A Thesis submitted to the Department of Informatics in partial fulfillment of the requirements for the degree of Master of Science

A Multi-Robot Platform for Mobile Robots with Multi-Agent Technology

Technical Aspects of Multimodal Systems
Distributed Systems and Information Systems
University of Hamburg

submitted by
Sebastian Rockel
July 2011

supervised by
Prof. Dr. Jianwei Zhang
Dr. Lars Braubach

One, a robot may not injure a human being, or through inaction, allow a human being to come to harm;
Two, a robot must obey the orders given it by human beings except where such orders would conflict with the First Law;
Three, a robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.

Isaac Asimov, Laws of Robotics from I. Robot, 1950

Abstract

In the field of robotics, typical, single-robot systems encounter limits when executing complex tasks. Todays systems often lack flexibility and inter-operability, especially when interaction between participants is necessary. Nevertheless, well developed systems for the robotics domain and for the cognitive and distributive domain are available. What is missing is the common link between these two domains.

This work deals with the foundations and methods of a middle layer that joins a multi-agent system with a multi-robot system in a generic way. A prototype system consisting of a multi-agent system, a multi-robot system and a middle layer will be presented and evaluated. Its purpose is to combine high-level cognitive models and information distribution with robot-focused abilities, such as navigation and reactive behavior based artificial intelligence. This enables the assignment of various scenarios to a team of mobile robots.

Zusammenfassung

In der Robotik stoßen heute konventionelle Systeme, auf der Basis einzelner oder mehrerer Roboter, bei komplizierten Aufgaben schnell an ihre Grenzen. Speziell wenn Interaktionen zwischen den Teilnehmern notwendig sind, haben heutige Systeme Nachteile hinsichtlich Flexibilität und Interoperabilität. Ungeachtet dessen existieren bereits weit entwickelte Systeme im Bereich der Robotik sowie der Kognition und der verteilten Systeme. Was fehlt ist eine generische Verbindung dieser Bereiche.

Die vorliegende Arbeit beschäftiget sich mit den Grundlagen und Methoden eines Middle-Layers, der genau das tut. Des Weiteren soll anhand einer prototyphaften Integration in ein Gesamtsystem, bestehend aus einem Multi-Agentensystem, einem Multi-Robotersystem und des Middle-Layers, die Anwendbarkeit demonstriert und das Ergebnis evaluiert werden. Der Zweck dieses Systems ist die Verbindung von abstrakten, kognitiven Modellen und verteilten Systemen mit Roboterfähigkeiten, wie Navigation und Verhalten. Das ermöglicht, im Rahmen der Arbeit, den Einsatz verschiedener Szenarios mit einem Team mobiler Roboter.

Table of Contents

List of Figures

Nomenclature

ACL	Agent Communication Language
AI	Artificial Intelligence
AMCL	Adaptive Monte Carlo Localization algorithm
API	Application Programmer's Interface
BBAI	Behavior Based Artificial Intelligence
BDI	Belief-Desire-Intention model
Blob	A colored object that can be detected by a blobfinder device
Blobfinder	A model for a visual blob-detection device, such as a blob-detecting camera, that is capable of detecting specific colors in images
Device	A sensor or effector of a robot, or the robot itself
DNS	Domain Name Service
FIPA	The Foundation for Intelligent Physical Agents
GPS	Global Positioning System
Gripper	A robot device that is capable of grasping and carrying an object
HAL	Hardware Abstraction Layer
ICP	Iterative Closest Point algorithm
IP	Internet Protocol
JADE	Java Agent DEvelopment framework
Jadex	JADE Extension multi-agent system
JCC	Jadex Control Center
JNA	Java Native Access
JNI	Java Native Interface
JVM	Java Virtual Machine
MAS	Multi-Agent System
MDA	Model Driven Architecture
Meta-Platform	A combination of an MAS and MRS connected using a multi-robot middle layer
MRS	Multi-Robot System
ND	Nearness Diagram path planning algorithm
ROS	Robot Operating System
RSAL	Robot System Abstraction Layer
SLAM	Simultaneous Localization And Mapping
SND	Smoothed Nearness Diagram path planning algorithm
SOAP	Simple Object Access Protocol

VFH	Vector Field Histogram path planning algorithm
XML	Extensible Markup Language

Introduction 1

This work was written at the Group of Technical Aspects of Multimodal Systems (TAMS) at Hamburg University, where the research and education focus is on multimodal sensing and representation, knowledge-based robot control and learning as well as mobile service-robots. The work was done in close cooperation with the Group of Distributed Systems and Information Systems (VSIS), which focusses on distributed systems, service-oriented software architectures and content management.

1.1 Motivation

In modern robotics, simple tasks still demand complex solutions. An important development has been the use of multi-robot systems (MRS) to provide high-level access to robot hardware. Although such platforms provide transparent access to sensors and actuators, difficulties still remain. Typical tasks for a mobile robot require at least some sensors such as a stereo-vision camera, a robot arm or even a hand. To interconnect these sensors in software in a meaningful way is not trivial, although device access itself might be simple. The execution of even simple commands such as “grip that trash”, “open that door” or “find the pink ball” can be assumed to be complex tasks. They can be solved either by one or more sophisticated and specially adapted robots, or by multiple, relatively simple, robots that coordinate their activities. There are important advantages in the latter approach, such as task flexibility, scalability, cost and platform autonomy, which will be expanded upon as requirements to this project in the next section.

Extensive research and implementation effort has also gone into the area of multi-agent systems (MAS) in order to assist the development of distributed systems and of cognitive autonomous agents with learning intelligence.

1.2 Goals

This work describes a platform combining an MRS and a distributed and intelligent MAS. A stable robot platform will serve as the basis for high-level services. The platform should provide a framework for modeling the objects needed for various tasks. In addition, the following requirements should be fulfilled.

In order to provide *task flexibility* the platform should be able to execute tasks of arbitrary complexity by managing a dynamic and heterogeneous group of robots. Flexibility in the number of robots taking part in a task is termed *scalability.* The platform should perform with an arbitrary number of robots, selected according to robot availability and task requirements. Preparatory work should be done to facilitate future dynamic addition of extra robots and the replacement of malfunctioning robots. In addition, regarding the *cost*, this work assumes that a group of small, heterogeneous robots is cheaper to maintain in the mid- to long-term than a few, highly specialized ones. Last but not least, the platform should be ready for future enhancements in the scientific research fields involved. This means that preparations should be made with regard to the possible exchange of the MRS and MAS with other potentially better suited systems. This is called *platform autonomy.*

The main goal of this work is to develop a layer of software between the MAS and MRS and to integrate a prototype platform that involves these three layers. The aim is to have real robots cooperating in the TAMS indoor environment.

1.3 Outline

This document is structured as follows.

The chapter 1, *Introduction,* establishes the topic of the work. Multi-robot systems are mentioned as advantageous systems in modern robotics. Their use serves as the motivation for the advanced system presented in this work. The fundamental goals of the approach are described.

In chapter 2, *State of the Art,* the basics of multi-robot and multi-agent systems are described and widely-used systems are discussed. A choice of assorted tools is made and presented. Preliminary work in the scientific field of mobile robotics is mentioned and an overview of related work in the field of cooperative robotics is given. A discussion of the advantages of a combined multi-agent and multi-robot platform concludes the chapter.

An approach to the development of such a combination is presented in chapter 3, *Approach.* The constraints on the approach are elaborated, the robot hardware is introduced and the requirements are developed. The basic architecture of a new meta-platform is presented, as is the detailed design of the middle layer between multi-agent and multi-robot systems.

The use of the meta-platform presented here is described in chapter 4, *Scenarios.* Selected scenarios representing use-cases with multiple cooperating robots are implemented and the practical use of the platform is demonstrated. Finally a preview of possible future enhancements is given and instructions for migration to other multi-agent and multi-robot systems complete the chapter.

In chapter 5, *Conclusion,* the technology, scenarios, user interface and performance of the system are evaluated. The development of multi-robot scenarios is discussed, as are areas of scientific research that could benefit from the system presented. Finally the work is summarized and several possible enhancements are discussed.

State of the Art

2

2.1 Introduction

This chapter summarizes current robot and agent-systems technologies. Dedicated open-source software for these areas will be introduced and preliminary work and related research will be mentioned. Finally the advantages of a combined robot and agent system will be discussed.

2.2 Multi-Robot System

A multi-robot system is described in [GVH03] as a software system providing tools that simplify controller development, particularly for multiple-robot, distributed-robot, and sensor network systems. Typically, and especially in this work, various heterogeneous mobile robots are controlled by a dedicated MRS.

An MRS provides hardware abstraction and driver encapsulation, where a driver is a control algorithm that supports the underlying hardware. For example, a driver might control differential drives, ranging sensors or localization.

The two MRS introduced in section 2.2.1 and 2.2.2 also support a level of network abstraction. These MRS were chosen because they are well established and maintained, are widely used and have a large user community. Any robot, sensor or effector (for example a motor) hereafter referred to as *Devices*, can be accessed from anywhere within the network regardless of the client's[1] point of access.

An MRS typically provides tools, drivers and software frameworks for accessing a range of different robots. Generally, concurrent robot activity and inter-robot communication are supported. Nevertheless there is currently no framework for intelligent, networked Device behavior.

The two MRS mentioned provide a basic set of functionality for a lone robot: perception, manipulation and representation of the environment in the form of a map. In particular any client based on these MRS can rely upon motion control, object

[1]Client here means some user program

manipulation, such as controlling a robot arm, perception of the environment via sensors and motion planning towards a specified target. Motion planning includes the mapping of sensory data to map coordinates (localization) and navigation. The following systems are open-source tools that are widely used in robotics research.

2.2.1 Player/Stage

In this work, the Player/Stage MRS is used. Player/Stage was developed in 1999 at the USC Robotics Research Lab to address interfacing and simulation for MRS. Since then it has been modified and extended by researchers and now has an active support community. It is free and open-source.

Player/Stage provides C and C++ API-libraries with additional bindings for script languages, such as Ruby and Python. Third-party open-source projects provide further client support, currently for Ada, Octave and Java (section 2.2.1). Moreover, Player/Stage comes with out-of-the-box tools that make testing and debugging easier. Such tools include graphical interfaces for navigation[2] and control as well as for visualizing sensory data. A well integrated robot simulator is also included. Figure 2.1 depicts the *Stage* simulation interface.

Stage, originally developed to simulate a two-dimensional world, has evolved into a so called 2.5D simulator because many (although not all) model aspects are based upon three dimensions. A pure three-dimensional simulator, called *Gazebo*, is available too. This simulator also integrates a physics engine and can be quite demanding in terms of computing time. Gazebo[3] is best suited to small robot populations with high model fidelity in an outdoor environment, whereas Stage is optimized for a large robot population with lower accuracy and indoor scenarios.

Javaclient[4] is third-party software for the Player/Stage project and provides a Java-API suitable for any Java client. It uses the socket interface from Player/Stage and provides a client API in native Java that is functionally similar to the native C/C++ interface of Player/Stage. The socket interface provides platform-independent control of Player/Stage because socket interfaces are implemented in many programming languages and operating systems. Player/Stage has a client/server infrastructure. Figure 2.2 depicts Player/Stage socket communication and the client/server architecture.

Player/Stage is released under the GNU General Public License[5], which allows all code to be used, distributed and modified freely, on the condition that any derived work be distributed under the same license terms.

[2]http://playerstage.sourceforge.net/wiki/Robot_Navigation (July 26, 2011)
[3]http://playerstage.sourceforge.net/gazebo/gazebo.html (July 26, 2011)
[4]http://java-player.sourceforge.net
[5]http://www.gnu.org/licenses/gpl.html (July 26, 2011)

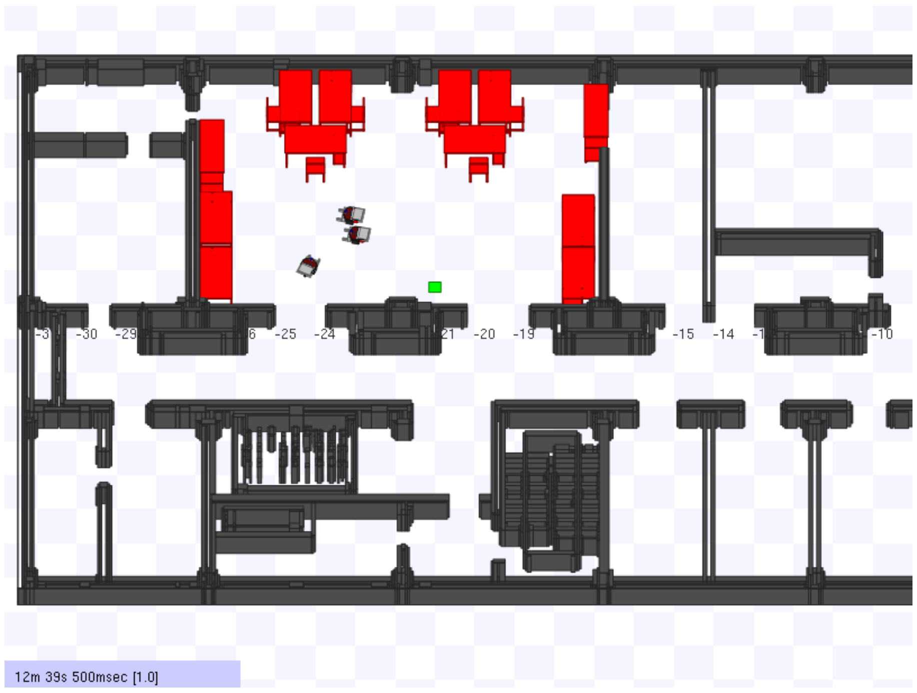


Fig. 2.1: The Stage (graphical) simulation interface shows in oblique projection a kind of three dimensional robot environment. The environment itself consists of a floor, a map and various objects. Here the map is represented by the dark walls. The red objects represent furniture such as tables, chairs and sideboards. The map actually shows a part of the TAMS floor and its laboratory. Three simulated robots reside inside the laboratory. The square green spot on the laboratory floor represents a blob, which can be detected by any blobfinder device, such as blob-detecting cameras.

2.2.2 ROS

ROS[6] stands for *Robot Operating System*. It provides features similar to the Player/Stage project and uses algorithms from that project and others [QCG$^+$09]. Its focus is more on dynamic, peer-to-peer communication between robot devices and on modular design. ROS supports client code in C++, Python, Octave and LISP. Other language support, such as a Java-API via the Java Native Interface (JNI)[7], is either planned or currently in an early phase of support[8].

ROS is released under the Creative Commons Attribution 3.0 license[9], which allows free use, distribution and modification of all code under the condition that derived works attribute to the work in the manner specified by the original author.

[6]http://www.ros.org
[7]http://en.wikipedia.org/wiki/Java_Native_Interface (July 26, 2011)
[8]http://www.ros.org/wiki/rosjava (July 26, 2011)
[9]http://creativecommons.org/licenses/by/3.0 (July 26, 2011)

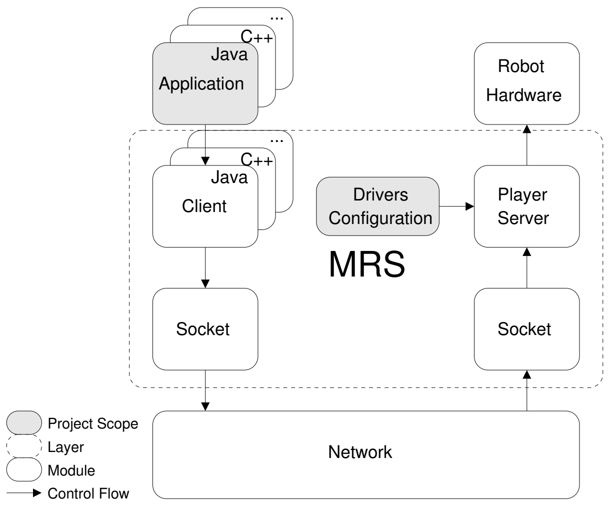


Fig. 2.2: Here a native Java application calls the MRS. Commands are sent through the socket interface via a network (LAN or WLAN) to the Player server, which could be running either locally or remotely. Finally the commands are sent to the locally connected robot hardware, e.g. the robot wheel motors. Arrows show the call direction. Instead of using Java, the application could be coded in C++ (or another MRS-supported programming language) and would use the appropriate bindings in the MRS client libraries.

2.2.3 Discussion

Player/Stage provides driver support for the hardware available at the TAMS laboratory. The reliable Pioneer series mobile robot hardware, from the MobileRobots Inc., is especially well supported. New or unsupported hardware can be integrated through Player/Stage's modular plugin-driver architecture and experience and code can be re-used from earlier research (section 2.4). Probably the most important current advantage of this MRS is its well supported and functional Java-API. This API can easily be integrated with multi-agent systems (section 2.3), such as the Java Agent Development Framework[10] (JADE) or Jadex[11].

Other programming languages (such as C/C++ or Python, which are used by the ROS project) can be integrated with Java code via JNI or Java Native Access[12] (JNA). Although this augments the variety of usable programming languages, it still has the disadvantage of being a non-native interface from the client language perspective. This could introduce constraints on programming flexibility and maintainability and could potentially introduce errors.

Both Player/Stage and ROS support distributed Devices interacting within a network, serve as a kind of robot operating system and provide a Hardware Abstraction Layer (HAL).

[10]http://jade.tilab.com
[11]http://sourceforge.net/projects/jadex
[12]http://en.wikipedia.org/wiki/Java_Native_Access (July 26, 2011)

In this work the MRS Player/Stage has been chosen as a result of experience gained with it during preliminary work and because of its well supported Java-API.

2.3 Multi-Agent System

A multi-agent system is a software system providing a platform to implement, integrate and run agents.

> An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors. A human agent has eyes, ears, and other organs for sensors, and hands, legs, mouth, and other body parts for effectors. A robotic agent substitutes cameras and infrared range finders for the sensors and various motors for the effectors. A software agent has encoded bit strings as its percepts and actions. [RN03]

Of note is the frequent definition of a robot as a physical agent. Thus in related work a physical robot is often called an agent when it acts autonomously to a certain degree. That does not necessarily mean that in such scenarios a dedicated MAS has been integrated.

When it comes to software design, the question of why we use agents and not objects arises (in the sense of object-oriented programming). Agents can be viewed as an evolution of programming methodology. Many properties of agents are also associated with objects such as modularity and code reusability or message passing for information exchange. Nevertheless agents have some more advanced abilities. For example they are typically autonomous. Consequently, the mapping from perceptions to actions relies not only on pre-defined knowledge but also upon experience. What is more, agents are normally *dynamic*: they do not just wait for messages or method invocations to perform an action but can invoke methods by themselves. In other words, they have their own thread of execution.

Agents are encapsulated within a software system, here called a platform, that starts, runs, and stops them and provides network services for distributing work across multiple computers. A computer, here, can be plain, without special equipment, or dedicated, having particular attached hardware, such as the embedded computers integrated within robots. Moreover the development of agents provides an *adaptive system* that handles high software *complexity*, *scalability* and *modularity*. Service- and goal-oriented modeling and flexible deployment are some of its advantages. Model Driven Architectures *(MDA)* and *declarative programming* are commonly used for MAS applications.

So an MAS consists of a platform and the agent code that together provide out-of-the-box capabilities for distributed systems. With the ideal agent-system the developer would just pick the agents needed for a task and the agents would organize themselves to achieve the goal. Another ideal approach would be an agent-system that could be given a task and the necessary agents would be selected automatically.

There are lot of MAS platforms available today[13]. They can be classified broadly as either middleware- or reasoning-oriented systems [BPL04]. Beside the functional focus, other aspects are also important in choosing the right software for a project. License conditions might enable the use or modification of the software without fees. Standards compliance, especially for communication protocols or interoperability, might be important depending upon the project goals. That the software has a large supporting community or has been integrated in third-party projects gives an indication of its maturity. Last but not least, project oriented tools already included might change the selection in favor of other software.

The MAS used in this work combines middleware and a reasoning-oriented layer. It is open-source and has a large user community. The preference for this software arises from its standards compliance and its design as an operating system independent application. In addition, basic tools for agent-debugging are included and the available local knowledge of the software provides for fast support.

2.3.1 Jadex

Jadex (JADE Extension) is a MAS that was originally an extension to the widely used JADE [BPL05]. Although it can still interoperate with JADE [PBL03] it has been developed into its own MAS. It provides middleware and a reasoning engine [PBL05]. The Jadex middleware handles interoperability, security and maintainability as standardized by the Foundation for Intelligent Physical Agents[14] (FIPA), as well as transparent network communication via the Simple Object Access Protocol (SOAP) for agent implementations. The reasoning layer deals with rationality and goal-orientation, exploiting cognitive architectures based on theories of modeling individual behavior. One important theory used is the Belief-Desire-Intention model (BDI, section 4.2.1 on page 49).

The latest Jadex architecture benefits from ubiquitous computing and multicore hardware. It is under active development and implements state-of-the-art programming methodologies in the fields of distribution and concurrency [PB09, PBJ10]. It is implemented in Java and uses XML definitions and plans for agents [PBWL07]. Jadex was developed in 2003 at the Distributed Systems and Information Systems Group at the University of Hamburg.

Jadex is released under the GNU Lesser General Public License[15], which covers the same permissions as the GPL but additionally allows proprietary extensions to the original (unchanged) software or use of the software within closed-source programs.

[13]http://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software (July 26, 2011)
[14]http://www.fipa.org
[15]http://www.gnu.org/copyleft/lesser.html (July 26, 2011)

2.4 Preliminary Work

In an earlier project [RG10] we used Player/Stage to implement a ball-tracking mobile robot. A Pioneer-2DX robot with on-board sonar, laser and omni-directional camera served as the hardware (the mobile robot hardware platform can be seen in figure 3.2 on page 17). With a wall-following exploration algorithm and the sonar, laser and camera data, its goal was to look for a small pink ball in its environment. Once detected the robot headed towards the ball while continuously tracking it. The project provided practical experience with a robot platform and with mobile robot hardware that will be relied upon by this work.

2.5 Related Work

Multiple robots collectively committed to a task play a more important role in today's robot research than in the past. In earlier research one, probably quite sophisticated, robot was adapted to solve a task very efficiently. There has been some scientific research into the collaborative effort of multiple robots assigned to a task.

Robot teams have been applied in various research fields, with the navigation of multiple robots in a synchronized manner being one of the most active. In [AKBF10] a team of two hexapod robots collaborate in navigating unknown territory. [SN10] proposes another architecture for navigation and tracking using multiple robots in unknown environments. [Cer08] investigates a kind of negotiation protocol between multiple robots in determining the shortest distance to a target position. The latter two papers use the JADE MAS for their agents. In [UM09] a team of robots cooperates to navigate and to organize dynamic formations.

The latter work overlaps another quite active field in robotics, namely research into *formations* of multiple robots. [MBF10] researches a grid-based formation and the synchronization and configuration of agents. Another study [MLW09] discusses fault-tolerant formations of mobile robots, while in [STSI09] the focus is on stable and spontaneous self-assembly of an MRS.

Another frequently investigated topic is that of a team of robots *building a map*. [KKF$^+$10] shows a ground-mobile robot and a quadrocopter cooperating to build a three dimensional map. Traditional two dimensional SLAM with multiple robots in unknown territory, using Pioneer-3DX mobile robots and barcode markers, is described in [CND$^+$10]. Theoretical algorithms for cooperative, multi-robot, area exploration are described in [YHTS10]. In contrast to conventional two or three dimensional grid maps, [CDGU10] solves the problem of an MRS collaboratively creating a topological map. Another study, focusing on three dimensional, laser-based modeling with a heterogeneous team of mobile robots combining ICP, SLAM and GPS in an outdoor scenario can be found in [KNT$^+$09].

Other areas of research concentrate on the problems of *task allocation and sub-division,* which arise when a task is to be split into sub-tasks for each robot. [RAB09] uses a box pushing *scenario* in a heterogeneous MRS, while a framework for multi-robot coordination and task allocation is proposed in [SC09]. Other research suggests another scenario: a number of mobile robots are combined to find, collect and drop trash into dustbins to clean the floor [MAC97, BBC$^+$95].

An important aspect of concurrent and autonomous software systems (with agents and robots) is the design of software such that it remains efficient, easy to maintain and re-usable. [BRA94] investigates how efficient communication in a multi-agent robotic system can increase overall performance, whereas [TvS10] focuses on modular and re-usable software in autonomous robot systems. A system with intelligent power management for teams of robots is introduced in [KDP09].

A traditional Artificial Intelligence (AI) topic, reinforcement learning in cooperative MRS, is described in [SMKR09]. Finally, an increasing area of motivation and effort, the field of mixed reality simulation, was studied for mobile robots in [CMW09].

2.6 Advantages of a Combined MAS and MRS Platform

In preceding chapters, a “platform” was introduced as an operating system or a system providing the framework for specific features. A combination of platforms leads to a “platform of platforms”. To avoid confusion such a platform should be called a meta- or combined-platform. Beginning here, the term meta-platform means the combination of a multi-agent system (MAS, such as Jadex or JADE) communicating through a middle layer with a multi-robot system (MRS, such as Player/Stage or ROS). The middleware layer, the agents and their communication services are new and are implemented as part of this work.

Having more than one robot adds *task-flexibility* to a platform. Tasks can be solved more quickly and, in a heterogeneous, multi-robot environment, more effectively. A platform can use an arbitrary number of robots, selected according to robot availability and task requirements. Selecting the robot best suited to a given (sub-) task uses available resources in an efficient manner. Within the platform, malfunctioning robots can be detected and replaced. Older robots that are otherwise no longer useful can still contribute as a small part of the whole platform.

As agents must coordinate their actions, a communication network is required. This network can, as a bonus, be used for additional input and output, such as debugging and testing and for graphical user interfaces (GUIs).

2.7 Summary

In this chapter we have defined multi-robot and multi-agent systems and have described the features they typically provide. We have discussed commonly used, open-source software projects providing these features, summarized preliminary and related work and introduced recent, related, research into this topic. Finally the advantages of a combined MAS and MRS platform have been presented.

Approach

3

3.1 Introduction

This chapter describes the core work of this thesis. The platform developed will be introduced and described in detail. Explanations of the concept, the architecture, the detailed design and the implementation will shed light on the ideas behind this work.

3.2 Goals

The goal is to connect existing MRS and MAS using a new middle layer, the *Robot System Abstraction Layer* (RSAL), to create a three-layer architecture, as depicted in figure 3.1.

The introduction of the RSAL alleviates some mutual constraints imposed by the MAS and MRS. Robot and device actions take significant time to execute (moving an arm or actuator for example), whereas agent methods must return immediately or are at least designed for operations separable in small chunks of work. The middle layer has to manage the transition between a synchronous interface to the robot hardware and an asynchronous interface provided to the MAS. Therefore it encapsulates service oriented facilities, such as subscribing to notifications of lower level device events. Moreover, while MRS APIs differ, it is desirable that the MAS be independent of the MRS. In summary, an additional middle layer decouples the MAS and MRS.

3.3 Constraints

This work combines a high-level MAS with an MRS. Current technologies in MAS and mobile robotics are used to achieve a high degree of task flexibility. Some specialized algorithms are implemented, such as for robot control, but in general, out-of-the box drivers and interfaces are used.

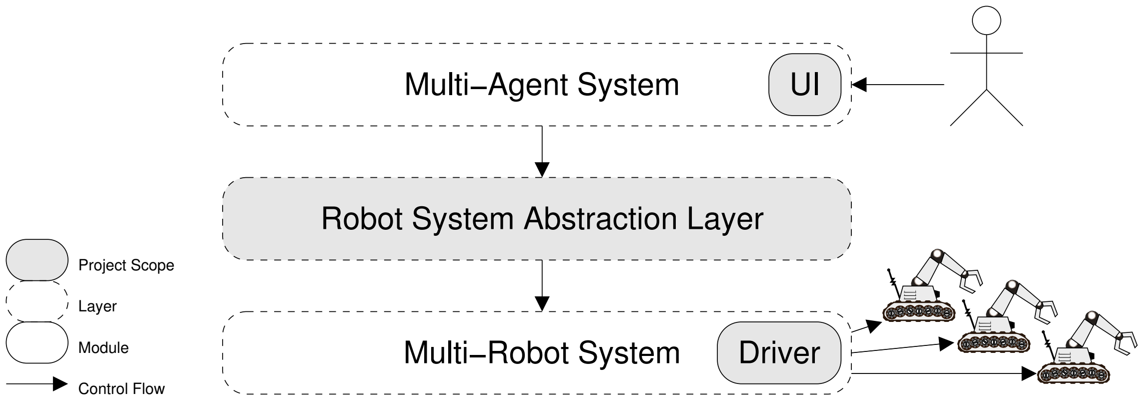


Fig. 3.1: The overall meta-platform architecture shows three layers. An existing MAS encapsulates agents and related services. The user has access to the platform using the MAS user interface through which agents can be started and stopped. Multiple agents contributing to a task are managed through scenario selection. The MRS layer contains the robot hardware related code and drivers. The new middle layer, RSAL, negotiates between these layers, manages functional hardware related code, provides object-oriented access to robots, their devices and behaviors and thus decouples the MAS and MRS.

An MAS is assumed to provide certain features:

- that a sample task can be *described* by the MAS tools and that the required *definitions* (such as of agents) can be created conveniently within the framework;
- that complex tasks can be *divided* into sub-tasks and *distributed* to multiple agents;
- and that distributed agents can use the MAS network *communication* layer to exchange data.

The MRS also has to fulfill some minimal requirements. The most important feature is path planning, although there might be use-cases where this is not needed. The literal meaning of this is that the robot is able to move from its current position to a given target position. Path planning therefore involves several sub-tasks, especially the ability to *localize* on a map using sensors, such as laser or sonar rangers. Knowing its position, the robot should be able to plan a valid trajectory through unoccupied space (floors and rooms) and not hit anything, including both static objects (those on the map) and dynamic obstacles (those not on the map, such as people, furniture or other robots). Last but not least, the MRS has to provide the drivers for all hardware, such as the robot, sensors and effectors.

Some other constraints also exist, especially to the practical demonstration: the types and numbers of robot hardware available at the TAMS laboratory are limited, and as the algorithms used for robot control are part of the MRS, they rely upon the current driver implementations.

3.3.1 Robot Hardware

In this work we will use Pioneer robot models 2DX and 3AT with, respectively, 8 and 16 mounted sonar ranger devices. Both are three-wheeled models with differential drive capabilities and each robot has a laser ranger mounted on top. The ranger is required for accurate localization on the map, as although it is possible to localize with sonar rangers alone, this approach does not provide the necessary accuracy. The Pioneer hardware is depicted in figure 3.2 and 3.3. For some tasks a robot-attached gripper is required.

> A Gripper is a device capable of closing around and carrying an object of suitable size and shape. On a mobile robot, a gripper is typically mounted near the floor on the front, or on the end of a robotic limb. Grippers typically have two “fingers” that close around an object. Some grippers can detect whether an object is within the gripper (using, for example, light beams) [quoted from Player/Stage project].

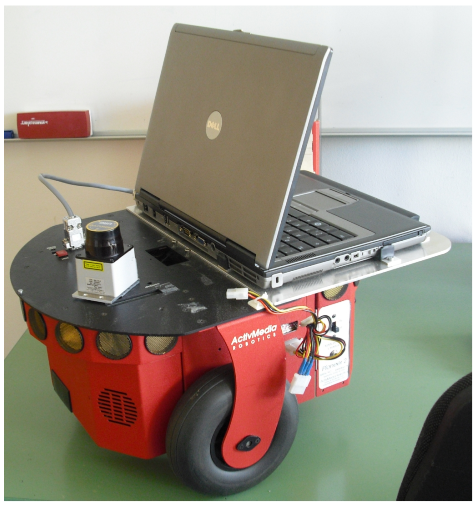


Fig. 3.2: A Pioneer robot used in this work. The mounted laptop runs the control program and communicates wirelessly. The (dismounted) laser can be seen on top of the robot and the fixed sonar sensors are visible as circular devices underneath the top board at the front.

```
driver
(
  name "p2os"
  provides ["odometry:::position2d:0" "sonar:0" "power:0"
      "gripper:::gripper:0" "lift:::actarray:0" "dio:0" "audio:0"]
  port "/dev/ttyS0"
)
```

List. 3.1: This driver configuration for the Player/Stage MRS invokes the built-in *p2os* driver. Its purpose is to provide access to the interfaces of the Pioneer robot hardware. Here for instance, several features of the Pioneer-2DX are provided: the odometer and motor (both via the *position2d* interface), sonars, battery power, gripper (if equipped), the lift to which the optional gripper is attached, digital input and output for optical gripper sensors and an audio interface.

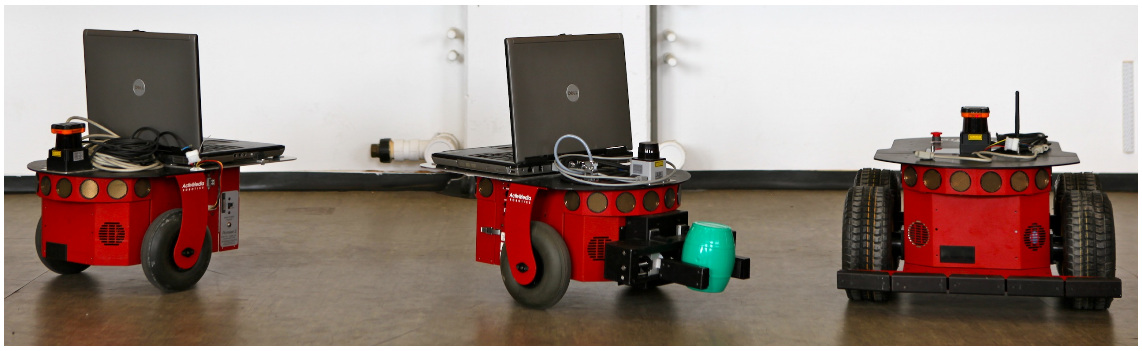


Fig. 3.3: Three robots used for the real-world demonstration: from left to right, a Pioneer-2DX, another 2DX with a front-gripper attached and a Pioneer-3AT with an embedded computer.

3.3.2 MRS Driver

In order to use robot hardware, a device-specific driver must be available to the MRS. The Pioneer robots are supported by the Player/Stage *p2os* driver, which supports the Pioneer robot capabilities, including those of peripheral devices such as an odometer, gripper or sonar ranger. In listing 3.1 the driver configuration used for a Pioneer-2DX is shown.

3.4 Requirements

Requirements can be functional or non-functional. Function requirements indicate essential demands that are specified in detail, such as that the robot shall have a maximum weight of fifteen kilograms. Non-functional requirements specify properties that cannot be given accurately, for example that robot control has to be performance-optimized. In the following subsections, both requirement types will be presented, with distinction between those applying to the middle layer and those for the whole platform.

3.4.1 Functional Requirements

The middle layer RSAL shall be implemented in the Java programming language, as the layer is closely coupled with the Jadex agent layer which itself uses Java. Other widely used MAS also support Java and thus the implementation will be easy to migrate to another MAS. Furthermore a variety of operating systems shall be supported in order to allow task distribution over many hosts. The Java implementation benefits from a Java Virtual Machine (JVM, available for various operating systems[1]), which allows the compiled byte-code to be downloaded and run directly on any host.

The meta-platform shall work on:

- the 64-bit Suse Linux desktop computers available in the TAMS laboratory;
- the 32-bit Suse Linux notebooks mounted on top of or integrated into the robot hardware (Pioneer-2DX and Pioneer-3AT respectively) available in the TAMS laboratory;
- a 64-bit Mac OS X host.

The MRS components Player, Stage and Javaclient shall be used at their most recent versions, 3.1, 4.0, and 3.x respectively.

All Player/Stage driver configurations must be set up. A driver here refers to an available MRS hardware driver or to robot control algorithm support. A driver configuration adapts the available algorithm to a specific setup. It might consist of a robot device configuration (listing 3.1) or of algorithm parameters, such as for localization matching the TAMS floor environment. There are detailed configurations for the following drivers: Pioneer robots (also includes some peripherals, such as sonar and gripper), laser ranger, localization, path planner and map. Their driver configurations are listed in appendix A.

The base platform hardware of the Pioneer robots is similar, with differences only in the peripherals. All robots use the same types of motors and odometers. Although all models have similar sonar ranger sensors their number and orientation differs: one model has only eight front sensors whereas the other also has eight rear sensors. Thus the driver configuration and code shall be able to support both. Moreover the detection and appropriate use of the available sensors shall occur at runtime. The gripper device attached to one robot shall be supported. Furthermore the two laser range finders (Hokuyo UTM-30LX and URG-04LX) shall be detected and configured

[1]http://en.wikipedia.org/wiki/List_of_Java_virtual_machines (July 26, 2011)

at runtime. In order to provide accurate localization, a map of the TAMS floor must be created with a fidelity of at least 0.05 meters per pixel. Robot navigation requires that a set of appropriate Player/Stage drivers be set up for localization and for both local and global navigation. It also requires a map driver that must manage the TAMS floor grid map.

The task of robot navigation is typically divided into local and global path planning, as described in the following paragraphs[2].

> Path planning, sometimes called “Motion Planning”, is the act of finding a path to go from location A to B. [...] There are many approaches to solving path planning, but usually it involves a local and global path planner.
>
> A global path planner usually generates a low-resolution high-level path from A to B, avoiding large obstacles and dealing with navigation around the arena. This is analogous to the path Google Maps might give you from home to a park. Local path planning usually gives a high-resolution low-level path only over a segment from global path A to B, avoiding small obstacles and dealing with motion planning: angles of turn, appropriate velocities, etc. This is analogous to choosing how fast to turn your car when moving around traffic while on your Google Maps path.
>
> Path planning is a major topic in Computer Science, Electrical Engineering, and Mechanical Engineers. Many topics that serve as the basis for path planning include graph theory, geometric algorithms, potential fields, etc. Path planning is known to be an algorithmically intensive problem to solve.

In robotics, the coordinate systems in which positions are defined are called coordinate *frames*. Frames can relate to each other and coordinate transformations enable a position defined in one frame to be expressed as coordinates in a different frame. When referring to a position it is always related to some frame of reference. An absolute and fixed coordinate system is called the global or world frame.

To elaborate on the requirements for the path planning component, there follows a short description of how path planning is typically implemented.

A path planner typically accepts a target position specified in global coordinates. As the robot’s frame of reference typically differ from the global one, the target coordinates must be transformed to the local frame. Planning is focused on a direct trajectory from the current position to the target, which is assumed to be in the line of sight. To avoid dynamic obstacles, an avoidance component is integrated.

In contrast to a local path planner, a global one parses the static map for permanent obstacles, such as walls and divides the planar space into occupied and free grid cells.

[2]quoted from http://psurobotics.org/wiki (July 11, 2011)

Cells that are close to an obstacle are assigned a higher cost than cells in open space. According to this grid and to additional algorithm parameters, such as the minimal allowed distance to an obstacle, the final path will be planned.

A successful trajectory consists of a start, a goal, intermediate goals and intermediate paths. Intermediate goals lie in the line of sight of adjacent ones and are given to the local path planner. Upon completion of an intermediate goal, the local planner is assigned the next one and so on. Local and global path planners must be tuned (by parameter tuning) in order to achieve robust overall performance. For example this affects (intermediate) goal position deviation. The permitted planar deviation (from the real position) must always be more restrictive for the local planner than for the global planner. This avoids a blocking situation where the local planner is satisfied with the robot position whereas the global planner is waiting endlessly for a more accurate position to be reached.

A variety of local path planning algorithms is available and each has its advantages. Their comparison is not in the scope of this work. An overview of planning algorithms can be found in [LaV06, SN04]. Algorithms implemented as Player/Stage drivers are the Vector Field Histogram (VFH), the Nearness Diagram (ND) and the Smoothed Nearness Diagram (SND). The VFH algorithm has been chosen for this work due to its proven practical robustness with the Pioneer robots and its efficient path planning. A global path planning algorithm has been implemented with the Wavefront driver, which is delivered with the Player/Stage software and must be configured to suit the project. The meta-platform shall support all four path planning drivers.

To efficiently localize the robot within the map the Adaptive Monte Carlo Localization (AMCL) algorithm has to be supported. A maximum deviation of 30 centimeters from the real position compared to the localization hypothesis shall be achieved. The Monte Carlo algorithm is based on a probabilistic particle filter. It is capable of using multiple sonar rangers or one laser ranger in combination with odometer values to calculate the robot's most probable position. While this position has the highest probability, other hypotheses are possible and are tracked in parallel. After initialization, all particles are randomly distributed with equal weight. When processing new sensory data, each particle position is matched against the retrieved sensor data and assigned a new weight according to its probability. The advantage of the algorithm is its dynamic assignment of particle counts. The more particles that point to the same hypothesis, the more probable the hypothesis and the fewer particles are needed. In contrast, if all current hypotheses are improbable, more particles will be generated and tracked. Once a suitable position estimate has been found, further position updates take the position history into account. Thus the algorithm requires very few resources when iteratively updating the position estimate.

3.4.2 Non-Functional Requirements

Robot services, such as localization, have to be available early in the start up process, so that higher layers can access relevant data. For example, the initial pose hypothesis is the first estimated position from the localization component. It is either obtained automatically, by mapping sensor values in a particle-based localization algorithm, or, if that fails, it is set by the user placing an icon on a map-like interface. This is important, as higher layers might broadcast their initial state to other participants and rely on early receipt of correct data.

Due to high innovation rates for robot hardware and drivers, the platform shall allow for the easy configuration of different devices. It must be possible to support hardware other than that described in this work without major architectural changes to the concept of a meta-platform.

The platform ensures that operations are started but does not guarantee their completion (as they might be dependent upon the hardware state). Nevertheless operation failure and success shall be notified to the calling application, typically an agent.

The user is likely to choose a different MAS and MRS from those described in this work. Therefore it shall be easy for the user to adapt the meta-platform for their MAS and MRS of choice.

The platform shall be operating system independent and compatible with today's standard PC environments.

Software components shall be designed for re-use in different use-cases.

The platform shall provide a graphical user interface to start, stop and modify scenarios.

A robot simulator shall be provided to allow the platform to be exercised without real robot hardware.

The hardware might enter an erroneous state and the software shall therefore provide error prevention and recovery facilities.

The software shall be adaptable to new robots, devices and behaviors.

As another software layer necessarily introduces more code, it shall introduce the smallest possible delays and shall be performance optimized. The latter requirement also applies to all driver configurations.

3.4.3 Performance Requirements

The system shall be optimized for typical use-cases. Furthermore the requirements for each layer vary. The execution overhead introduced by the MAS layer cannot normally be changed. Such overheads include delays introduced by non-user code, such as agent creation and destruction, task management and distributed communication. However the performance of agent code implemented by the user varies according to the task and must be optimized. Agent designs shall follow the design guidelines given by the multi-agent system (Jadex in this case).

An agent must typically remain responsive to the system at all times and must therefore split its actual task into small steps of work. In Jadex terminology, an agent performs tasks in steps called *IComponentSteps* that are efficiently planned and executed by the Jadex scheduler. In order to promote responsiveness, tasks shall be asynchronous wherever possible. Hence an execution step shall either be very short or it shall be capable of subdivision into smaller steps. Failing that, a task shall provide a callback or future interface so that an agent can set the task running asynchronously and receive notification when it finishes. This releases the agent's resources for other activities. As the agent will typically use robot services from the RSAL, these services have to be designed in such a manner.

The RSAL consists mainly of robots, behaviors and devices. Devices are hardware dependent and thus have to fulfill real-time requirements. A laser ranger may deliver range values periodically (for example at 10 Hertz) and the appropriate software device must respond at this rate. Other hardware might require servicing at different rates. As a robot can have many devices their servicing must be very fast.

As described, requirements differ from top to bottom of the architectural model. Each layer has to fulfill its own (performance) requirements. Moreover each layer has to contribute to the overall scalability requirement of the meta-platform, as defined in section 3.5.

3.5 System Architecture

The meta-platform consists of three layers. The user interacts with the top-most layer, the MAS. More specifically the GUI is provided by Jadex[3], the MAS in the current implementation. Jadex provides a clear and efficient interface for starting scenarios and individual agents and for passing the necessary arguments. The arguments differ according to whether an agent or a scenario is started. When starting an agent, the actual type of agent is also significant. The following arguments are typically passed when creating an agent via the Jadex GUI:

- Host: The *host* field is a string specifying the networked computer on which the Player server is running. It can take a name resolved via the Domain Name Service (DNS) or an Internet Protocol (IP) address. This field defaults to *localhost* and need only be changed when the agent and the robot controlling

[3]Screenshot at http://jadex-agents.informatik.uni-hamburg.de/xwiki (July 11, 2011)

computer are at different addresses. For example, this is the case when all agents run on a central computer rather than on the robot local computer.

- Port: The *port* field contains the computer port (of integer type) on which the Player server is listening. It defaults to the Player standard port 6665. Again, this is only needed when the agent is not running locally on the robot computer.
- (Robot) Identifier: *robId* is the network-unique numerical identifier of the robot. This field is used to distinguish between robots taking part in a scenario and to map them into a virtual environment.
- (Device) Index: It is possible to have multiple sonar, laser or other devices on the same host/port combination. In this case the *devIndex* can be used to define the appropriate device tuple, where the tuple is a combination of certain devices provided to the agent. The order of tuple fields must be consistent, such as sonar first and laser second. This numerical field is only needed when running multiple agents on a central computer and corresponds to the Device (class) *index* field (figure 3.18).
- Initial Position: This x, y and angle tuple, consisting of double-precision values, optionally specifies the robot's initial position. The angle field defines the robot initial (front) orientation. The tuple defaults to (0,0,0) and any different values (if set) modify the robot localization device accordingly. The units are: meter (x), meter (y) and degree (angle). When the default value is kept (the tuple is not set) the robot position will be estimated using the laser ranger and odometer readings. This tuple typically remains unset and the localization device automatically estimates the initial position. Nevertheless, if position estimation repeatedly fails, setting the tuple helps to correct the robot's initial position. Moreover setting this field affects the robot proxy in a virtual environment, if any is available, as it sets the proxy robot directly to that position.
- Laser: This boolean flag can be used to disable the laser, which is normally enabled. This affects robot wall-following and obstacle avoidance but does not affect robot localization.
- Simulation: Another boolean flag that is used to disable automatic searching for a simulation environment on the network (see section 3.6.2.2 on page 35).

Scenario arguments differ from those of agents. Typically a scenario does not need to be given an argument as fixed arguments are configured internally for each agent. Nevertheless a scenario typically has several configurations that can differ in the number of robots taking part or in the choice of virtual, real or mixed reality environments.

Note that although the Player/Stage MRS has a built-in robot navigation GUI that can observe current plans and waypoints, the typical user interface is the MAS.

The MAS includes agents that will be started according to the scenario. Additional agent-related components, such as services, are also included in the MAS layer. Agents call robot facilities from the RSAL. A robot can be controlled using its external interface or using interfaces to attached devices, such as a planner device for navigation.

The interface between the MAS and RSAL layers is asynchronous. It provides call-back facilities that allow different timing requirements to be accommodated and that decouple the MAS and MRS. The RSAL robots are device modules that include calls to the MRS. In the current Player/Stage MRS these calls are passed to *PlayerClients* and are synchronous (as supported by the MRS). Thus the device implementation has to decouple these synchronous, blocking calls and its asynchronous interface. A PlayerClient is, in RSAL terminology, a *DeviceNode*. A DeviceNode is a unique access point within the network scope that provides interfaces to a group of hardware devices (section 3.6.2.3 on page 38). Normally each device has a driver included in the MRS to access the hardware. For example, the p2os Player/Stage driver controls the Pioneer motors, whereas another driver (*hokuyo_aist* from the GearBox project[4]) controls the laser ranger hardware. An overview of the architecture is given in figure 3.4 and 3.5. The latter depicts all three layers as well as internal modules.

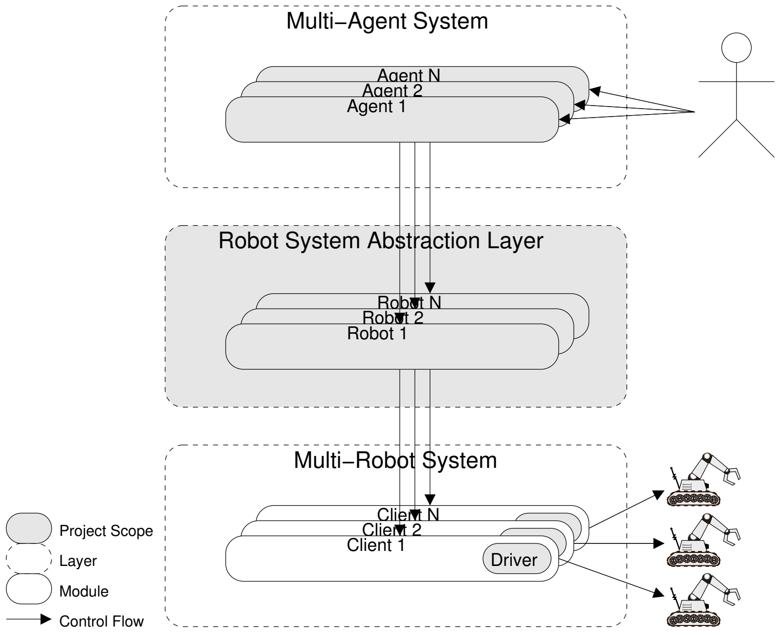


Fig. 3.4: The user interacts via agent scenarios. An agent might have a dedicated robot module with specific features. The robot module calls down to the robot hardware client that has the necessary drivers for interaction with the hardware.

[4]http://gearbox.sourceforge.net

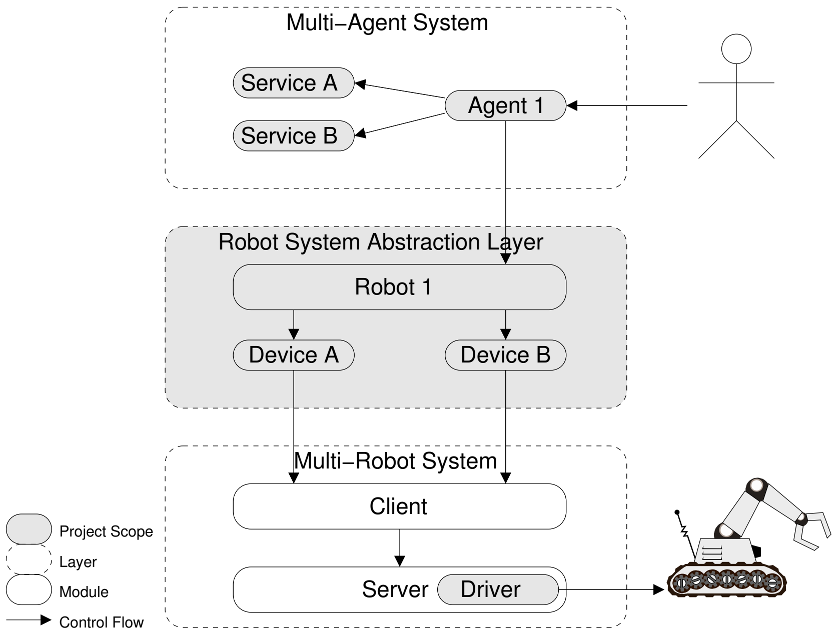


Fig. 3.5: The MAS, here Jadex, is the top layer. The user-invoked agent uses dedicated services in order to communicate with other agents. The choice of the services selected depends upon the type of agent and its target. To interact with its environment the agent controls a robot (class) that interacts via devices with its underlying hardware. The hardware interface is encapsulated within the MRS, here Player/Stage.
Player/Stage consists of client and server modules that communicate over the network via sockets. The server receives commands and issues them via its drivers to the (robot) hardware.

Agents typically use services to perform tasks and for communication throughout the network (figure 3.6b). These communication services are used to create information channels to which an agent can subscribe in order to read or publish interesting information (figure 3.6a). Each agent can subscribe to a different set of services according to its activities and abilities.

The MRS typically provides another method of remote access to robot hardware, shown in figure 3.5 as a client/server system. The client is a local proxy for the device hardware from the caller's point of view. Any call is transparently transferred by the client through the network to a server. The server, which normally runs on the robot-attached host computer, parses messages and controls the hardware via its drivers.

3.6 Detailed Design

This section will describe all (meta-) platform components showing those of each layer. Each component will be described in detail. The MAS and MRS layers might

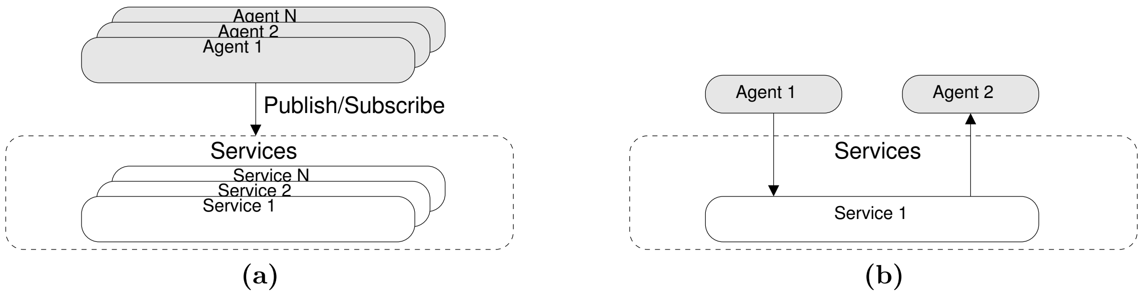


Fig. 3.6: Agent Services: (a) Agents can subscribe to arbitrary services. The choice of services depends upon the type of agent (what are its facilities) as well as the information policy (what information is the agent allowed to send and receive). A message sent through a service is published to all subscribers. (b) A basic communication sequence between two agents via a service.

provide further unshown components (depending upon the actual systems used, such as Jadex, JADE, Player/Stage or ROS), which, while necessary, are not the focus of this work. Components described in this section are new and created for this work. The driver configuration handles MRS features that are specifically arranged to fit the TAMS indoor environment. Figure 3.7 depicts a component overview of the system.

In the following sections, the individual component designs will be described in detail. It is worth mentioning that some aspects of the detailed implementation have been omitted for simplicity. Omissions include method and constructor arguments (if not mentioned within the description), methods with similar functionality (returning only different data types from the same source, for example), private methods and getter/setter methods. Typically each class has its own logging facility through the Java Logger, which is also omitted from the diagrams. To obtain a comprehensive API specification, refer to the online documentation[5].

3.6.1 Agents, Services and Scenarios

The MAS layer contains components related to high-level services that include cognition and distribution. It is implemented in Java and XML. Java is used for agent definition including initialization, body and de-initialization. XML serves as a scenario container, in which any scenario participants are declared along with related initialization parameters. A scenario can consist of different configurations, such as varying numbers of agents from the start. These configurations (in Jadex: *Applications*) can be grouped within one scenario (file). The current implementation prefers the Jadex agent model of *MicroAgents* over the more complex BDI agents

[5]http://rockel.cc/master/javadoc (July 26, 2011)

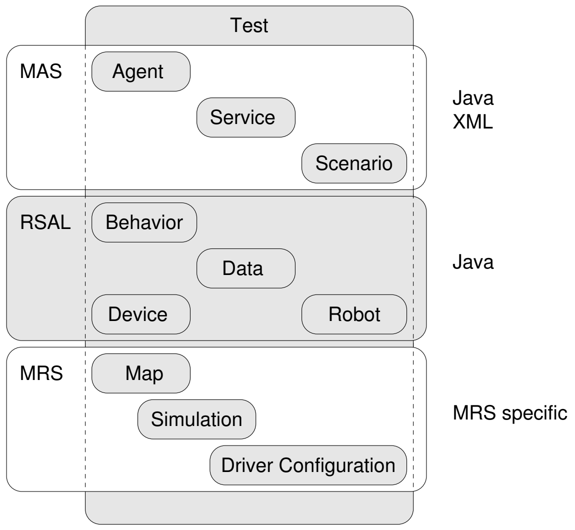


Fig. 3.7: A complete static overview of all components belonging to the meta-platform. Horizontal layers indicate the component abstraction and show the layer to which they logically belong. A test component is also included in the platform. As it tests all components it is indicated as an orthogonal component. Text on the right of the figure shows the techniques applied to components in each layer.

for reasons of simplicity and efficiency. Nonetheless, any agent model provided by the MAS is supported.

An overview of all currently implemented agents is shown in figure 3.8. Each agent body consists of three methods that model the lifetime of the agent. When the agent is invoked its *agentCreated* method is executed to initialize data structures; during its lifetime all steps implemented in the *executeBody* method will be started; when it is terminated the *agentKilled* method typically performs clean-up operations before the agent is actually killed. In order to communicate with other agents, each agent typically has at least one service. Whereas some agents have robots and devices, some only have a blackboard (see section 3.6.2.1 on page 32), depending on the type of agent. Moreover the parameters passed to an agent upon its creation also depend upon agent type. In figure 3.9 some important agent classes are shown.

Present services and their inheritance are shown in figure 3.10. All services inherit from a basic service class and implement sending and receiving facilities to communicate with subscribers. A subscriber can filter received messages so that it receives only those of interest. Each service represents an information channel and follows a defined (simple) protocol.

Scenarios are groups of agents with pre-defined arguments. Arguments can be set in the Jadex GUI when starting a scenario and are defined in XML rather than in Java. The Jadex facilities allow for a mapping between the Java class name and the

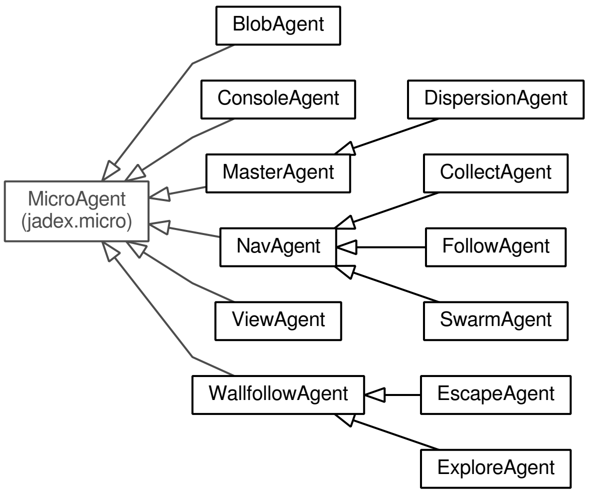


Fig. 3.8: The current implementation uses Jadex MicroAgents. All agent classes inherit from the MicroAgent class. Abstract agent classes such as the MasterAgent and NavAgent have specialized classes designed for certain tasks.

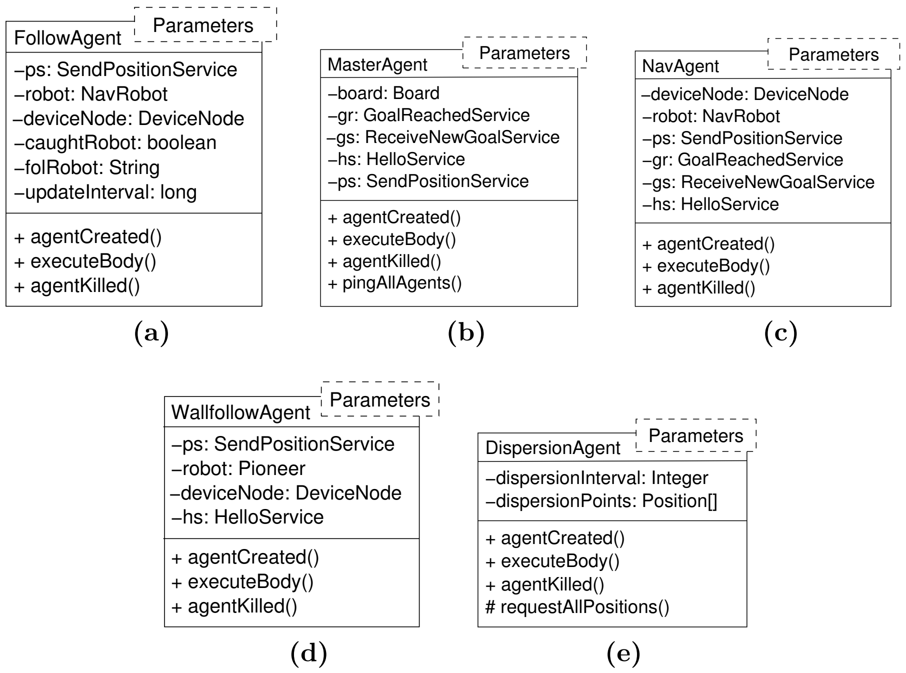


Fig. 3.9: Agent Classes: (a) A FollowAgent subscribes to the SendPositionService and has a NavRobot to control. The identity of the robot it should follow and the interval at which it should check for positional changes are passed as parameters. (b) To store information on scenario participants, a MasterAgent has a blackboard and dedicated service subscriptions. (c) The basic agent for navigating a robot is a NavAgent. (d) A WallfollowAgent has a Pioneer robot that is behavior-controlled to follow a wall. (e) To control other agents on the network a DispersionAgent is capable of organizing swarm formations.

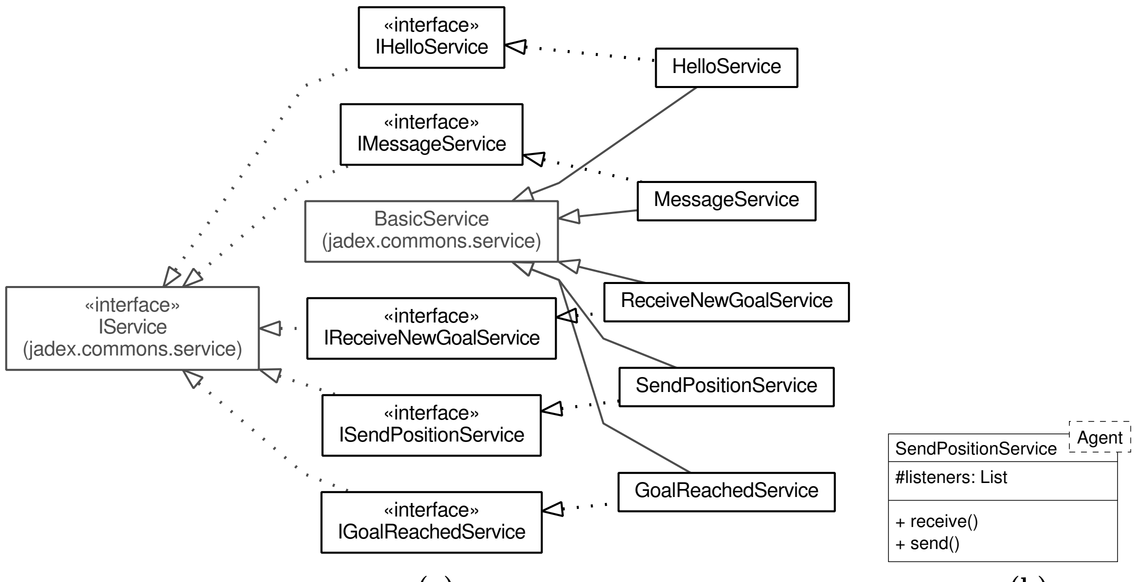


Fig. 3.10: (a) All services inherit from BasicService (continuous arrows). Services implement sending and receiving facilities to communicate with all subscribers. Any subscriber can filter received messages. Each service represents a dedicated information channel. Interface implementations are indicated by dotted arrows. (b) Basic declaration of a service class.

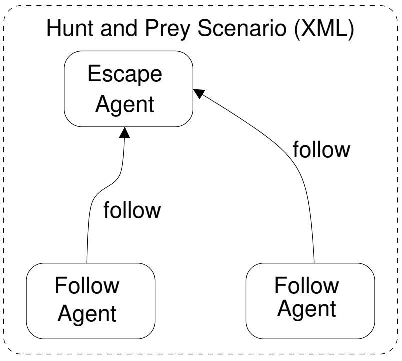
label in the XML file. In figure 3.11 an example scenario definition is shown. All demonstration scenarios implemented are described in section 4.2.

3.6.2 RSAL Middle Layer

The RSAL layer embeds data, robot, device and behavior components. Furthermore it serves as the middle layer between MAS and MRS.

3.6.2.1 Data

The data component contains central data types used throughout other components. The fundamental type is *Position* which is depicted in figure 3.12a. The *distanceTo* method calculates the Euclidean distance ($\sqrt{|x - x'|^2 + |y - y'|^2}$) between two positions. *equals* returns true if two positions match exactly, false otherwise. *isNearTo* returns true if one position is near to another, considering the given acceptable deviation. An acceptable delta is given in meters for the planar distance and in radians for the angle. *getCartesianCoordinates* considers the actual coordinates in polar form (x takes the range and yaw the angle θ) and returns its transformed ($x = r \cdot \cos(\theta),\ y = r \cdot sin(\theta),\ yaw = 0$) cartesian coordinates.



```
[...]
<arguments>
  [...]
</arguments>

<componenttypes>
  <componenttype name="Escape0" [...]
  <componenttype name="Follow0" [...]
</componenttypes>

<applications>
  <application name="1 Robot, Tams floor, real">
    <components>
      <component type="Escape0" name="Escape0">
        <arguments>
          [...]
        </arguments>
      </component>
    </components>
  </application>

  <application name="3 Robots, Tams floor, real">
    <components>
      <component type="Escape0" name="Escape0">
          [...]
      <component type="Follow0" name="Follow0">
          [...]
    </components>
  </application>
</applications>
```

Fig. 3.11: Here the XML definition of the Hunt and Prey scenario is shown. On the left, a schematic of an example configuration is given: Two follow-agents hunt or follow an escape agent (as long as it is not caught). On the right is an extract from the corresponding XML application file. A lot of XML-specific lines and other components and parameters have been removed for simplicity. A scenario can consist of different configurations (called Applications in the Jadex convention). At the top any global arguments are defined. These are valid for all available applications and are invoked when an application is started. The arguments set a default value that can be manually changed when an application is to be started. Below the arguments, there follows a declaration of all available components, here agents. The remaining lines define the available applications, each consisting of dedicated components and their arguments. An application has a literal representation that can later be selected within the Jadex Control Center (JCC), the graphical user interface. It can be seen that this XML definition file contains two applications.

getGlobalCoordinates performs a affine matrix transformation to obtain the position coordinates in a global coordinate system referenced by the (global) position given. A homogeneous matrix contains the rotational and translatory transformation as shown in equation (3.1). A typical use-case for this method would be the transformation of the position coordinates of an object in the robot's local coordinate system into global coordinates within the world frame. For example, the camera mounted on top of the robot detects an object and its image coordinates are transformed into robot-local coordinates; these must then be transformed into the world coordinates in order to exchange position information with other robots. The robot frame origin and orientation can typically be obtained by the localization driver (see section 3.4.1). In order to preserve object orientation in the world frame, normalization of the object's and robot's combined orientation is necessary, as shown in equation (3.2). The normalization is done by the static method *getRelativeAngle*, which normalizes the given angle within the range $-\pi$ to π where π itself is excluded.

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}_O^W = \begin{pmatrix} cos(\theta_R) & -sin(\theta_R) & x_R \\ sin(\theta_R) & cos(\theta_R) & y_R \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}_O^R \tag{3.1}$$

$$\theta_O^W = norm(\theta_R^W + \theta_O^R) \tag{3.2}$$

Big letters indicate frames (coordinate systems), such as for the world (W), the robot (R) and the object (O). A vector can have superscript and subscript frame letters. If a vector is given with a superscript frame letter it means this vector contains coordinates relative to that frame origin. In other words, the coordinates are local to the frame origin. A subscript frame letter means that the vector contains coordinates of the given frame (origin) relative to another frame (origin). The position class and frame conversion are depicted in figure 3.12.

Goals within a scenario can be specified by the *Goal* class (figure 3.13). They are typically defined by a planar position within the world frame. A Goal is used for example by the *BoardObject* of the blackboard implementation.

Another central type is that of a *Board* depicted in figure 3.14a, which implements a blackboard pattern. With this class, memory is provided to store various chunks of data that would represent notes on a bulletin board in the real world. This blackboard has several advanced features compared to its real world analogue. Notes or board objects can be ordered into topics and can have an associated timeout value. If the timeout occurs, the board object is automatically removed from the blackboard. Note that every new board object has a default timeout value, which should be changed. Important considerations in the design of this class are algorithm efficiency and the minimization of the memory footprint. Consequently, the class does not have its own execution thread. Activities that might be executed in the

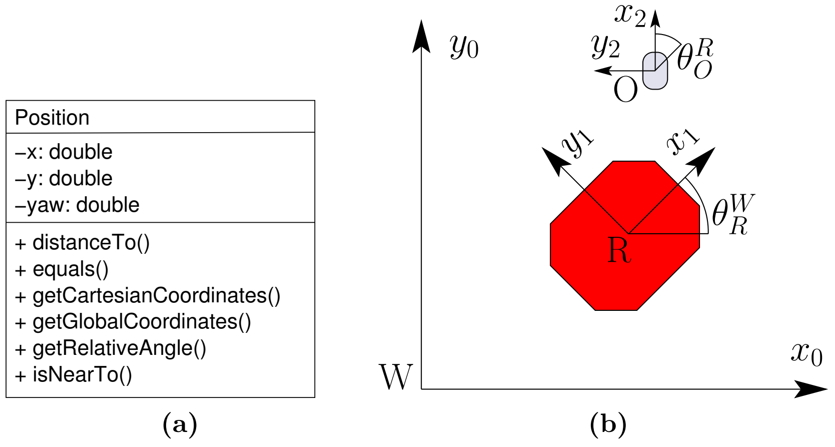


Fig. 3.12: (a) The *Position* class represents a two dimensional position. Position objects are frequently used throughout the system, for example when navigating the robot or during agent communication. The latter use imposes a footprint constraint on serialized objects sent through the network. Nevertheless the current design “costs” only 24 bytes, as planar positions can be represented by three coordinates, each requiring eight bytes. As for later improvement, another augmented class is necessary to represent a three dimensional position. (b) The world frame (coordinate system) *W*, the robot frame *R* and the object frame *O*. Frame transformations are provided by the Position class.

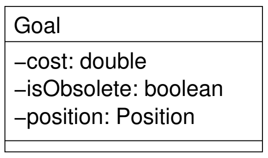


Fig. 3.13: The Goal class consists only of its fields, setters and getters. The *cost* field is intended to store the current cost to reach the goal; on reaching the goal, the *isObsolete* flag is set to true. When the goal can be found at a specific location the *position* info contains the coordinates.

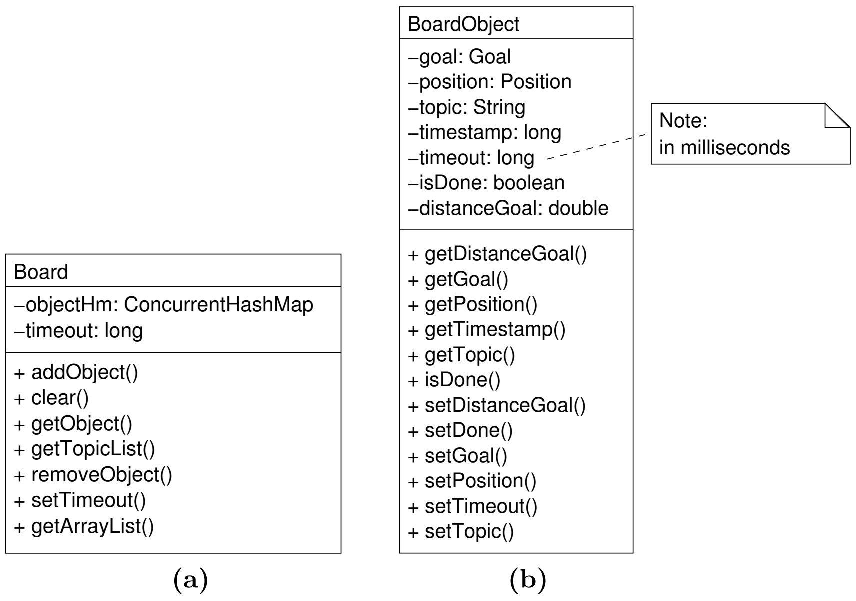


Fig. 3.14: Flexible information storage is achieved using the blackboard design pattern. (a) A Board object can store and manage many BoardObjects within an associative memory type. Objects can be accessed by a topic-sensitive filter. An advanced feature is a garbage cleaner that efficiently clears expired (too old) board objects. (b) A board object should be flexible in terms of the information it can contain. The fields are optional and should be used according to the information being stored by the caller. Unless the caller specifies its lifespan, a default timeout is set when creating a new object.

background by a class thread, such as removing expired board objects, must be performed when board methods are externally called, slightly increasing method latency. Information is provided within a *BoardObject*.

The type *BlobfinderBlob* (figure 3.15) is used to store information about colored areas of an image retrieved by the *Blobfinder* device.

3.6.2.2 Robot

The *Robot* component contains a generic Robot class from which currently implemented specialized robots inherit, as depicted in figure 3.16. The Robot base class represents a mobile robot moving on the ground. The specialized Pioneer robot adds a behavioral model and states that can be monitored. It can be controlled either with the *setCommand* method to give it manual motor commands, or using the *setWallfollow* method to activate autonomous wall-following. Both behaviors apply obstacle avoidance (figure 3.23).

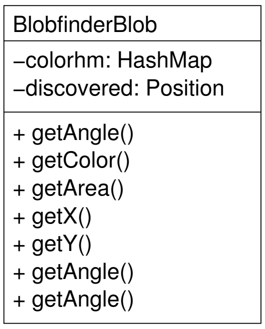


Fig. 3.15: This class has a color hash-map that stores key-value pairs of color codes and their names as well as positional information on where the blob was detected in the world frame. Methods provide access to the relative image position of the blob.

Specializations of the Pioneer class are the *NavRobot* and the *ExploreRobot* classes. The former overwrites the super class update method (section 3.6.2.3). The Pioneer update method contains the state-machine (handling) of the implemented subsumption behavior (section 3.6.2.4). In order to avoid unnecessary behavior, a sub-class must override the update method, for example to let the robot be planner-controlled. That means this robot can be given a goal position on the map and will follow the shortest trajectory to the goal while avoiding walls and other obstacles. The ExploreRobot class extends its super class with additional exploration algorithms.

Creating a Robot object requires a list of devices belonging to the new Robot. At a minimum, the Robot needs a *Position2d* and a *Ranger* device, others, such as a *RangerLaser*, *Planner* and *Localize*, being optional. With the addition of a special *Simulation* device to the device list, the robot becomes aware of the simulation environment. In this case, the Robot class *setPosition* method, when executed, will try to locate the simulated robot by requesting its *robotId* from the simulation. If found, the simulated robot can be accessed through the Simulation device. An optional Simulation device is used to update the simulated robot's position in the virtual environment (besides setting the robot's localization device to update the odometer, if present).

3.6.2.3 Device

The device class provides the abstract class for all real robot hardware devices, and for virtual devices such as those of the *Blobfinder* or *Simulation* type. Devices can inherit from each other for specialization, which is the case for the laser ranger and sonar ranger. These devices both inherit from the type *Ranger*.

A device is typically active, in the sense of having a dedicated thread that updates its internal state continuously. This design decision is based upon the advantages of this approach. As the lowest-level components within the RSAL layer, devices are

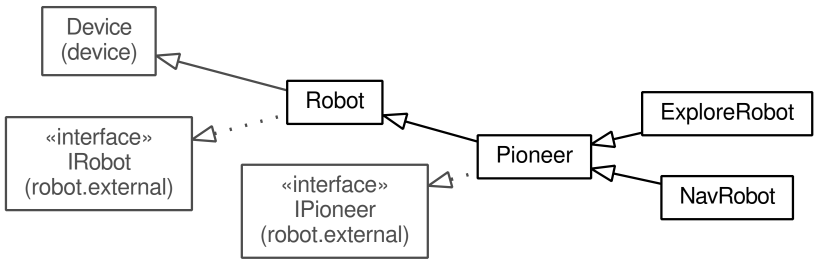


Fig. 3.16: The current implementation provides a Pioneer class, which can be specialized as exploration and navigation robot classes. As the Pioneer class inherits from Robot, which itself inherits from Device, it has its own runtime context implemented as a thread. Internal robot processes, such as dedicated state-machine handling and behavior execution, benefit from this implementation as well as from optional sensor data processing, such as parsing of video camera images.

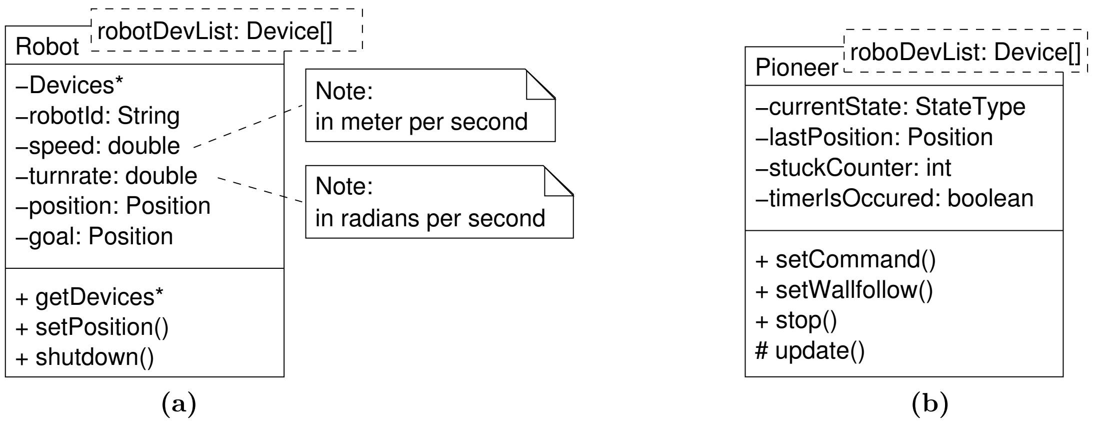


Fig. 3.17: (a) The Robot class represents a robot that can move in two dimensions. It will, if possible, use any devices provided in the given device list. A robot can have a current location and a goal position (in the global coordinate system). Robot object creation requires a list of available devices. This list can contain arbitrary devices anywhere on the network, not necessarily attached to the robot. The robot object will initialize recognized devices upon creation and store them in a device list, internal to the Robot object. External entities, such as agents, can request the use of devices belonging to a Robot. A *shutdown* method handles robot internal memory-cleaning operations. (b) The Pioneer class is a specialization from Robot and represents a minimal or standard configuration of a Pioneer-2DX robot at the TAMS laboratory. Specialized classes can inherit from the Pioneer class to add new devices. The Pioneer implements a behavioral model and monitors the model's states.

normally heavily dependent on sensory input, such as range values or localization hypotheses from the underlying drivers and hardware. This imposes strong real-time constraints on the design (and hardware dependencies).

In the high-level interface provided to a Robot class (or to even more abstract "users", such as agents) the requirements are high availability, concurrency and asynchronism. These can be achieved by using the thread to poll the hardware and fill internal data structures. Requests for device data can then be provided with the latest readings without the delays associated with access to the hardware. This is indicated in figure 3.19. Certain operations can take a long time, such as arriving at a goal position. Therefore the callback pattern provides the caller with information only if relevant data is ready, which lets the caller conserve its resources for as long as possible.

As some devices have other devices attached (logically or physically), objects can be linked within an ordered hierarchy of devices. For example a robot is represented as a device and typically has other devices, such as ranger sensors, mounted. This introduces recursion into the class model. To cope with this requirement, an object of the Device class has an internal data structure listing other devices.

In order to be able to search for or compare devices at runtime, several related methods are provided. To check for identical devices the *equals* method is available. A similar method *matches* also compares the given devices but can accept a device that does not have all information defined. Such a partially defined device can be viewed as a device template or search mask. The list counterparts of these two methods, implemented as *isInList* and *matchesList,* search the internal device list with the information provided.

The abstract method *update* must be implemented by sub-classes. It contains the periodic task that a device must maintain to remain in sync with its underlying hardware. This method is called periodically within the Device (thread) context. Typical update method tasks are reading the most recent values and updating internal data structures and states. For a gripper device, the periodic task would be to read the current optical sensor states via the Dio device (page 41), to update the internal state machine and, if requested, to set a new state. The periodic task is processed in the run-loop of the device.

After each update cycle, the Device class suspends the device (thread) for the time set in the class *sleeptime* field. Upon resuming, the update is repeated. The duration of the update processing is typically very much smaller than the sleeptime. Unless the device is shut down, this cycle will repeat forever. There is a trade-off between a highly responsive device (which has less sleeptime) and a resource efficient one (which has a higher sleeptime). Whereas some devices, such as grippers or localization devices, take rather long times to update their internal states, often more than half a second, there are devices that require small sleep times: for example, less than a tenth of a second, for a laser ranger device. Typical robot devices work well with a sleep time of 100 milliseconds. This is a good trade-off between responsiveness and resource saving and is the default value if not changed.

A Device object has to be explicitly started by the *runThreaded* method. This invokes internal threading and takes care of the startup of internal devices. To stop a device, the *shutdown* method must be invoked, which also handles sub-device shutdown. Both startup and shutdown, where there is a list of internal devices, can be seen as a recursive cascade that calls all devices in the list. If a new device has an embedded, dynamic, data structure, it should overwrite the shutdown method.

A special Device class is the *DeviceNode*. Its purpose is to encapsulate all device access from the underlying MRS. Different MRS might have similar but nevertheless slightly different methods to access their devices. A DeviceNode must be created with a list of hosts and devices, where a host is identified with its name or address and the port on which a client listens. The device list contains templates of any combination of *id*, *host*, *port* and *index*, some of which may be omitted to provide a device mask matching multiple combinations. Here the *id* field identifies the type of device; *host* and *port* fields are as described in section 3.5. It is possible to have multiple devices of the same type (*id*) on the same host/port combination. In this case the *index* can be used to define the exact device. Upon creation of the DeviceNode object, the given hosts will be searched for devices matching any templates. When a matching device is found it is added to an internal device list; when the device is bound to a new PlayerClient (a Player/Stage specific device node, see section 4.5), the PlayerClient is added to a DeviceNode internal list (which is a dedicated field of the DeviceNode class). Note that the PlayerClient class is provided by Player/Stage and, as it introduces a dependency on the MRS, must always be encapsulated within a DeviceNode.

After initialization, a DeviceNode has a flat list of all sub-devices and DeviceNodes (this is the list/field inherited from the Device class, figure 3.18a). During runtime, DeviceNode provides a dynamic interface for managed devices and moreover allows masked searches for specific devices.

The *Localize* Device class implements features that locate the robot within a world frame. It uses the underlying MRS localization driver, currently AMCL, encapsulates any dependencies and provides a consistent interface. Furthermore it decouples performance constraints from the driver by providing an asynchronous interface. The user of the class can rely upon the immediate return of the latest position data. The interface provides a callback notification feature. Any listeners interested in periodic position updates can subscribe with the *addListener* method. Clients interested in current location information use the *getPosition* method. In the run-loop method *update,* the localization device is periodically updated. To allow the position belief to be explicitly set, the method *setPosition* is provided. As most localization algorithms keep track of the current position hypotheses with a covariance matrix, the matrix is contained as a field. This field sets the probability distribution at the start.

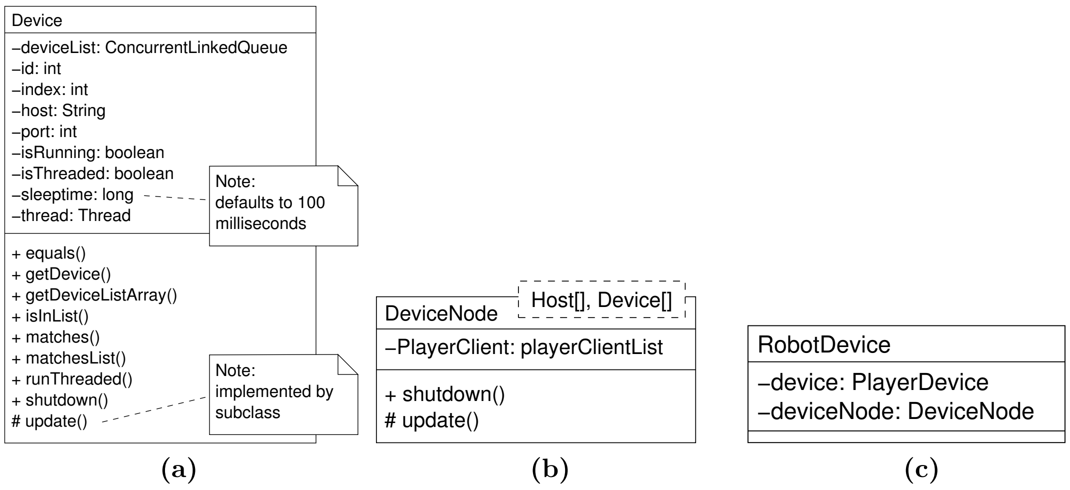


Fig. 3.18: (a) The Device class holds its optional sub-devices in a linked list. Normally a device can be identified by its *id* (type), the *host* to which it is attached, the *port* and, in case there are multiple similar devices, an *index*. Other fields are related to device threading. (b) A DeviceNode encapsulates the underlying MRS client to which devices are connected. It overrides the shutdown and update methods to remain in sync with the lower layer. (c) Each robot hardware device is a *RoboDevice*.

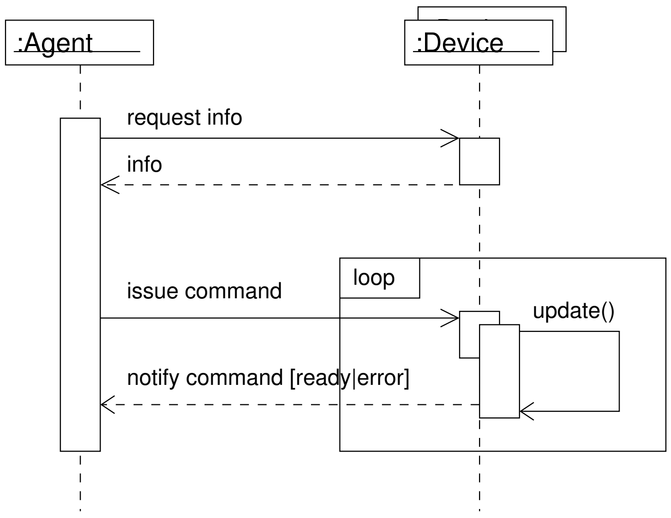


Fig. 3.19: Devices provide an asynchronous API that returns the latest device state. Internally, each device has its own thread that keeps in synch with the underlying hardware. The thread periodically executes the *update* method and “sleeps” in between.

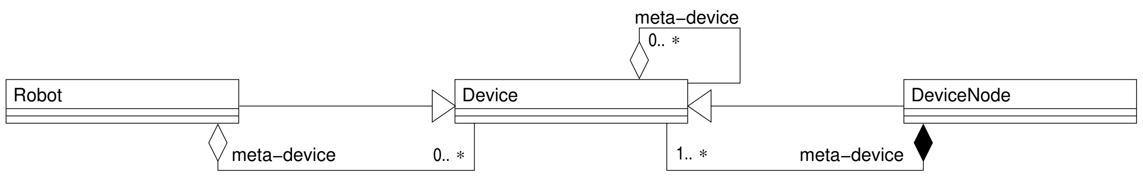


Fig. 3.20: Devices can implement a recursive hierarchy. In this case a device is called a meta-device, such as a Robot or DeviceNode. Note that Robot and Device can implement arbitrary devices whereas a DeviceNode binds physically connected devices. Thus a hardware device requires a DeviceNode but not any other meta-device.

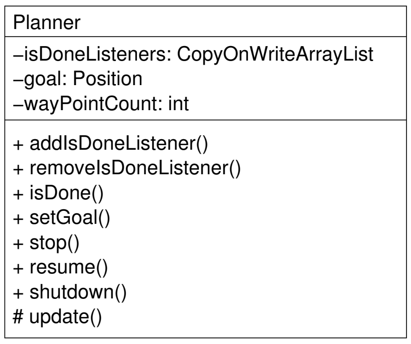
As time passes and the robot moves, odometer and ranger data changes will lead the localization device to update the matrix periodically. The matrix consists of nine elements, since a planar robot position can be defined by three coordinates x, y and its angle θ. The elements contain all permutations of coordinate pair covariance to measure to what extent a pair of coordinate directions correlate with respect to the combined deviation. For example the covariance of the pair x,x indicates the deviation between the real position and the position belief in x direction, whereas the covariance of the pair x,y states the deviation in the x and y direction. Since some permutations contain redundant information, the matrix is symmetric and can be defined by six elements only (the lower half of the matrix). The covariance matrix implemented is depicted in equation (3.3) and equation (3.4), where a_i is a raw coordinate value, $\overline{a}$ is the mean of all values and N is the number of all values. The Localize class declaration is shown in figure 3.21b.

$$COV^{3x3} = \begin{pmatrix} cov(x,x) & ... & ... \\ cov(y,x) & cov(y,y) & ... \\ cov(\theta,x) & cov(\theta,y) & cov(\theta,\theta) \end{pmatrix} \tag{3.3}$$

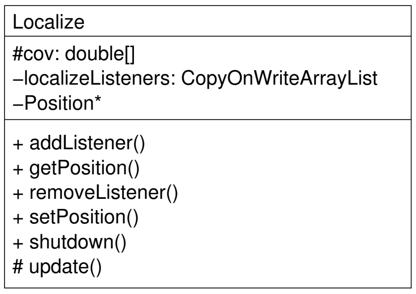
$$cov(a,b) = \Sigma(a_i - \overline{a})(b_i - \overline{b})/N \tag{3.4}$$

The *Planner* class provides access to the underlying path planning driver. It encapsulates different drivers and specifies a generic interface. Client access typically involves subscribing to the planner for a new goal with the *addIsDoneListener* method. The client will then be notified either if a goal set with the *setGoal* method is invalid, or if the planner has aborted, or in the successful case of reaching the target location. All listeners are managed in an internal list and will be notified of events accordingly.

It is possible to pause the current plan with the *stop* method and also to *resume.* The update method keeps track of the current plan and its parameters, such as the waypoint count and the current waypoint. A waypoint is an intermediate goal set for the local path planner. The Planner declaration is depicted in figure 3.21a.



(a)



(b)

Fig. 3.21: The navigation feature is implemented by the Planner (a) and Localize (b) classes. Both classes provide asynchronous interfaces with callback notification triggered by appropriate events. Throughout their simple interface they hide the complexity of the underlying navigation algorithms and drivers.

The *Simulation* class provides access to a simulation environment such as that provided by Stage. Interaction between software clients and the virtual world is provided so that the position of dynamic objects, such as robots or furniture, can be changed dynamically. This is currently used to test software units with repeatable environmental configurations. Moreover this interface is integrated to allow a mixed reality approach. This means that the positions and orientation of real robots can be used to place virtual proxies of those robots into a simulation environment. Such a scenario allows interaction between real and virtual robots. In order to allow access to the simulation, the method *setPositionOf* is used. This method takes a Position object and a string identifier that selects the Stage simulation object.

The *Blobfinder* class represents a blob finding device, such as a camera capable of detecting colors in an image. It is used to provide the system with a camera-like device without actually having such a device. Thus it is to be used within a simulation environment. The class provides a callback interface to notify clients upon blob detection and the callback delivers information on the blob count as well as on the detected blob itself (figure 3.15).

In order to provide access to the robot-attached gripper, the *Gripper* class has been added. It provides a callback interface to notify clients when the paddles are opened or closed, as well as when the paddle lift is raised or released. An internal state machine keeps track of the hardware. Each of the states can be requested manually and will be notified individually upon completion or abort. For complete control of the gripper hardware, other devices might also be necessary. In the current implementation the *Actarry* and *Dio* devices are used: the actuation array interface (*Actarray*) provides access to robotic arms, or in this case the gripper, and the digital input and output interface (*Dio*) provides access to the photo diodes of the Pioneer-2DX gripper hardware.

The *Position2d* class provides access to the motors of a mobile robot. It allows the motors to be enabled or disabled, their speed and turn rate to be set or read and the odometer position estimate to be set. Note that a mobile robot, such as the Pioneer robot, does not move like a car. Typically a mobile robot uses a differential drive with at least two wheels, each attached to a dedicated motor. The robot's speed is some combination of the individual wheel speeds and the turn rate is derived from the difference in wheel speeds. This class hides the underlying design from the user.

An overview of all currently implemented devices and their inheritance is depicted in figure 3.22.

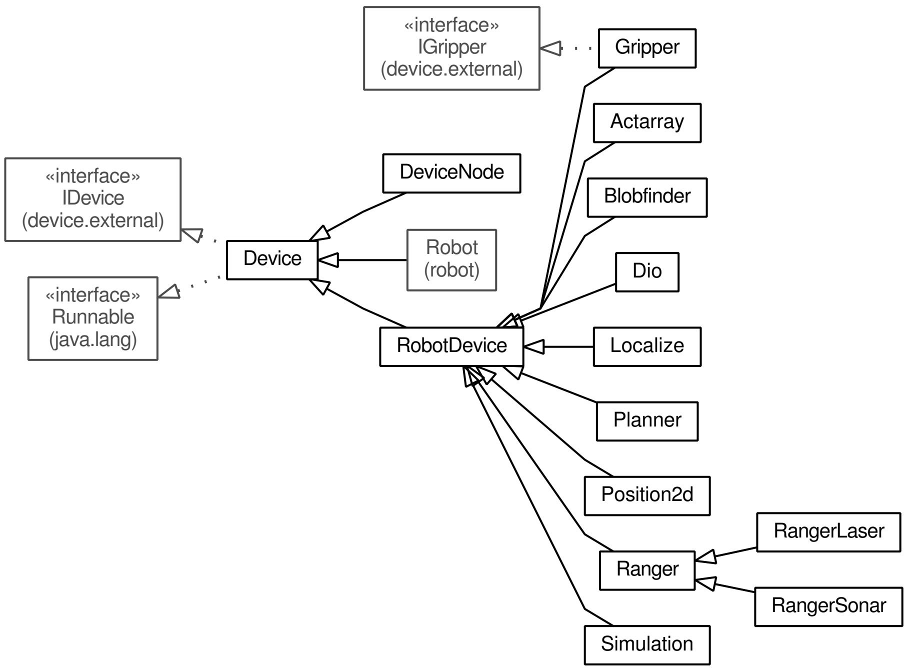


Fig. 3.22: An automatically generated inheritance diagram. The Device class is a super class of a variety of specialized devices and meta-devices. It implements the Java *Runnable* interface. As all sub-classes inherit this interface, all devices are capable of handling device related things transparently in their own context, thus increasing responsiveness and providing an asynchronous interface.

3.6.2.4 Behavior

The Behavior component implements a basic set of behaviors for a mobile robot. The behaviors are combined in a hierarchical subsumption architecture [Bro86] to provide robust obstacle avoidance and escape strategies in case the robot becomes stuck. Sensory input is provided by laser and sonar rangers. These sensor inputs are combined to allow accurate detection of environmental obstacles and to avoid weaknesses of standalone sensors: highly reflective or glass walls confuse laser rangers

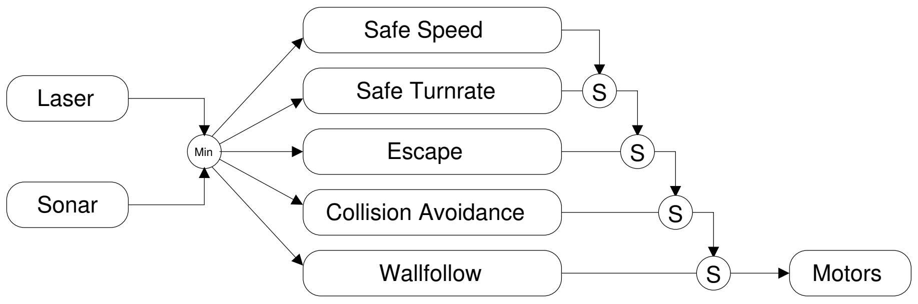


Fig. 3.23: The implemented behavioral model fuses laser and sonar range values. The minimum of the two values is adopted for reasons of safety. Each (sub) behavior maps the fused ranges to its internal tables and triggers if a match occurs. If triggered, the motor control output of the lower priority behavior is overridden. In the figure the lowest-level, which is the one with the highest priority, is top-most and vice versa.

and soft surfaces confuse sonar rangers. The behaviors are optimized for the Pioneer robot and work out of the box. Nevertheless they are easily adaptable to other robots and sensors and can be combined with behaviors at a higher abstraction level than wall-following (implemented here).

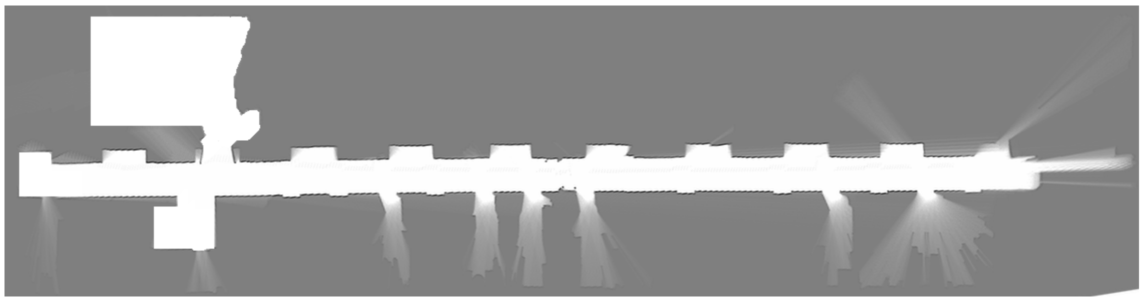
3.7 Integration

In order to create scenarios that work in the environment of the TAMS floor, additional MRS-related configuration must be applied. Each device class described represents a dedicated MRS driver and hardware. Each driver must be setup with parameters optimized for the environment. Driver configurations can be found in appendix A.

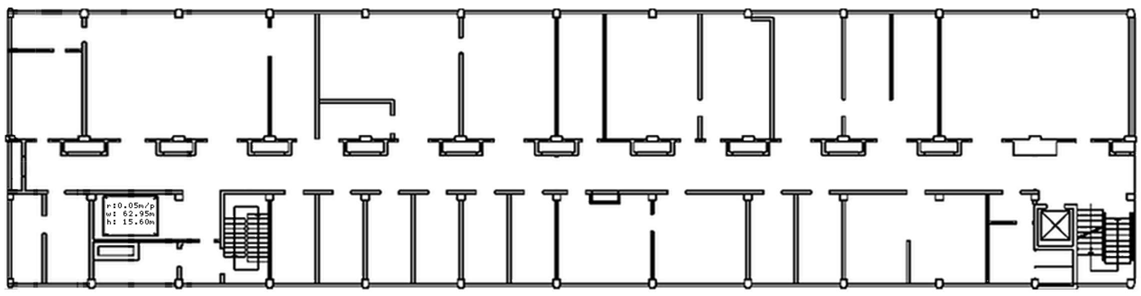
As described in section 3.4.1 the localization component requires a detailed grid map of the environment. Such a map is obtained by recording laser ranger values while the robot explores the TAMS floor. The SLAM algorithm used (*pmaptest*) is included in the Player/Stage MRS. The process stages are depicted in figure 3.24.

3.8 Summary

In this chapter we have defined the goals, constraints and requirements of the intended meta-platform. A new software architecture and the detailed design have been developed, all participating components have been described and the advantages of the implementation have been explained. The result is a software framework providing customized features for higher level multi-robot scenarios.



(a)



(b)

Fig. 3.24: (a) A grid-based map created with laser and odometer data serves as the basis for a map with an accuracy of five centimeters per pixel. The robot recorded its data while driving along the floor. The SLAM algorithm provided by Player/Stage (pmaptest) processed the data and created the map. (b) An inaccurate map, which nevertheless provided the correct topology, was manually mapped to the grid map to obtain an accurate map of the Tams floor. These stages are necessary to provide the localize and path planning components with an accurate map with the right proportions. Otherwise the robot's real position and its position estimate can deviate by up to a few meters or in the worst case localization and path planning can fail totally.

Scenarios

4

In chapter 3 we developed a middle layer to abstract from an MRS and to integrate an MAS. The MAS provides high-level cognitive features that allow the planning of complex scenarios in which multiple robots cooperate in order to reach a common goal. A scenario plan depends strongly upon the specific use-case but should ideally be suitable for re-use. In this work, sample scenarios were created to demonstrate the working concept of an integrated middle layer. These scenarios can be reused in other scenarios.

4.1 Introduction

This chapter describes how the presented middle layer can be used within a meta-platform that also integrates MAS and MRS. A choice of scenarios serves as a framework to describe all necessary software modules, techniques and configurations. In the following section, each scenario is discussed briefly and possible improvements are suggested. Subsequently, migration to another MAS and MRS is introduced before a summary closes the chapter.

4.2 Example Scenarios

This section introduces scenarios that demonstrate the basic features of the meta-platform. The ideas for the *Hunt and Prey* and *Find and Collect* scenarios are taken from the MAS Jadex code examples (comes with Jadex) whereas the *100 Robots* and *Swarm Distribution* scenarios are new. Together, these scenarios demonstrate the basic functionality of the platform and provide a code template for other tasks.

In the *Hunt and Prey* (an extended Hide and Seek) scenario, two or more robots look for and try to catch a hiding robot. The hunting robots coordinate their search to increase efficiency.

In the *Cleaner World* scenario, two or more robots collect trash or specially marked objects, coordinating their search to cover the work area efficiently. As an extension, a constraint could be to conserve robot battery life. An example implementation is

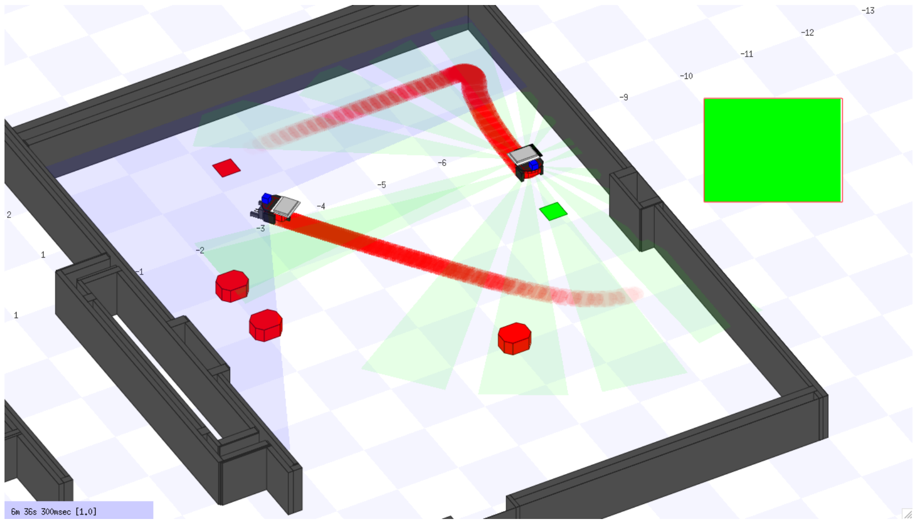


Fig. 4.1: Two mobile robots and the Cleaner World scenario. The robot on the right is equipped with a camera (Blobfinder) to look for certain colors (Blobs) and the robot on the left has a gripper to collect objects detected. The robot exploration and blackboard communication as well as the sensor interpretation algorithms are implemented in Java. The picture shows the scenario in the multi-robot simulator Stage included in the MRS Player/Stage.

shown in figure 4.1. The *100 Robots* scenario consists of 100 robots, each controlled by an agent. Robots perform a collective exploration algorithm, sharing their global positions. This demonstrates the platform scalability.

Finally the *Swarm Distribution* scenario shows how to distribute multiple robots in an efficient manner.

4.2.1 Hunt and Prey Scenario

This scenario serves as an example of basic communication between agents and of hunt and escape behaviors.

A group of prey robots try to escape by avoiding their hunters (if they can sense them). In "idle" mode the prey execute a standard task, which might be marching around, wall-following or another behavior involving motion. The part of prey or escape robots can be arbitrarily extended by implementing more complex behaviors, as described in the *An Advanced Hunt and Prey Scenario* section on page 49. Another way of augmenting the scenario is to let the robots share knowledge about their hunters and thus to develop efficient escape strategies.

The hunter robots have the task of catching their prey by approaching close enough, i.e. the prey is considered to be caught if a single hunter comes within a pre-defined distance. Task complexity can be increased by allowing the hunters to cooperate and share knowledge about their prey. Moreover, a multi-layered behavior model would allow hunters to react to prey behavior or even to forecast prey behavior based upon prior experience.

A learning aspect could also be applied to prey robots, and different learning algorithms could be compared. An extension to the scenario would be to allow prey robots to move into un-blocked space even after the close approach of a hunter.

The demonstration scenario implementation can be run in various configurations. Although only three physical robots are available, there is no limit to the number of virtual robots. As a simulated robot can be equipped with virtual hardware devices similar to those of the physical robot, an agent is not necessarily aware whether the robot it controls is simulated or real. It is therefore possible to let physical and virtual robots interact in a mixed reality environment. This is made possible by a *ViewAgent*, which subscribes to the *SendPositionService* and listens to robot position broadcasts. Knowing robot identifiers and positions, the *ViewAgent* can update a simulation in order to position a proxy robot for each existing physical robot. This use-case can be configured to filter certain robots, as a simulation environment can host both proxy and simulated robots simultaneously. A proxy robot is a simulated placeholder for a real world robot. Using proxies, the simulated and the real environments are normally “identical”. All positional changes of the real robot are reflected onto its proxy and thus to the simulation state. A proxy robot is only updated by a ViewAgent and does not have a control component by itself. In contrast a simulation robot is not controlled by a ViewAgent (although it can have another proxy robot in another simulation). A simulation robot’s actions (and sensing) are limited to the simulation and it acts as if the simulation were real. Proxy robots act and sense in the physical environment and are mapped into the virtual environment where they can be sensed via sensors, such as laser or sonar, by simulation robots. In contrast proxy robot have no possibility of sensing simulated robots via ranger sensors. This is a restriction of the current Player/Stage software and can be worked around by implementing a driver handling virtual-to-real sensory mapping, such as is done in [CMW09] (for the Gazebo simulator). However the meta-platform presented here makes transparent abstract sensing possible between real and virtual robots (and vice versa). This is possible because each robot can be localized (whether simulated or real) and positions can be transmitted between agents (simulated or real).

The use of a mixed reality environment provides certain advantages, in particular by allowing testing with more robots than are physically available. Moreover, it enables the use of unavailable or non-existent devices, allowing novel ideas to be simulated rather than being limited to present reality.

The Hunt and Prey scenario can be demonstrated with either a virtual-only configuration or a real-only configuration as well as with a mixed reality configuration. The environment in either case (simulated and real) does not impose additional constraints on the scenario.

The *EscapeAgent* is derived from the *WallfollowAgent* class and therefore has access to the wall-following behavior of the *Pioneer* robot class (used here), which includes a robust fusion of sonar and laser ranger sensors, as well as effective obstacle-avoidance reactive behavior. The EscapeAgent does not need a planner or localization device because no navigation is required. Instead the *Position2d* device is used to control the robot motors directly. Subscribing to the *SendPositionService* allows the agent to broadcast its position periodically. This simple and effective communication strategy could be enhanced by not letting the Pioneer robot announce its own position information. Instead the hunter robots would detect their prey using an optical device such as a camera and suitable image processing. The present system design allows the easy integration of such an approach. A camera driver would give access to the hardware and an image processing algorithm could be implemented in an additional camera device class. The existing *Blobfinder* class could be also used for its blob-detecting facilities.

The *FollowAgent* derives from the *NavAgent* class and provides navigation support using the *NavRobot* class. This robot is controlled by giving goal positions to the planner device and retrieving updates from the localize device. The necessary global coordinates can be obtained using the *Position* class facilities (section 3.6.2). In the demonstration, the coordinates are retrieved by subscribing to the *SendPositionService*. Using coordinates broadcast by prey robots is not a real world example, since a typical prey would not provide such information. Nevertheless this configuration allows a working system to be tested and demonstrated. In an augmented scenario the coordinates should be retrieved by prey-independent information retrieval such as the camera class mentioned above. This is beyond the scope scope of this work.

When the scenario begins, the escape robot automatically starts wall-following and continuously updates its position using the localize device. The robot periodically broadcasts this position (at the configured interval) using the *SendPositionService*. Rather than just sending, it also listens to position reports broadcast over this service and checks for those near to its position. The distance between hunter and prey below which the prey is considered captured is specified when an agent is created. The hunter agent, represented by the *FollowAgent* class in its task, listens for positions sent by the prey and updates its goal appropriately. This uses the *NavRobot's Planner* device, setting a new goal whenever a new prey position is known. Each hunter listens only to updates from its assigned prey. Prey assignment is currently made at scenario invocation but could be extended. For example, the agent could hunt its nearest prey (see the *An Advanced Hunt and Prey Scenario* section). The hunter measures the distance to the prey using its *Localize* device. Both, prey and hunter, follow their primary task (wall-following and hunting, respectively) unless the given minimum threshold is triggered. In this case the hunter knows that the prey is trapped in an unescapable situation by its distance. The prey recognizes that no escape is possible and indicates this by spinning. The scenario sequence is

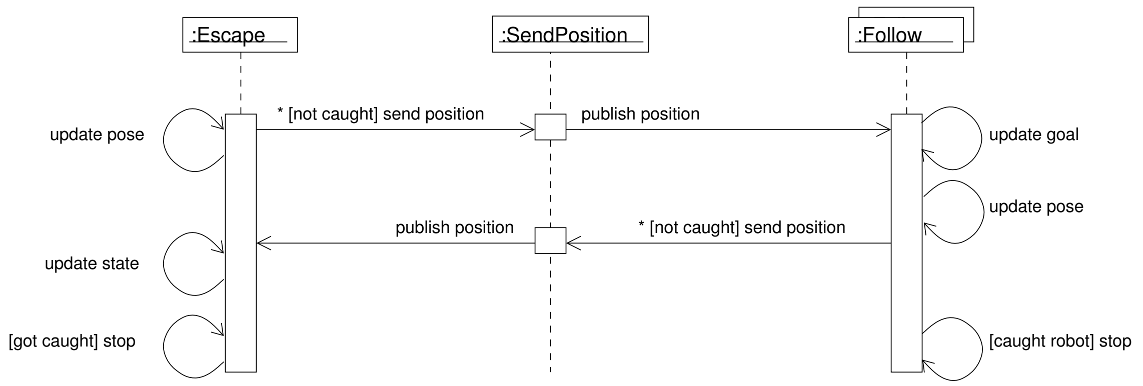


Fig. 4.2: Here, a simplified Hunt and Prey sequence is presented. The initialization and registration phases have been omitted. Both the *Escape* and *Follow* agents have subscribed to the *SendPosition* service in order to exchange information. Initially, all agents repeatedly update their position using the robot's localization device. Periodically all agents publish their positions to the SendPosition service, which broadcasts to all subscribed participants. The Follow agents update their goals according to newly received data. As long as the Follow agents have not been triggered by the targeted minimum distance to the Escape agent and vice-versa, the scenario continues.

indicated in figure 4.2.

This scenario intends to show basic interaction in the completion of a cooperative task use-case with mobile robots. It is not intended as a comprehensive example of the widely used Hunt and Prey robot scenario but demonstrates the implementation of such a scenario with the (meta-) platform of this work. Furthermore a proposal for an advanced Hunt and Prey scenario will be given. The view of planning activity during the scenario is given in figure 4.3. A real-world scenario run mapped into the virtual environment is shown in figure 4.4.

An Advanced Hunt and Prey Scenario

The hunt and prey scenarios presented above can be improved by exploiting the Jadex BDI model. The BDI model describes the agents cognitive system. There are goals, plans and sub-plans. The prey agent's main goal would be exploration; a second goal might be gathering beneficial objects such as energy stations to regain battery life. This imposes a multi-layered behavior gained through agent intelligence instead of the subsumption architecture presented (section 3.6.2.4 on page 42). The behavior BDI model [PBL05] could also implement other behaviors, such as dedicated escape strategies, which would be triggered by certain relative hunter positions (in other words, when hunters are close).

On the hunter side, improved cooperation and communication can be applied. In order to optimize hunting efficiency (time to catch prey), a strategy based upon

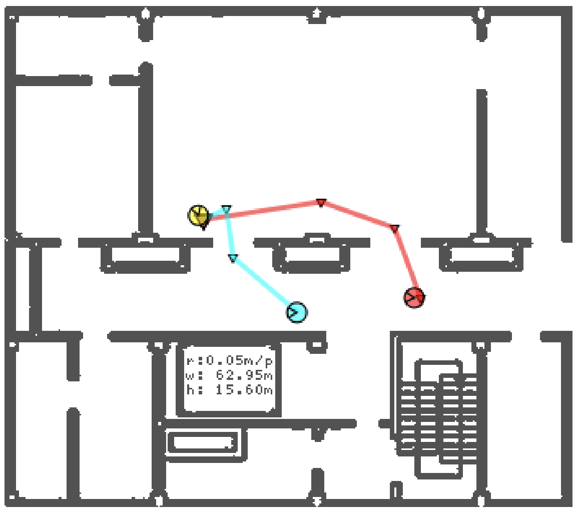


Fig. 4.3: Two robots (the blue and red circles) are collectively hunting one prey robot (yellow). Each robot follows its optimal path to the goal. The colored lines indicate the planned trajectory. The small triangles on the trajectory mark temporary waypoints for the local path planner. The global trajectory goal will be updated according to the prey robot's varying position. The robot's current heading is indicated by the small triangle within the circle.

prey position relative to the hunter group formation can be applied. The hunters can decide which team members chase which prey. This approach can increase the number of prey that can be hunted in parallel. It can also save resources by directing hunters to the nearest prey. This could be achieved by implementing the FIPA-Contract-Net protocol [Cer08], in which agents agree prey attribution depending upon relative distances.

4.2.2 Find and Collect Scenario

The Find and Collect scenario was inspired by the Jadex Cleaner World example and the trash collecting robots in [MAC97]. The Cleaner World example can be considered a similar scenario. Two groups of robots participate: collecting robots and exploration robots.

The collecting robots fetch objects with their attached gripper and bring them back to their home position. A home position is defined for simplicity as the initial start position of the robot and can be defined anywhere within the map upon agent creation. An object can be any movable, physical item in the robot's environment that the robot can move (for example, with its gripper). In the scenario implemented, an object is represented by a little square blob on the floor, a little smaller than the footprint of the robot itself.

The exploration robots explore unknown territory and can detect interesting ob-

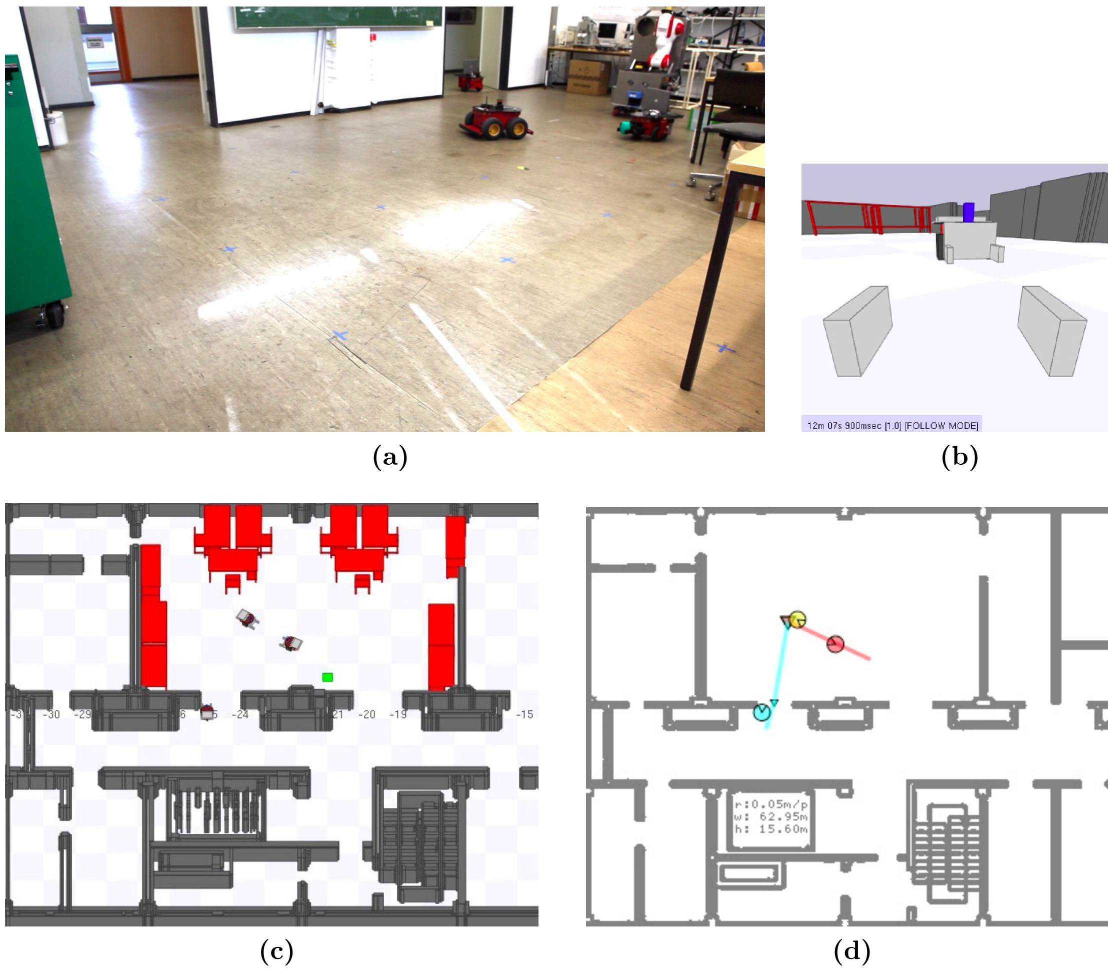


Fig. 4.4: A snapshot during the Hunt and Prey Scenario. Four perspectives of the same situation are shown. The (start) configuration of the real-world scenario is as follows. Two Pioneer robots (3AT and 2DX) are hunting a Pioneer-2DX escape robot carrying a green bin between the gripper paddles. The moment depicted in the pictures shows the event of the escaper being caught by the fastest hunter, the 3AT model. Although only real robots take part, the scenario is also mapped into the simulation environment. (a) The laboratory camera shows a total scene of all three participants (in the top-right area). The slower hunter is entering one of the two doors to the TAMS laboratory. (Note that the camera position is in the top-right corner of the laboratory on the map) (b) The first person view of the escaper in the virtual environment. The hunter robot (with a blue laser model on top and a gray gripper model at its front) can be seen approaching. Note the that the paddles in the foreground belong to the robot from which the scene is viewed. (c) This overview of the scene, created by the (Stage) simulation, is continuously updated. Additional simulated robots could be added on-the-fly. (d) The planner view (*Playernav*) shows the hunters locked on the escape robot.

jects by their color. In the virtual environment this can be implemented using a blobfinder device representing a real world camera capable of detecting object colors or shapes. In the scenario implemented, a wall-following exploration algorithm is used, but any suitable algorithm can be used. The exploration robot's main goal is to keep exploring the environment while reporting newly discovered blobs via the *ReceiveNewGoalService*. The collecting robot listens to this service and stores new blobs in local memory. Both participants use the blackboard implementation to store their blob information.

The goal of this scenario is to demonstrate communication and cooperation between two distinct groups of robots. A mixed reality run of the scenario is shown in figure 4.5 and 4.6.

An Advanced Find and Collect Scenario

An enhanced scenario could be implemented by augmenting the number of participants of the individual robot groups, by adding multiple and more sophisticated behaviors for both groups and by implementing a standardized FIPA Agent Communication Languages (ACL). The latter approach could benefit from the use of the FIPA-Request-Interaction or Contract-Net protocol. The Request-Interaction protocol can be used to request support from a nearby collecting robot to collect detected objects; the Contract-Net protocol can be used by collecting robots to negotiate between themselves, and to agree which is closest to the requesting exploration robot.

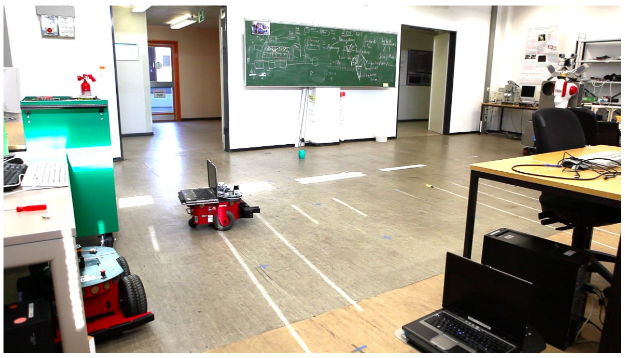
This scenario could also be adapted for a distributed sensory network use-case in which multiple wall- or ceiling-sensors watch the environment and trigger idle collectors when an appropriate object is detected.

4.2.3 Swarm Scenario

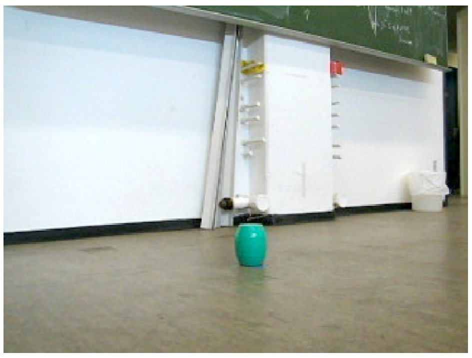
This scenario demonstrates the flexibility and scalability of the meta-platform presented. It includes 100 robots that spread out from their closely spaced starting positions and explore the map by wall-following. Low-level obstacle avoidance guides each robot to avoid collisions.

In order to stress the MAS layer, each robot subscribes to the *SendPositionService,* broadcasts its own position and reads other position values. This results in one hundred simultaneous transmissions and receptions within a fraction of a second.

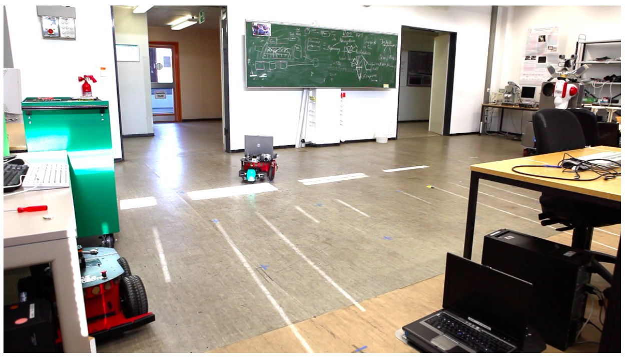
This scenario also stresses the MRS layer. Each robot integrates a planner and a localize device running the AMCL, Wavefront and VFH drivers in the robot's context. This also shows that the localization driver supports not only the laser ranger but also the sonar ranger sensor in combination with the odometer. A sample of the swarm scenario is depicted in figure 4.7.



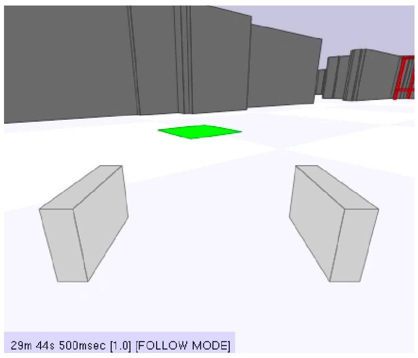
(a)



(b)



(c)



(d)

Fig. 4.5: Several snapshots during the Find and Collect scenario. Four perspectives of situations during the scenario are shown. The (start) configuration of the mixed reality scenario is as follows. A virtual Pioneer robot carrying a blobfinder device (a virtual camera) searching for green blobs (objects) in the (virtual) environment. The presented wall-following behavior serves as the exploration algorithm. In order to allow real-world interaction, a green bin is placed at the exact blob position in the TAMS laboratory. A real robot, the Pioneer-2DX with a gripper attached, waits for blob position targets in order to approach, grasp the real bin and bring it back to its start position. The simulation environment is a necessary part of the scenario. (a) The active collector robot (center) waits for new targets. (b) The green bin target seen from the perspective of the approaching collector. (c) The collector has grasped the target object and returns home. (d) The same perspective as in *(b)* but in the virtual environment.

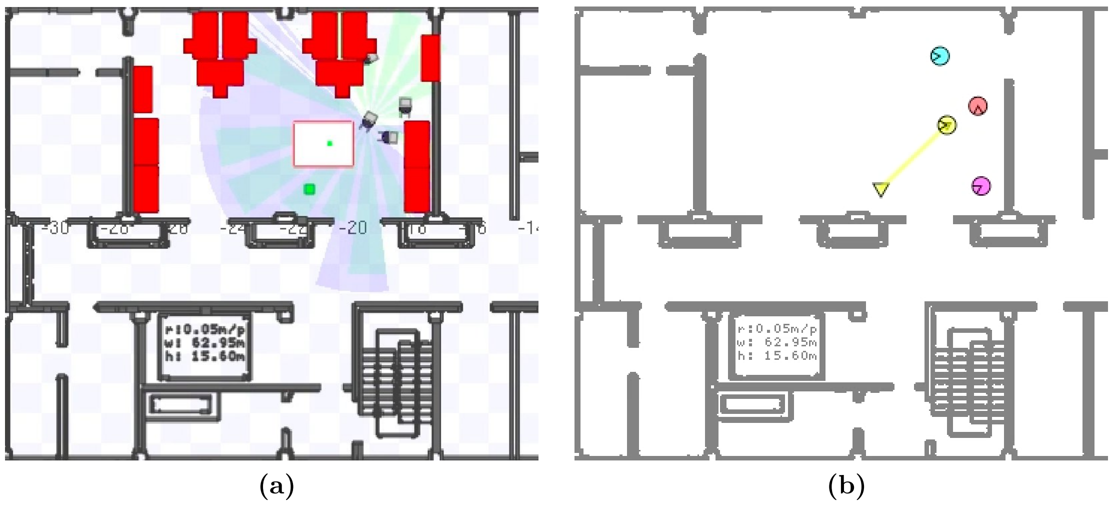


Fig. 4.6: Another event within the same run of the Find and Collect scenario depicted in figure 4.5. (a) Virtual environment view. The event of the searching robot detecting a green blob within its field of view (white square in center) is depicted. Communication regarding the newly found target is triggered. (b) The collector robot listens for such targets and has planned a path to the target, shown here in the planner view.

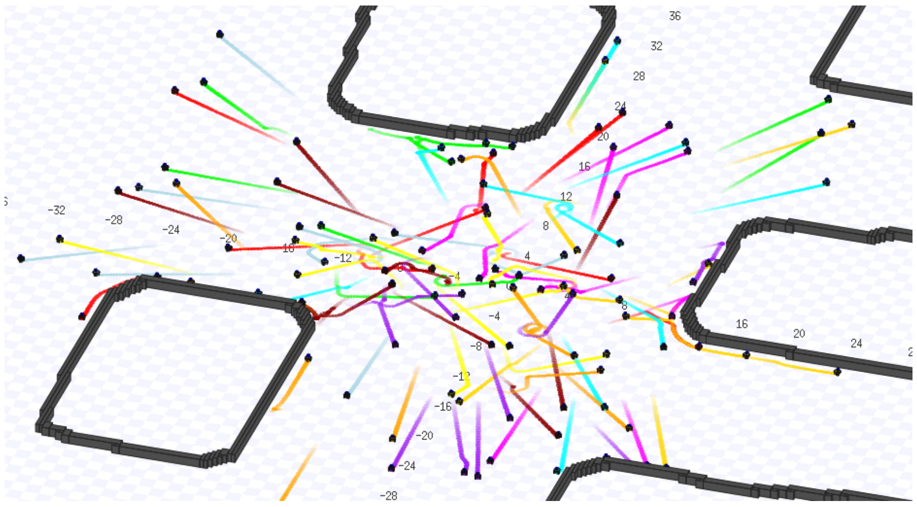


Fig. 4.7: The Swarm Scenario consists of 100 Pioneer robots and agents. Each robot has sixteen sonar ranger sensors to perceive its environment and includes particle localization and Wavefront path planning algorithms. The robots apply a wall-following subsumption behavior model. Each robot is controlled by a Jadex agent which also propagates the robot's current location to the other agents. The robots start close together in a crowd from which the implemented behavior distributes them towards a wall. Here different colors indicate the robots' recent trajectories. The same color means belonging to the same group. In order to create space for the robots a map of 128m x 128m has been created, derived from Player/Stage's *simple.world*.

An Advanced Swarm Scenario

The Swarm scenario is well suited to the exploration of any behavioral model that consists of multiple, less sophisticated, individual behaviors. An example would be research into formations. Another would be research into task allocation and sub-allocation for teams of robots. These teams can be grouped together for individual (sub-) tasks. Again the BDI model can be applied to divide goals into sub-goals and to assign goals to an appropriate team.

4.2.4 Distribution Scenario

Rather than being a standalone scenario, this is typically part of another, such as Hunt and Prey or Find and Collect. It demonstrates the use of a central planning agent controlling the available agents. The purpose is to dynamically distribute a group of robots across the traversable environment. This can be helpful in cases where robots need to be well distributed over the map in order to process a scenario properly.

The Distribution Scenario adds a new agent class, the *DispersionAgent*. Other participating agents, such as navigating agents, are required to subscribe to the *HelloService* and the *ReceiveNewGoalService* in order to communicate with the DispersionAgent, in particular to be assigned distribution goals. The former service is used to request all available agents to identify themselves (with a unique identifier and agent information) in order to be classified (such as *NavAgent),* while the latter is used to send new goals to agents (a goal instructs an agent to guide a robot to a given position).

The central planner algorithm is based upon a set of predefined positions within the environment map. These positions are currently manually defined for optimal spatial distribution but can be later be found automatically. Positions on the TAMS floor are chosen to ensure a minimal degree of spatial separation of physical agents. A sample assignment of initial robot positions is presented in figure 4.8. The central planning agent, upon invocation, triggers a reply from all other agents by using the SendPositionService to "ping" all subscribers. Each receiver reacts with a confirmation message containing the requested information. All such responses are parsed and added to the planner's local memory (using the blackboard class). In order to structure the participants taking part in the conversation, each note on the blackboard is given a *topic* field, for which the class of the responding agent, contained in the reply, is used. The advantage is the possibility of reacting individually to each agent according to its type (class). After collecting and structuring the participants they are listed in arbitrary order. The list is scanned and each entry is assigned a position. The current allocation is made in order to select the nearest available position in the remaining set of selected distribution points. Finally each agent is sent its target position via the *ReceiveNewGoalService*.

The advantage of this implementation is its flexible handling of available agents and robots. Robots not reacting, such as those not supporting the proper services or malfunctioning robots, will be ignored. This scenario can be combined with other scenarios that benefit from spatial distribution of participating robots and can manage both virtual and real-world robots simultaneously, as they are handled transparently. A real-world run of the scenario is shown in figure 4.9.

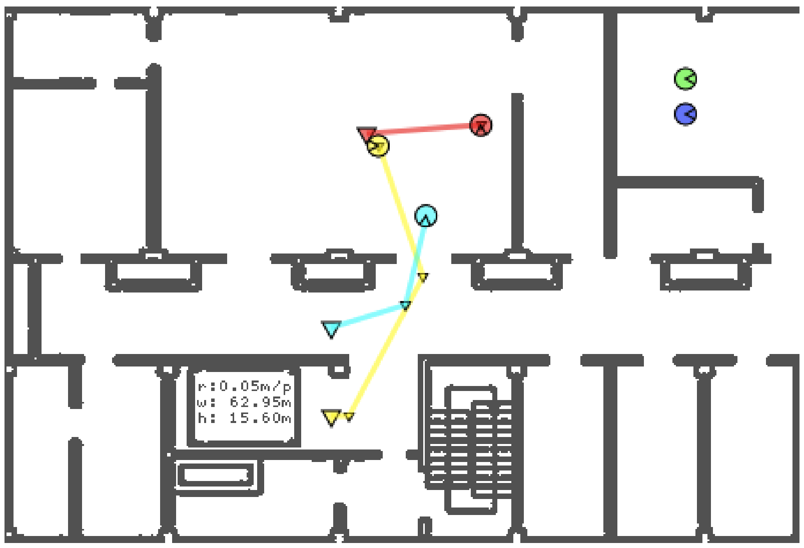
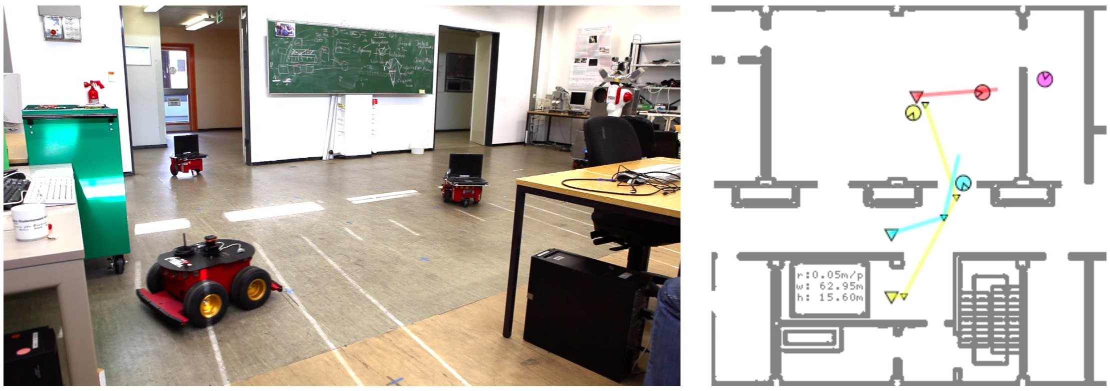


Fig. 4.8: A dedicated agent pings all available robots on the network, computes optimal distribution points on the TAMS floor map and sends individual goals to each robot. Here three robots are currently active and responsive. Two robots (on the right top) are not participating as they are assigned to different tasks.

An Advanced Distribution Scenario

The present design depends on a pre-defined set of strategic positions on a pre-defined map. For added flexibility, a dynamic approach to defining strategic points within arbitrary maps can be implemented, with map information parsed at runtime. The retrieved map information can then be used in order to calculate best-suited positions according to specific needs.

In the distribution case, a spatial distribution with equal, maximized distances is wanted. For such use-cases an algorithm must find points on the map that are mutually reachable and not too close to a wall. The algorithm can reach a high level of sophistication as the environment becomes more complex, such as the TAMS floor with its single narrow floor and its many rooms, the doors of which are likely to be closed. The long floor itself is not complex but would lead to a distribution of points along a straight line, when considered alone. The algorithm must also target rooms



(a) (b)

Fig. 4.9: A snapshot during the Distribution scenario. Two perspectives of the same situation are shown. The (start) configuration of the real-world scenario is as follows. A group of three Pioneer robots (one 3AT and and two 2DX) are distributed amongst the rooms of the TAMS laboratory, here one to a room. The moment depicted in the pictures shows the point at which all robots have been assigned a target according to responses by active robots to the DistributionAgent. Although only real robots take part, this scenario is also mapped to the simulation environment. Virtual robots could be added on-the-fly. (a) The three robots follow their path to their target positions. (b) The planner view depicts the target position assigned to each (active) robot and their intermediate goals.

that are reachable through doors that are wide enough (for the robot to pass), such as the TAMS laboratory, in order to use the territory efficiently.

4.3 Mixed Reality

Some of the scenarios presented in this chapter run in a mixed reality environment. The difference between this and purely real environments is as follows. In a purely real environment robots can sense only physical objects present in their environment. In contrast, in an augmented real environment physical robots interact with virtual objects, such as simulated robots.

The Hunt and Prey scenario can be run in either real-only, virtual-only or mixed reality configuration. For the MAS and RSAL layer components this distinction is not present as device access is encapsulated transparently within the MRS. One exception might be that a robot can be aware of a simulation environment when passing an appropriate simulation device to it (section 3.6.2.2 on page 35). Thus it is possible that real robots hunt a virtual robot, virtual hunters follow a real prey or a team consists of both types. In either case there is interaction between virtual and real agents/robots. In the Find and Collect scenario interaction in the mixed reality configuration has been demonstrated.

A big advantage in such a configuration is the possibility of simulating (robot) hardware that is not present in reality, for example where the required number of robots (section 4.2.3) or the required device type (section 4.2.2) is not available. Another advantage is the abstraction from hardware problems. Often an algorithm has been designed and needs to be tested. Typically a simulation environment is chosen for initial testing. Later on, the algorithm is tested on the real hardware. In the latter step, unexpected difficulties often arise as hardware introduces other possible errors. In preliminary work (section 2.4) a wall-following algorithm performed well in the simulation and performed poor in reality (in early development state). In order to focus the research effort on the main goal such problems can be avoided by running a stable mixed reality environment having only components in reality that are mature enough. A step-wise approach to migrating each component to reality is transparently possible. Research can focus on the global problem instead of dealing with low-level (hardware) issues.

4.4 Adoption of another MAS

In order to replace the current MAS with another, the system design allows for seamless migration. Only the top MAS layer within the three layer approach must be changed, with changes to the agent implementation and the agent services. As it is unlikely that another MAS supports currently implemented agents, agent bodies must be adapted in order to conform to the new system. Furthermore the interaction between the agents is crucial to the whole platform and has to be adapted as well. If the new MAS supports the Java language, migration should be straightforward. For a different language the JNI must be used to provide a native language interface to the Java middle layer. This JNI wrapper must be provided by the user.

4.5 Adoption of another MRS

The replacement of the MRS affects the bottom layer of the three layer system. Special attention must be focussed on the driver components. The use of ROS as the new MRS would lead to minor changes in this area. Since most drivers from Player/Stage are also supported and integrated in ROS, no major changes are necessary. To use the same driver configurations as before, the configurations must be converted between Player/Stage and ROS (XML) formats.

As components of the middle layer rely upon the MRS interface, these components have to be adapted. The RSAL classes affected are those derived from Device, except for the Robot class. Within these classes the MRS calls are encapsulated and must be replaced by the appropriate ones from the new system. The concept of a DeviceNode is abstract and typically does not need adaption, although it might have less significance; in Player/Stage, devices are bound to a specific node (a PlayerClient), but this need not be so in other MRS. In systems that are completely peer-to-peer based, such as ROS, the grouping of devices might be optional and arbitrary.

4.6 Summary

In this chapter, selected cooperative scenarios are described. The chapter is based on common scenarios in robotic research. Here they are integrated into the meta-platform and demonstrate its maturity. Finally a discussion about the applied mixed reality mode is followed by a description of the process of migration to another MAS or MRS.

Conclusion 5

The meta-platform developed here provides multi-agent facilities to a group of mobile robots. These robots can interoperate within a mixed reality environment and the platform allows efficient integration of new robots and behaviors. How does this approach perform in an objective evaluation?

5.1 Introduction

This chapter evaluates the meta-platform and key aspects of the platform will be highlighted and discussed. General enhancements that can, in the future, improve the implementation are mentioned. Finally an outlook to possible use-cases of the presented work is given.

5.2 Evaluation

In this section the demonstration example is evaluated according to various criteria, namely the technology used, the scenarios demonstrated, the user interface and the performance. Both initial requirements and new features are discussed.

5.2.1 Technology

The technologies used include recent developments in the areas of multi-agent systems (Jadex) and multi-robot systems (Player/Stage). Jadex is a proven, reliable and efficient system for research and industrial purposes[1]. The Player/Stage project has been used by many problem-solving activities in robotics world-wide[2]. Based upon these major systems, the implemented abstraction layer (RSAL) benefits from stability and flexibility. RSAL is implemented in Java, which supports state-of-the-art object-oriented software development. In particular, the RSAL implementation

[1]http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/Usages/Projects
[2]http://playerstage.sourceforge.net/wiki/PlayerUsers (July 11, 2011)

is heavily threaded: each robot and device runs in its own thread, handles synchronization and provides an asynchronous interface. This approach dynamically de-couples platform components and also benefits from current and future multi-core computers.

In addition to the major components described above, other implementation aspects are noteworthy:

- Current classes for real-world robots, devices and agents facilitate software re-use within extended or alternative scenarios;
- The callback mechanism in use provides for easy, event-driven, access to RSAL services by any client;
- The blackboard pattern allows easy object storage and retrieval and provides basic features for a robot-learning architecture. The lightweight implementation does not degrade performance, allowing components, such as agents, to have their own "memory" in addition to memory distributed across a network of participants. This could be achieved by a dedicated central agent, without physical representation, concentrating on message exchange between agents, such as the MasterAgent and DispersionAgent (section 3.6.1).

The use of different robot types, or even of similar robots with different device configurations, often introduces a high platform configuration effort. The presented implementation allows for automatic robot device detection. This feature focusses on a tree-node approach where devices are connected to a root node, the *DeviceNode*. Dynamic retrieval of currently active devices is provided and includes the recognition of devices that malfunction and can no longer be used. This approach adds a layer of abstraction above the robot hardware and allows agent design to focus on cognitive features.

The mixed reality configuration presented has been implemented using pre-existing components of the platform. These include a device handling a simulation interface and an agent that listens on active services to retrieve the information needed to update the simulation. This feature benefits from software reuse and provides a completely new facility. The mixed reality approach is currently an active field of robot research. It provides facilities for a graphical user interface, enabling it to display virtual and real robots simultaneously within a simulation environment. The approach can be extended.

5.2.2 Scenarios

The scenarios presented in chapter 4 provide an overview of basic platform features. All scenarios runnable with real robots such as Hunt and Prey, or Find and Collect, demonstrate successful integration of the robot hardware in use. Navigation, as implemented, allowed accurate localization within the TAMS laboratory and the whole floor. The planner worked well with the configuration and hardware. Additional devices such as a gripper and robot-mounted laser ranger finder were integrated successfully.

The integration of the MAS demonstrated the seamless coexistence of MAS and MRS platforms. All features of the agent system worked successfully.

The Hunt and Prey scenario shows inter-agent communication as well as basic cooperation. Interaction happens in a dynamic environment applying multiple, state-of-the-art, robotic features such as wall-following, sensor fusion and a subsumption behavior architecture. This software performs well, is robust in the case of hardware errors (such as erroneous laser readings and hardware defects), and successfully handles different hardware configurations.

Whereas the Hunt and Prey scenario focuses on a simple behavioral model of participating agents, in which the behavior of each agent/robot does not change during the scenario, the Find and Collect scenario implements a two-step interaction. The collector agents are triggered only upon receipt, via the agent services, of information describing an “interesting” object. Both finders and collectors use local memory that implements the blackboard model, thus allowing them to represent an autonomous model of the real or simulated world.

The Swarm scenario demonstrates the scalability and stability of the platform on a single host. Each robot consists of an agent component that communicates via services, a robot model that uses its own devices and finally the localization and navigation component of the MRS. This single-robot model is replicated 100 times. In addition, this scenario provides a starting point from which to explore and implement dedicated robot swarm behaviors.

5.2.3 User Interface

The user interface provides a multi-window and component focused surface. The Jadex Control Center provides a well arranged user interface to start/stop agents and scenarios. Runtime parameters can be changed and passed to agents and sophisticated agent debugging features can be applied. An optional Java console supports information display and error localization. This is the main interface, as the user typically interacts at agent/scenario level. Each MAS typically provides its own, optimized interface.

For a more robot-centric view of a scenario, an MRS typically provides a variety of tools. For watching current robot localizations and planned trajectories, Player/Stage provides the *Playernav* utility. This provides an easy overview of all participating robots and their goals. Other MRS, such as ROS, provide their own tools for visualization, such as the *RVIZ* utility.

When working with simulated environments instead of real ones or when applying a mixed world scenario, the Stage simulator provides a convenient, almost three-dimensional, interface and allows robot data to be visualized. ROS also integrates the Stage simulation.

5.2.4 Performance

Overall performance depends upon the applied scenario. In general, total processing delays are the sum of delays in the MAS, RSAL and MRS layers. The threaded design of the RSAL and the flat, optimized, call hierarchy are apparent in the low overall impact of the RSAL on total delays. MAS and MRS delays vary according to the current state of a scenario. The MAS can produce significant startup delays. However, by creating many services when starting, it can reduce its impact on the overall delays within a running scenario. The major impact comes from localization components: calculation of initial pose estimates at startup takes a long time, as no previous positions are known (unless positions are exactly predefined) and many particles have to be processed by the localization component. With the currently implemented and optimized configuration, this delay is minimized. Moreover the delay is not recognizable during the main scenario processing. The AMCL algorithm used needs very few resources once it has a position estimate, as further updates take the previous position into account. Nevertheless, the localization and navigation components limit overall system responsiveness when many robots operate simultaneously, such as in the swarm scenario.

5.3 General Remarks

When multiple agents cooperate to solve a task, several key issues have to be considered. The meta-platform provides the basic infrastructure for nearly any task-solving strategy and there are usually many possible configurations, each with advantages and disadvantages. A suitable system configuration is the key to the effective solution of a given problem. The key issues to be considered are highlighted in [Bek05] and are discussed below.

When approaching a task, one must consider the jobs each robot has to handle. Can all sub-tasks be managed by the same robot type or are robots with differing capabilities required? The answers to this question constrain the scenario configuration. If all robots are the same, they can be seen transparently, as the physical representation of an agent that can manage any task. In this case, the selection of a robot for a task depends only upon robot location. In contrast, where robots of differing capabilities are used, the selection of a robot for a task depends upon both robot type and location. Heterogeneous robot teams require additional interaction and communication effort, which has to be considered in the scenario design.

Homogeneous robot teams require coordination, but are more flexible in terms of the specific robot assigned a task. In the scenarios presented, this differentiation can be understood. The Hunt and Prey and the Swarm scenario are based upon homogenous robots, whereas the Find and Collect scenario directs a heterogeneous group consisting of explorer and collector robots with differing hardware that notably changes their use and behavior. Whereas in the homogenous cases the role of each participant can be chosen arbitrarily, this is not true in the heterogeneous case. Here the role of each robot type is distinct in terms of its basic usage. The explorer has a blob-detecting device and no effector to collect anything. Therefore it has to interact with a supporting collector robot that does not have an optical detection device but instead has a gripper, allowing it to grip, move and release objects. The two groups need each other to complete the task of searching and collecting objects in their environment. The implemented scenario handles this by interactively triggering events on the appropriate agent communication channels (services).

Another key aspect of cooperative task solving is the control of distributed robots. Whereas centralized control over multiple participants allows efficient and redundancy-free management, distributed or autonomous control of each robot introduces another level of robustness. The central control approach implements a dedicated planner that communicates with scenario participants in a peer-to-peer manner. Information retrieval in this case is simple for the planner, as it processes all data anyway. It can collect data and make decisions from experience in order to control other robots. This design can be exploited for complex scenarios where a lot of information from different sources must be processed in order to make reasonable and time-critical decisions. In contrast, a decentralized design introduces more responsibilities to individual robots. Each robot has its own goal, resources and plans. Coordination is achieved by each robot solving its individual task. This approach requires less complexity from each individual participant than a central planner, which reduces the risk of design or implementation failures. Nevertheless this comes at the cost of adjusting each component in order to integrate a team for cooperative task solving. The implementation introduces a central planner, such as the distribution agent, which triggers and receives robot data and controls their formation according to the number of robots and their positions. A practical scenario would most likely benefit from a mixed control design.

When formations are required for a task, the design has to focus on the question of loosely or tight-coupled robot teams. Although in the presented scenarios almost no relative robot positions were important, in other scenarios, robots might operate as a group and therefore have to dynamically coordinate relative positions.

Environmental and task requirements can introduce constraints on communication links. Whereas in some scenarios it is possible to share all robot information, for example by broadcasting, in other scenarios such sharing is not allowed or is impossible because of the environment. Good communication should prefer information-hiding to polluting the network with unwanted or unimportant data. Nevertheless information should be available to all receivers that need it.

Another key aspect of such a meta-platform is the human-robot interface, or in other words, task assignment. How is a task assigned to a group of robots? This topic is related to control design: where a central planner is available, it can accept a task, process it and delegate sub-tasks to participating robots. If no such central process is available, the task has to be subdivided in advance in order to delegate sub-tasks directly.

Finally certain environmental constraints, such as indoor or outdoor territories, introduce special system designs. Furthermore, learning requires its own cognitive components and must be handled at a more abstract level of interaction, as for single robots.

5.4 Practical Usage

The meta-platform presented here provides all services to support scenario development in the field of robotics. The following paragraphs introduce some suitable application areas.

In the research field of search and rescue robots, interaction becomes mandatory as typical scenarios are too complex to be accomplished by a single robot. A search operation benefits from the number of search participants. The deployment area can be covered in less time with an increased number of robots. In order to efficiently cover the environment without re-discovering the same area it is important to coordinate the robots' targets dynamically.

Swarm formation, a recent research topic, benefits from this work by applying dedicated formation algorithms to the system.

The scenarios presented are based on a pre-defined static map that is available to planner components. Another scenario would be to explore unknown territory using a team of autonomous robots. Present state-of-the-art SLAM algorithms can robustly map a territory using a single robot system. For a distributed SLAM, new algorithms can be designed and implemented in the meta-platform in order to exploit its communication and data memory facilities and to benefit from a multi-robot team.

Communication between a group of agents or robots is another field of current research, one in which efficiency, scalability and flexibility are important topics. The currently implemented service infrastructure can be enhanced in order to explore these issues.

The important service-robot use-case for a foraging population can also take advantage of such a highly interactive platform. Different kinds of household robot might share information on activities to be performed and on environmental changes, or could pass operator requests to the best suited robot. Nevertheless, such complex scenarios can be combined with distributed sensor networks to support data retrieval by small, fixed sensors (both dedicated or robot-attached) in the domestic environment, such as refrigerators and washing machines.

A practical environment for the Find and Collect scenario would be that of a production line. Unused or rejected parts, as well as trash, are created during manufacturing. Such parts could be spotted by mobile or fixed sensors, such as cameras. Mobile collectors would be given the spot positions and would autonomously find their way to the target, grasp it and bring it to a disposal position before returning to their idle task.

Often certain facilities have to be under permanent surveillance, such as museums, company buildings and military territories. Such a task can be solved by multiple cameras watching the important area. The field of view can be augmented by various mobile robots that periodically observe covered territory. In the case of an emergency event, appropriate actions can be triggered, such as sending mobile robots to the location of the problem. Robots following surveillance routes through indoor or outdoor environments could maintain their battery charge state and servicing autonomously, for example by heading to a recharging point when necessary.

5.5 Summary

This work deals with a generic software platform integrating multi-agent technology and a multi-robot system. It describes the use of the platform for typical cooperative robotic scenarios. One Pioneer-3AT and two Pioneer-2DX from MobileRobots Inc. serve as the hardware base for real-world applications. The robots have sonar and laser ranger sensors attached and some have grippers. With the ranger devices, the robots are able to locate their position on a static map in an indoor environment. The gripper is used for object manipulation. The multi-agent system, Jadex, is integrated for cognitive and task-distribution services. Furthermore, the multi-robot system Player/Stage is used to interact with the robot hardware and to provide a simulation environment.

The development of the software is described in figure 3.5. The robot navigation stacks were configured to reach high localization and path planning accuracy within an indoor environment. For this purpose, a proportional and accurate grid-map of the territory was created by a particle-filter-based SLAM algorithm. The requirements for a generic system integrating MAS and MRS lead to the design of a generic middle layer. Implementation and testing use current software design patterns and focus on run-time efficiency. The middle layer consists of a flexible concept for robots, behaviors and devices independent of the specific MAS and MRS in use. Moreover agents, communication services and scenarios have been implemented and tested. A mixed reality graphical user interface allows simulated and augmented-reality scenarios. The efficiency of the overall system is scalable, allowing for control of a large number of robots simultaneously.

A multi-robot platform with multi-agent technology based on Player/Stage and Jadex has been introduced. Core components can be used with other systems of the same kind, such as other MAS and MRS. Two real-world scenarios and one virtual scenario served as the basis for the realization of interactive collaboration. A concluding discussion of current and future research into various topics related to the meta-platform has been presented.

5.6 Outlook

This section discusses possible improvements to the presented work.

In order to augment the currently supported scenarios, additional robot behavior can be added. In particular, this might involve higher-level behaviors such as wall-following or exploration. The low-level behaviors, such as obstacle avoidance, provide a generic module that can be used with other behaviors. A completely object-oriented behavioral model that allows new behavioral components to be added or removed would improve the flexibility and maintainability of the system. Furthermore, advanced swarm scenarios would give the system the ability to perform large area exploration and communication.

In the present implementation, the robot system consisting of the MRS and the central MAS are invoked manually. For a more dynamic approach, central and local systems can be started automatically upon start-up of the robot hardware. Exploiting the Jadex *Awareness* feature, which allows automatic detection of agents present in the network, would enable autonomous agent invocation. When invoked, a default behavior such as wall-following or system monitoring, could enable more rapid reaction to environmental changes. It could also trigger appropriate tasks on the discovery of interesting objects.

The use of standardized, mature and reliable communication protocols, such as the FIPA-Contract-Net, could make maintainability and debugging more transparent.

Task assignment is currently encapsulated within pre-defined scenarios. Accordingly, a specific task-solving activity is defined by the choice of dedicated agents that are known to be able to contribute to the task solution. Another approach to task assignment is not to define how the task is to be solved but rather to let the (meta-) platform decide. The task description can then be parsed, automatically divided into sub-tasks and then assigned to individual robots. The assignment process is influenced by the agents currently available. Considering the Find and Collect scenario, it could be specified that certain objects have to be collected and brought to a specific position, but not which robot or how many robots take part in solving the task. Another example in the Hunt and Prey scenario would be to define the task of hunting a certain group of prey robots instead of assigning each robot a task.

A straightforward enhancement of the current system would be the exploitation of the BDI cognitive model using Jadex BDI agents instead of the micro agent implementation. This approach comes with an external definition of agent goals, with plans and sub-plans to solve. It gives a abstract view of the agent goal definition and allows complex and long term targets to be defined.

Acknowledgements

This thesis would not have been possible without all the people who supported and helped me.

I especially want to thank all employees of the TAMS group who supported me by answering all my questions and supplied the needed equipment. Particular I am grateful to Dipl.-Inf. Denis Klimentjew for providing me with uncountable ideas, hints and the support of my master study. Furthermore I like to thank the VSIS group for assisting me, namely Dr. Alexander Pokahr and Dipl.-Inf. Dirk Bade for their time and help.

I would like to show my gratitude to my supervisor Prof. Dr. Jianwei Zhang for the assignment of this work and his confidence in me and the permit to use the laboratory equipment.

Beside my supervisor, I would like to thank my secondary reviewer Dr. Lars Braubach for the takeover of the second opinion and the given guidance.

My sincere thanks also goes to Dr. Andreas Mäder for the installation and assistance of hardware and software equipment and to Tatjana Tetsis for handling all the organizational things as well as her always happy mood.

The following private people I like to thank in particular.

For his endless efforts to review this thesis and giving me his professional advices, I owe my deepest gratitude to my friend William Morris, B.Eng. Hons.

I would like to thank my partner Nicole, who was a steady guide, for her inspiration and patience not only during the time of this work but also during the past years.

Last but not the least, it is a pleasure to thank my family: my mother Roswitha and Frank as well as my sister Julia for supporting me in numerous ways during my life.

Bibliography

[AKBF10] M. Ahmed, M.R. Khan, M.M. Billah, and S. Farhana. A collaborative navigation algorithm for multi-agent robots in autonomous reconnaissance mission. In *Computer and Communication Engineering (ICCCE), 2010 International Conference on*, pages 1 –6, May 2010.

[BBC+95] Tucker Balch, Gary Boone, Tom Collins, Harold Forbes, Doug MacKenzie, and Juan Carlos Santamaria. Io, ganymede and callisto – a multiagent robot trash-collecting team. *AI MAGAZINE*, 16:16–2, 1995.

[Bek05] George A. Bekey. Systems of robots: from cooperation to swarm behavior. IEEE SMC 2005, Hawaii – Computer Science Department, University of Southern California, December 2005.

[BPL04] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Jadex: A short overview. In *Main Conference Net.ObjectDays 2004*, pages 195–207, 9 2004.

[BPL05] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Jadex: A bdi agent system combining middleware and reasoning. In M. Klusch R. Unland, M. Calisti, editor, *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhäuser-Verlag, 9 2005. Book chapter.

[BRA94] Tucker Balch, Ronald, and C. Arkin. Communication in reactive multiagent robotic systems. *Autonomous Robots*, 1:27–52, 1994.

[Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[CDGU10] Cristina Carletti, Maurizio Di;Rocco, Andrea Gasparri, and Giovanni Ulivi. A distributed transferable belief model for collaborative topological map-building in multi-robot systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2010*, 2010.

[Cer08] E. Cervera. Practical multi-robot applications with player and jade. In *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, pages 2004 –2009, 2008.

[CMW09] I.Y.-H. Chen, B. MacDonald, and B. Wunsche. Mixed reality simulation for mobile robots. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 232 –237, May 2009.

[CND+10] L. Carlone, M.K. Ng, Jingjing Du, B. Bona, and M. Indri. Rao-blackwellized particle filters multi robot slam with unknown initial correspondences and limited communication. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 243 –249, May 2010.

[GVH03] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics (ICAR)*, Coimbra, Portugal, June 2003.

[KDP09] A. Kottas, A. Drenner, and N. Papanikolopoulos. Intelligent power management: Promoting power-consciousness in teams of mobile robots. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 1140 –1145, May 2009.

[KKF+10] Been Kim, M. Kaess, L. Fletcher, J. Leonard, A. Bachrach, N. Roy, and S. Teller. Multiple relative pose graphs for robust cooperative mapping. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 3185 –3192, may 2010.

[KNT+09] Ryo Kurazume, Yusuke Noda, Yukihiro Tobata, Kai Lingemann, Yumi Iwashita, and Tsutomu Hasegawa. Laser-based geometric modeling using cooperative multiple mobile robots. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 3200 –3205, May 2009.

[LaV06] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Also available at http://planning.cs.uiuc.edu.

[MAC97] Douglas C. Mackenzie, Ronald C. Arkin, and Jonathan M. Cameron. Multiagent mission specification and execution, 1997.

[MBF10] D. Miklic, S. Bogdan, and R. Fierro. Decentralized grid-based algorithms for formation reconfiguration and synchronization. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 4463–4468, may 2010.

[MLW09] R. Mead, R. Long, and J.B. Weinberg. Fault-tolerant formations of mobile robots. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4805 –4810, 2009.

[PB09] Alexander Pokahr and Lars Braubach. From a research to an industrial-strength agent platform: Jadex v2. In Hans-Georg Fill Hans Robert Hansen, Dimitris Karagiannis, editor, *Business Services: Konzepte, Technologien, Anwendungen - 9. Internationale Tagung Wirtschaftsinformatik (WI 2009)*, pages 769–778. Österreichische Computer Gesellschaft, 2 2009.

[PBJ10] Alexander Pokahr, Lars Braubach, and Kai Jander. Unifying agent and component concepts - jadex active components. In *In Proceedings of Seventh German conference on Multi-Agent System TEchnologieS (MATES-2010)*, 2010.

[PBL03] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: Implementing a bdi-infrastructure for jade agents. *EXP - in search of innovation (Special Issue on JADE)*, 3(3):76–85, 9 2003.

[PBL05] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A bdi reasoning engine. In J. Dix R. Bordini, M. Dastani and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, pages 149–174. Springer Science+Business Media Inc., USA, 9 2005. Book chapter.

[PBWL07] Alexander Pokahr, Lars Braubach, Andrzej Walczak, and Winfried Lamersdorf. Jadex - engineering goal-oriented agents. In F. Bellifemine, G. Caire, and D. Greenwood, editors, *Developing Multi-Agent Systems with JADE*, pages 254–258. Wiley & Sons, 1 2007.

[QCG$^+$09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully B. Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[RAB09] C. Rossi, L. Aldama, and A. Barrientos. Simultaneous task subdivision and allocation for teams of heterogeneous robots. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 946 –951, may 2009.

[RG10] Sebastian Rockel and Chen Gang. Mobile robotics project at hamburg university, http://www.sebastianrockel.de/irp09/html (checked July 11, 2011), 2010.

[RN03] S J Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach - 2nd Edition.pdf*, volume 82. Prentice Hall, 2003.

[SC09] P.M. Shiroma and M.F.M. Campos. Comutar: A framework for multi-robot coordination and task allocation. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4817 –4824, 2009.

[SMKR09] Xueqing Sun, Tao Mao, J.D. Kralik, and L.E. Ray. Cooperative multi-robot reinforcement learning: A framework in hybrid state space. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 1190 –1196, oct. 2009.

[SN04] Roland Siegwart and Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots.* Bradford Book, 2004.

[SN10] S. Srivastava and G.C. Nandi. Localization of mobile robots in a network using mobile agents. In *Computer and Communication Technology (ICCCT), 2010 International Conference on*, pages 415 –420, 2010.

[STSI09] Kazuya Suzuki, Tsunamichi Tsukidate, Masahiro Shimizu, and Akio Ishiguro. Stable and spontaneous self-assembly of a multi-robotic system by exploiting physical interaction between agents. In *Proceedings of the 2009 IEEE/RSJ international conference on Intelligent robots and systems*, IROS'09, pages 4343–4348, Piscataway, NJ, USA, 2009. IEEE Press.

[TvS10] D. Thomas and O. von Stryk. Efficient communication in autonomous robot software. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, Oct. 18 - 22 2010.

[UM09] P. Urcola and L. Montano. Cooperative robot team navigation strategies based on an environment model. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 4577–4583, Oct. 2009.

[YHTS10] Jing Yuan, Yalou Huang, Tong Tao, and Fengchi Sun. A cooperative approach for multi-robot area exploration. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1390 –1395, 2010.

Appendix A

This chapter lists additional material related to this work.

```
# 2011-06-07 Sebastian Rockel
# Pioneer model with devices
# File: devices.inc

include "pioneer.inc"
#include "utm30lx.inc" # another laser model
include "urgr.inc"
include "platte.inc"
include "laptop.inc"

define bobsblobfinder blobfinder
(
  colors_count 1
  colors [ "green" ]
  #fov 1.047196667 # 60 degrees = pi/3 radians
  fov 90 # degrees
  range 3
  #range_max 5
  # camera parameters
  #image [160 120]      #resolution
  image [80 60]         #resolution
)
define bobsgripper gripper
(
  pose [0.23 0.000 -0.20 0.000]
  color "gray"
)
define robot fancypioneer2dx
(
  laser240( pose [0.13 0 0 0] )
  #utm30lx( pose [0.13 0 0 0] )
  platte()
  laptop()
  bobsblobfinder()
  bobsgripper()

  obstacle_return 1
  ranger_return 1
  blob_return 0
)
define blob model
(
  size [ 0.3 0.3 0.0]
  boundary 0
  gui_move 1
  gripper_return 1
  obstacle_return 0
  laser_return 0
  ranger_return 0
  blob_return 1
)
```

List. A.1: The Pioneer robot device definition for the Stage simulator.

```
# Player drivers configuration for a Pioneer-2DX robot
# 2011-07-04 Sebastian Rockel
# File: pioneer.cfg

driver
(
  name "p2os"
  provides ["odometry:::position2d:0" "sonar:0" "power:0"
      "gripper:::gripper:0" "lift:::actarray:0" "dio:0" "audio:0"]
  gripper_outersize [0.5 0.5 0.5]
  gripper_innersize [0.4 0.4 0.4]

  port "/dev/ttyS0"
)
driver
(
  name "hokuyo_aist"
  provides ["ranger:1"]
  portopts
      "type=serial,device=/dev/ttyACM0,timeout=1,debug=0,baud=115200"
  pose [ 0.13 0 0 0 0 0 ]
  min_dist 0.02
  error_dist 5.6 # if range < min_dist, range is set to this value
)
driver
(
  name "rangertolaser"
  requires [ "ranger:1" ]
  provides [ "laser:0" ]
)
driver
(
  name "mapfile"
  provides ["map:0"]
  filename "bitmaps/tams_compl_red_map.png"
  resolution 0.05 # meters per pixel
  origin [-31.475 -7.8 ] # real-world location of the bottom-left-hand
      corner of the map
)
```

List. A.2: The Player driver configuration for a Pioneer-2DX robot includes the p2os driver for motor and sonar access, the hokuyo_aist driver for the laser ranger, an optional rangertolaser driver to convert laser into generic ranger data and a mapfile driver to provide a two-dimensional map of the environment.

```
# Desc: Player config file for localization and navigation
# Date: 2009-11-16
# CVS: $Id: amcl-sonar.cfg,v 1.2 2005-08-05 23:18:41 gerkey Exp $
# Copied and modified by: Sebastian Rockel, 2011-06-07
# File: planner_6666.cfg
# Load the map for localization and planning from the same image file,
# and specify the correct resolution (a 500x500 pixel map at 16m x 16m
# is 0.032 m / pixel resolution).
driver
(
 name "vfh"
 provides ["position2d:1"]
 requires ["6665:position2d:0" "6665:laser:0"]
 safety_dist_0ms 0.1 #The minimum distance the robot is allowed to get to obstacles when stopped.
 safety_dist_1ms 0.4 #The minimum distance the robot is allowed to get to obstacles when travelling at 1 m/s.
 #free_space_cutoff_0ms 2000000.0 #Unitless value. The higher the value, the closer the robot will get to
       obstacles before avoiding (while stopped).
 free_space_cutoff_1ms 1000000.0 #Unitless value. The higher the value, the closer the robot will get to
       obstacles before avoiding (while travelling at 1 m/s).
 max_speed 0.4 # The maximum allowable speed of the robot.
 max_speed_narrow_opening 0.1 #The maximum allowable speed of the robot through a narrow opening
 when stopped.
 when travelling 1 m/s.
 distance_epsilon 0.2 #Planar distance from the target position that will be considered acceptable. Set this
       to be GREATER than the corresponding threshold of the underlying position device!
 angle_epsilon 3 #Angular difference from target angle that will considered acceptable. Set this to be GREATER
       than the corresponding threshold of the underlying position device!
 #min_turn_radius_safety_factor 1.0 #??
 escape_speed 0.1 #If non-zero, the translational velocity that will be used while trying to escape.
 escape_time 5.0 #If non-zero, the time (in seconds) for which an escape attempt will be made.
 escape_max_turnrate 10 #If non-zero, the maximum angular velocity that will be used when trying to escape.
 #synchronous 0 #If zero (the default), VFH runs in its own thread. If non-zero, VFH runs in the main Player
       thread
 #weight_desired_dir 5.0 #Bias for the robot to turn to move toward goal position.
 #weight_current_dir 3.0 #Bias for the robot to continue moving in current direction of travel.
)
driver
(
 name "amcl"
 provides ["localize:0" "position2d:2"]
 requires ["odometry::6665:position2d:0" "6665:laser:0" "laser::6665:map:0"]
 #enable_gui 1 # Set this to 1 to enable the built-in driver GUI

 # Particle filter settings
 #
 odom_init 0 #Use the odometry device as the "action" sensor
 #pf_min_samples 100 #Lower bound on the number of samples to maintain in the particle filter
 #pf_max_samples 10000 #Upper bound on the number of samples to maintain in the particle filter
 #pf_err 0.01 #Control parameter for the particle set size. See notes below
 #pf_z 3.0 #Control parameter for the particle set size. See notes below
 init_pose [-21.0 4.0 0] #Initial pose estimate (mean value) for the robot
 #init_pose [-6.0 -5.0 0.0] #Initial pose estimate (mean value) for the robot
 init_pose_var [3.0 3.0 360.0] #Uncertainty in the initial pose estimate
 #update_thresh [0.2 0.52] #Minimum change required in action sensor to force update in particle filter

 # Set the 3 rows of the covariance matrix used for odometric drift
 #odom_drift[0] [0.2 0.0 0.0]
 #odom_drift[1] [0.0 0.2 0.0]
 #odom_drift[2] [0.2 0.0 0.2]

 # Laser settings
 #
 laser_pose [0.13 0 0] #Pose of the laser sensor in the robot's coordinate system
 laser_max_beams 50 #Maximum number of range readings being used
 laser_range_max 5.6 #Maximum range returned by laser
 #laser_range_var 0.1 #Variance in range data returned by laser
 laser_range_bad 0.02 #Probable laser min values
)
driver
(
  name "wavefront"
  provides ["planner:0"]
  requires ["output:::position2d:1" "input:::position2d:2" "6665:map:0"]
  alwayson 1
  safety_dist 0.3 #Don't plan a path any closer than this distance to any obstacle. Set this to be GREATER
       than the corresponding threshold of the underlying position device!
  distance_epsilon 0.4 #Planar distance from the target position that will be considered acceptable. Set this
       to be GREATER than the corresponding threshold of the underlying position device!
  angle_epsilon 10 #Angular difference from target angle that will considered acceptable. Set this to be
       GREATER than the corresponding threshold of the underlying position device!
  #cspace_file "planner.cspace" # Currently disabled in Wavefront driver
  #replan_dist_thresh 2.0 #Change in robot's position (in localization space) that will trigger replanning
  #replan_min_time 2.0 #Minimum time in seconds between replanning. Set to -1 for no replanning. See also
       replan_dist_thresh
  #force_map_refresh 0 # If non-zero, map is updated from subscribed map device whenever new goal is set
)
```

List. A.3: The Player driver configuration for the local path planner VFH, the global path planner Wavefront and the localization component AMCL.

```
# 2011-06-07 Sebastian Rockel
# Stage Simulation with robots
# file: uhh1.cfg
driver
(
  name "stage"
  provides ["simulation:0"]
  plugin "stageplugin"
  # load the named file into the simulator
  worldfile "uhh1.world"
)
driver
(
  name "stage"
  provides ["position2d:0" "ranger:0" "ranger:1" "gripper:0"]
  model "r0"
)

driver
(
  name "stage"
  provides ["6667:position2d:0" "6667:ranger:0" "6667:ranger:1" "6667:gripper:0"]
  model "r1"
)
driver
(
  name "stage"
  provides ["6669:position2d:0" "6669:ranger:0" "6669:ranger:1" "6669:gripper:0"
      "6669:blobfinder:0"]
  model "r2"
)
# Proxies of real robots
driver
(
  name "stage"
  provides ["6671:position2d:0" "6671:blobfinder:0"]
  model "r3"
)
driver
(
  name "stage"
  provides ["6672:position2d:0" "6672:blobfinder:0"]
  model "r4"
)
driver
(
  name "stage"
  provides ["6673:position2d:0" "6673:blobfinder:0"]
  model "r5"
)
# laser device still needed for vfh and acml driver
driver ( name "rangertolaser" requires ["6665:ranger:1"] provides ["6665:laser:0"])
driver ( name "rangertolaser" requires ["6667:ranger:1"] provides ["6667:laser:0"])
driver ( name "rangertolaser" requires ["6669:ranger:1"] provides ["6669:laser:0"])

# Map driver
driver
(
  name "mapfile"
  provides ["map:0"]
  filename "bitmaps/tams_compl_red_map.png"
  resolution 0.05 # meters per pixel
  origin [-31.475 -7.8 ] # real-world location of the bottom-left-hand corner of
      the map
)
```

List. A.4: The Player TAMS world configuration.

```
# 2009−11−17 TAMS
# file: urg.inc

define urg_ranger ranger
(
  sensor (
    pose [ 0.13 0 0 0 ]
    size [ 0.1 0.1 0.1 ]
    range [ 0.02 5.6 ]
    fov 240.0
    samples 682
  )

  watts 2.0
  color_rgba [ 0 1 0 0.15 ]
)
```

List. A.5: The Stage URG-04LX Laser model definition.

```
# 2011−06−07 Sebastian Rockel
# Hokuyo UTM−30LX definition
# file: utm30lx.inc

define utm30lx ranger
(
  sensor (
    range [ 0.1 30 ]
    fov 270.0
    samples 1080
    watts 8.0
    color_rgba [ 0 0 1 0.15 ]
  )

  # generic model properties
  model (
    color "blue"
    size [ 0.05 0.05 0.1 ]
    pose [ 0 0 −0.2 0 ] # to be set in robot definition
  )
)
```

List. A.6: The Stage UTM-30LX Laser model definition.

```bash
#!/bin/bash
# 2011-03-30 Sebastian Rockel
# Starts Stage simulation with robots
# file: startSimulation.sh

killall player
killall player
killall player
killall playernav

player uhh1.cfg &

sleep 3

./startPlanner.sh &
```

List. A.7: The TAMS Simulation Start Script.

```bash
#!/bin/bash
# 2011-03-16 Sebastian Rockel
# Start Player planner servers
# Multiple robots scenario
# file: startPlanner.sh

player -p 6666 planner_6666.cfg &
player -p 6668 planner_6668.cfg &
player -p 6670 planner_6670.cfg &

sleep 2

playernav localhost:6665 localhost:6666 localhost:6668 localhost:6670 &
```

List. A.8: The Player Navigation Start Script.

```bash
#!/bin/bash
# 2011-03-18 Sebastian Rockel
# Start Scale scenario
# startScale.sh

killall player
killall player
killall player

player scale.cfg &
player -p 6667 scale_laser.cfg &
player -p 6666 planner_scale.cfg &
```

List. A.9: The Swarm Scenario Start Script.

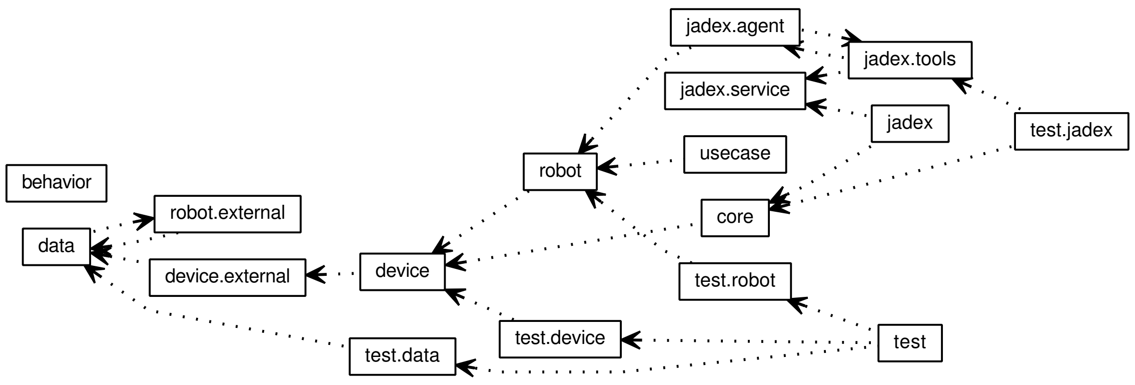


Fig. A.1: This dependency graph illustrates all middle layer (RSAL) components. As this graph is automatically generated from the corresponding components (in Java packages) some components are split into several Java packages. This separates the public interface from the internal code. Arrows indicate whether a package actually uses another package or inherits from it.

Erklärung

Ich, Sebastian Rockel, Matr.-Nr.: 6095961, versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereichs einverstanden.

(Ort, Datum) (Unterschrift)