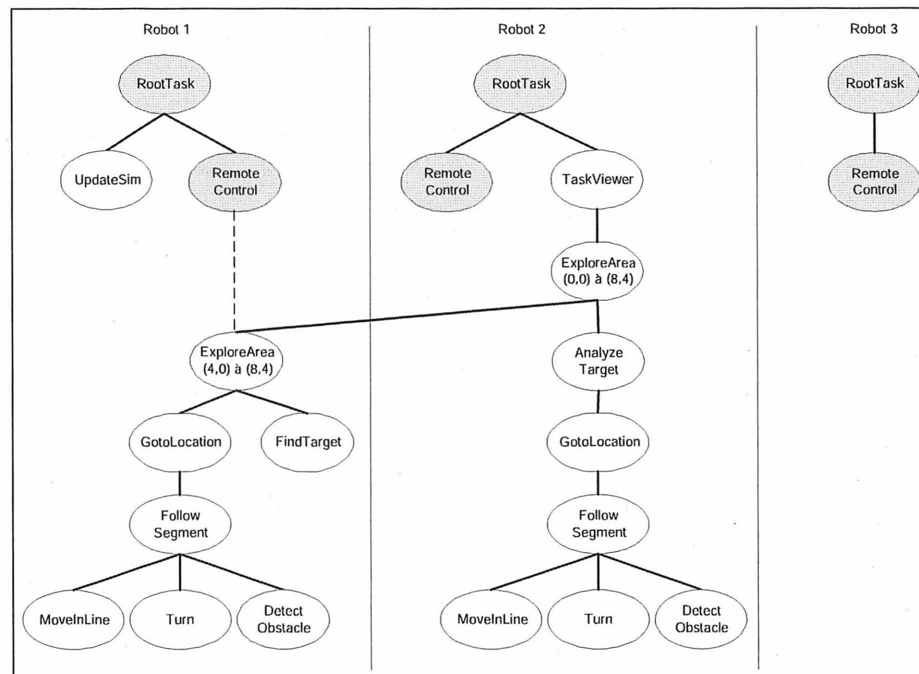

Figure II.4 – Arbre de tâches à l'étape 4

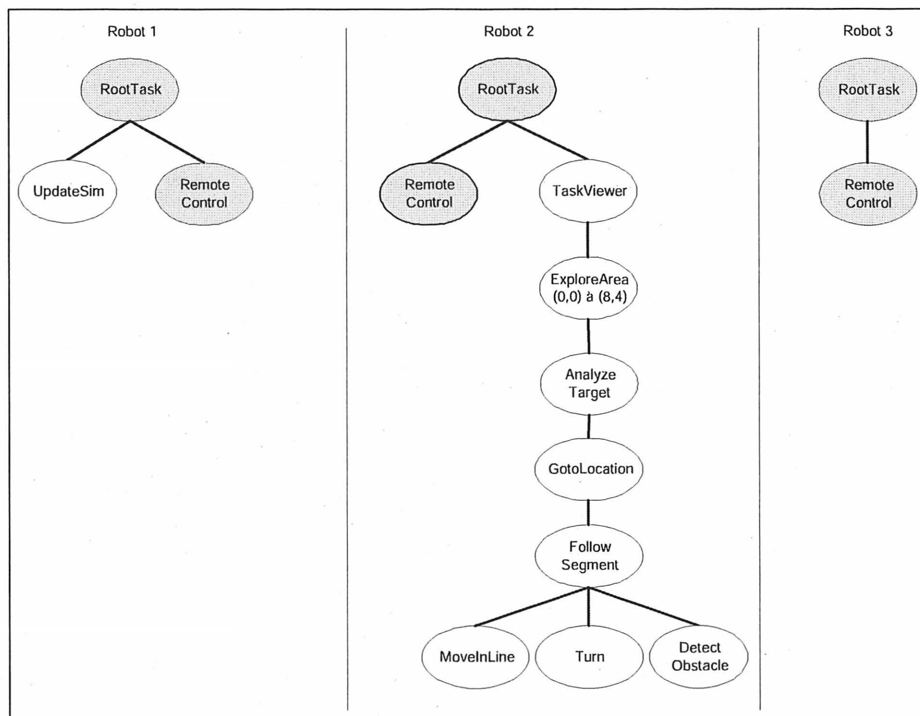

Figure II.5 – Arbre de tâches à l'étape 5

ANNEXE III - RÉSULTATS DES TESTS DE TEMPS DE RÉACTION

Temps de réaction pour le démarrage d'une tâche locale

#	Temps de réaction (ms)
1	12,0
2	8,4
3	9,6
4	10,8
5	11,4
6	9,6
7	10,8
8	8,4
9	14,4
10	9,6
11	14,4
12	9,6
13	9,6
14	10,8
15	12,0
16	10,8
17	11,4
18	9,6
19	13,8
20	9,6
	10,6

Temps de réaction pour le démarrage d'une tâche distante

#	Temps de réaction (ms)	Délai du réseau (ms)
1	112	30
2	98	24
3	133	34
4	112	27
5	116	26
6	115	27
7	114	28
8	95	28
9	80	25
10	72	24
11	90	27
12	61	22
13	81	28
14	77	26
15	97	26
16	83	27
17	87	26
18	101	30
19	113	35
20	63	21
	95	27

BIBLIOGRAPHIE

ALAMI, R., CHATILA, R., FLEURY, S., GHALLAB, M., INGRAND, F. (1998). An architecture for autonomy. *International Journal of Robotics Research*, volume 17, numéro 4, pages 315–337.

ASAMA, H., MATSUMOTO, A., ISHIDA, Y. (1989). Design of an autonomous and distributed robot system : Actress. Dans *Proceedings of IEEE/RSJ International Workshop on Intelligent Robot and Systems*, Tsukuba, Japan, pages 283–290.

BENI, G. (1988). The concept of cellular robot. Dans *Proceedings of the Third IEEE Symposium on Intelligent Control*, Arlington, Virginia, USA, pages 57–61.

CAO, Y. U., FUKUNAGA, A. S., KAHNG, A. B. (March 1997). Cooperative mobile robotics : Antecedents and directions. *Autonomous Robots*, volume 4, numéro 1, pages 7–23.

CHAIMOWICZ, L., CAMPOS, M., KUMAR, V. (2002). Dynamic role assignment for cooperative robots. Dans *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Washington, DC, pages 293–298.

DORF, R. (1990). *Concise International Encyclopedia of Robotics : Applications and Automation*. Wiley-Interscience, 1190 pages.

DUPUIS, E., L'ARCHEVÊQUE, R. (2003). Autonomous robotics and ground operations. Dans *7th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Nara, Japan.

FUKUDA, T., KAGA, T. (1997). Distributed decision making of dynamically reconfigurable robotic system. Dans *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems*, volume 3, pages 1604–1609.

KHOO, A., HORSWILL, I. D. (2002). An efficient coordination architecture for autonomous robot teams. Dans *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Washington DC, USA, pages 287–292.

KITANO, H., ASADA, M., KUNIYOSHI, Y., NODA, I., OSAWA, E. (1997). RoboCup : The robot world cup initiative. Dans *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, New York, pages 340–347. ACM Press.

KUBE, C. R., ZHANG, H. (1993). Collective robotics : From social insects to robots. *Adaptive Behaviour, 1993*, volume 2, numéro 2, pages 189–218.

LAENGLE, T., LUETH, T., REMBOLD, U., WOERN, H. (1998). A distributed control architecture for autonomous mobile robots - implementation of the karlsruhe multi-agent robot architecture (KAMARA). *Advanced Robotics*, volume 12, numéro 4, pages 411–431.

MACKENZIE, D., ARKIN, R., CAMERON, J. (1997). Multiagent mission specification and execution. *Autonomous Robots*, volume 4, numéro 1, pages 29–52.

MATARIĆ, M. (1998). Coordination and learning in multirobot systems. *IEEE Intelligent Systems*, volume March/April, pages 6–8.

MATARIĆ, M. J. (1994). Interaction and intelligent behavior. Rapport technique AITR-1495, Massachusets Institute of Technology.

MATARIĆ, M. J., GERKEY, B. P. (2002). Sold! : Auction methods for multirobot coordination. *IEEE Transactions on Robotics and Automation*, volume 18, numéro 5, pages 758–768.

NAKAMURA, A., ARAI, T. (2002). Human-supervised multiple mobile robot system. *IEEE Transactions on Robotics and Automation*, volume 18, numéro 5, pages 728–743.

PAGELLO, E., D'ANGELO, A., MONTESELLO, F., GARELLI, F., FERRARI, C. (1999). Cooperative behaviors in multi-robot systems through implicit communication. *Robotics and Autonomous Systems*, volume 29, numéro 1, pages 65–77.

PARKER, L. (1998). Alliance : An architecture for fault-tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, volume 14, numéro 2, pages 220–240.

PARKER, L. (2000). Current state of the art in distributed autonomous mobile robotics. *Distributed Autonomous Robotic Systems*, volume 4, pages 3–12.

PIRJANIAN, P., HUNTSBERGER, T., TREBI-OLLENNU, A., AGHAZARIAN, H., DAS, H., JOSHI, S., SCHENKER, P. (2000). Campout : A control architecture for multirobot planetary outposts. Dans *Proceedings SPIE Conference Sensor Fusion and Decentralized Control in Robotic Systems III*, Boston MA, USA.

SAITO, J., YAMADA, S. (1999). Adaptive action selection without explicit communication for multi-robot box-pushing. Dans *Proceedings of IEEE/RSJ International Conference on Intelligent Robot and Systems*, Kyonjyu, Korea.

SELLEM, P., AMRAM, E., LUZEAUX, D. (2000). Open multi-agent architecture extended to distributed autonomous robotic systems. Dans *SPIE Aerosense'00, Conference on Unmanned Ground Vehicle Technology II*, Orlando, FL, USA.

SIMMONS, R., APFELBAUM, D. (1998). A task description language for robot control. Dans *Proceedings of the IEEE Conference on Intelligent Robotics and Systems*, Vancouver, Canada.

SIMMONS, R., APFELBAUM, D., FOX, D., GOLDMAN, R., HAIGH, K., MUSLINER, D., PELICAN, M., THRUN, S. (2000). Coordinated deployment of multiple, heterogeneous robots. Dans *Proceedings International Conference on Intelligent Robots and Systems*, Takamatsu, Japan.

SIMMONS, R. G., APFELBAUM, D., BURGARD, W., FOX, D., MOORS, M., THRUN, S., YOUNES, H. (2000). Coordination for multi-robot exploration and mapping. Dans *Proceedings AAAI/IAAI*, pages 852–858.

VAUGHAN, R., GERKEY, B., HOWARD, A. (2003). On device abstractions for portable, reusable robot code. Dans *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, pages 2421–2427.

VOLPE, R., NESNAS, I., ESTLIN, T., MUTZ, D., PETRAS, R., DAS, H. (2001). The CLARAty architecture for robotic autonomy. Dans *Proceedings of the 2001 IEEE Aerospace Conference*, Big Sky, Montana, USA.

WERGER, B. B. (1999). Cooperation without deliberation : A minimal behavior-based approach to multi-robot teams. *Artificial Intelligence*, volume 110, numéro 2, pages 293–320.

ZLOT, R., STENTZ, A., DIAS, M., THAYER, S. (2002). Multi-robot exploration controlled by a market economy. Dans *Proceedings of the IEEE International Conference on Robotics and Automation*.