Karlsruhe Institute of Technology
Department for Eletrical Engineering
and Information Technology

Institute for Information Processing
Technology - ITIV

A Mobile Robot System for Ambient Intelligence

Bachelor Thesis of

Matthias Mayr

April the 1st, 2014

Head of Institute: Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat W. Stork

Supervisors: **Halmstad University:**
Roland Philippsen
Wagner De Morais
Nicholas Wickström
Karlsruhe Institute of Technology:
Johannes Schneider
Sven Schmidt-Rohr

Abstract

Over the last years, Ambient Intelligence (AmI) has been pointed out as an alternative to current practices in home care. AmI supports the concept of Ambient Assisted Living, which aims to allow older people to remain independent at their own homes for longer. The integration of a mobile robot into a database-centric platform for Ambient Assisted Living is described in this thesis. The robot serves as a first-aid agent to respond to emergencies, such as a fall, detected by the intelligent environment. To accomplish that the robot must 1) be able to receive tasks from intelligent environment; 2) execute the task; 3) report the progress and the result of the task back to the intelligent environment. The system of the robot is built on top of the Robot Operating System, while the existing intelligent environment on a PostgreSQL database. To receive tasks from the intelligent environment, the robot maintains an active connection with the database and subscribes to specific tasks. A task, for example, is to find a person in the environment, which includes asking if the person is doing well. To find a person a map-based approach and a face recognition are used. The robot can interact with people in the environment using text-to-speech and speech recognition. The active connection with the database enables the robot to report back about the execution of a task and to receive new or abort tasks. As a conclusion, together with an AAL system, mobile robots can support people living alone. The system has been implemented and successfully tested at Halmstad University on a Turtlebot 2. The code is available on Github[1].

[1] Link to the Github account: `http://www.github.com/matthiashh`

Details

First Name, Surname: Matthias Mayr

E-Mail: matthias.mayr@student.kit.edu

Degree Program: Electrical Engineering and Information Technology

Title of Thesis: A Mobile Robot System for Ambient Intelligence

Thesis ID: Bachelor Thesis ID-1806

Keywords: Robotics, Ambient Intelligence, Ambient Assisted Living, Robot System, Active Database, Healthcare

License

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbständig und ohne unzulässige fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die wörtlich oder inhaltlich übernommenen Stellen habe ich als solche kenntlich gemacht. Die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der Fassung vom Mai 2010 habe ich beachtet.

Halmstad, den 1. April 2014

Matthias Mayr

Contents

List of Tables

List of Listings

List of Figures

Chapter 1

Introduction

In the thesis the approach of a mobile robot as part of a home assisted living system is examined. As part of this system the robot receives and executes the task to search for a human and asks a found person for its wellbeing.
It is assumed that an occupancy map of the operation area is provided. The map has to include static obstacles. The home assisted living system provides places to search for a person.
The approach has been tested in scenarios with lying, sitting and standing humans.

1.1 Motivation

These days the most societies in the world are becoming older [Nat01] and a high percentage of elderly live at home and would like to stay there instead of going to a nursing home. But as they are often living alone and are just receiving visitors a few times a day this comes with additional risks. Incidents like a fall or a stroke often stay undetected for hours and cause avoidable health problems.
Home assisted living systems try to close that gap by monitoring the acitivities and offering the possibility to perform emergency calls. But most of these systems need either a huge effort to install them, like a system of cameras and appearance sensors or rely on the discipline of the users like wearable emergency buttons. In this thesis the approach of a mobile robot as an agent and sensor of a smart home is examined. With a mobile robot it is possible to turn a normal appartment into an assisted living appartment without the overhead of the installation of various sensors. A small mobile robot can be a personal care giver and help provider. It can be an affordable solution which is small enough to be taken whereever one goes.
Furthermore mobile robots can also be used in other applications like retirement homes and hospitals. With various appartments, sensor types and multiple robots these agents

can investigate unclear situations and fulfill tasks given by a coordination system.

1.2 Scenario

The developed system should be able to fulfill a defined scenario based on the existing 'smart bedroom' [dMW13]. This bedroom can detect when a person leaves the bed and is part of an Ambient Assisted Living (AAL) system.
If a person leaves the bed at night and does not return after a certain timeframe, the AAL system can assign the task to search for the person to one of the connected robot. The robot receives the task and can query for a list of places to search for the person. If the robot finds the person, it should ask for the well-being of that person. At the end of the search the robot should report the results to the AAL system.

1.3 Assignment

The robot has to be connected to the smart home approach presented in [dMW13] and presented in section 2.3. This approach is using an active database which is implemented with a PostgreSQL database. The robot should be able to report its status to the database and offer the execution of the tasks the robot can perform. It should be possible to receive tasks from the database and to manage the execution of several tasks. Furthermore the robot should be able to query for task specific information and report sensor information demanded by the database. Within this context, the applied system should be extensible with other tasks and should also avoid shutdown times if a new kind of task like 'jump-on-a-table' is installed or updated.
Based on this system the robot should be able to search for a person in an appartment. It is assumed that there is a static map available and the robot receives a list of places where the home assisted living system assumes the position of the person to be. If the search succeeds the robot should investigate the wellbeing of the person and inform the database about the situation.

1.4 Conventions

For the development of this approach the base of a Turtlebot 2 robot should be used. This platform is equipped with a Kinect depthcamera and navigation sensors. The execution of the search should not be based on the existance of other static sensors. It should be possible to find a human person after an incident happened. It can not be assumed that the robot observes the actual event.

1.5 Contributions

The contribution of this thesis is a setup to use a robot with Ambient Intelligence (AmI) and send an robot to search for a person in a known environment. It shows that a robot can be part of the AmI and can execute tasks to enhance the capabilities of such a system. Unlike other projects focusing on bigger and more expensive robots this setup is an affordable solution showing good results.
The detection software is storing detections with the assigned name, place and time for the whole runtime of the robot. Additionally detections can be verified using human robot interaction. The developed software for human robot interaction as well as the robot managing software have been offered as an easy to use interface for further purposes. The human robot interface as well as the person detection software can be used as standalone software on any robot.
The whole new written software is available on Github[1]

[1] Link to the Github repository: `http://www.github.com/matthiashh`

Chapter 2

Background and Related Work

This chapter presents the state of the art of the fields of the thesis and introduces the existing Ambient Assisted Living (AAL) system based on an active database.

2.1 Robots in Healthcare and Ambient Assisted Living

In the recent years major steps in healthcare robotics have been achieved. In Japan robots lift people from a bed [MHN+10]. In hospitals mobile robots manage the transportation of surgery equipment [OFD+09] or successfully assist at operations [BWB+].
The robot Care-O-Bot[1] is specially designed [GRH+09] for elderly care. But although it consists of state of the art hardware and has good set of capabilities this robot is still to expensive to be used as a common solution. The project Mobility Aid for Handicapped Persons (MAID) [mai] follows the idea of an motordriven wheeled walker. This approach has the advantage that such a helper instrument will have a higher acceptance by the elderly than a robot.
An overview of robots in health and social care will be given by [DB13]. But whereas a lot of prototypes and research projects exist in that field the commercial products are limited to telepresence robots.
Nowadays affordable service robots like vacuum cleaners and lawnmowers reduce prices of small robot platforms and pave the way to affordable personal service robots.
Recent developments in sensing and computing make it possible to build reasonably priced, small and autonomous robots with an increasing set of capabilities. Even cheap platforms come with gyrometer and depthsensors and first attempts of low-cost manipulators[2] are developed.

[1] The website of the Care-O-Bot project. `http://www.care-o-bot.de/en/care-o-bot-3.html`

[2] Link to the Turtlebot arm: `http://wiki.ros.org/turtlebot_arm`

2.2 Ambient Intelligence and Ambient Assisted Living

Ambient Intelligence (AmI) is the concept of 'using input from sensor systems distributed throughout the environment, computing devices could personalize themselves to their current user, adapt their behaviour according to their location, or interact to their surroundings'. [CVBK11] A discrete system can be built upon the devices to manage the behaviour and to adapt the enviroment.

Ambient Assisted Living (AAL) supports older adults in order to enable them to stay longer at home. The fact that populations of industrial societies are aging [Nat01] is a great motivation for the development of ambient assisted living technologies.

Starting with simple systems like emergency buttons this branch is developing towards aware smart homes.

A review of existing system is provided in [ARA12] and [RM13] and shows that the field has great opportunities especially for the support of the elderly.

In [SFR11] it is shown how robots could support and assist older adults. An overview of robots in home automation and their needs is given in [HTK+05].

A mobile robot can be a sensor as well as an actor for these systems. Whereas all other components of the system are stationary, the robot can be sent to interesting places. Furthermore, the robot is an additional communication channel to the person and gives the possibility to address a person directly.

2.3 Smart Home as an Active Database

In [dMW13] an architecture of an AAL system based on an active database is proposed. An active database is a relational database with active rules. Active rules allow reasoning based on the incoming and stored data. As shown in figure 2.1 multiple sensors and actuators can be connected to a database using resource adapters. This middleware builds the connection to the database and queries for information to connect with the attached hardware. The abstraction layer of the resource adapters makes it possible to connect a large number of devices to the database. Once a device is connected it can call a User Defined Function (UDF) to insert or access data.

The active database is enhanced with the techniques of big data analysis and machine learning of the MADlib[3]. This makes it possible to react to the stored data and incoming data. For example, to detect if a person is lying in the bed the value of the standard deviation of load cells under the beds legs is used. Based on active rules the database can also react to events and for example trigger the resource adapter of a lamp to turn on/off the light.

[3] Website of the MADlib project: `http://madlib.net/`

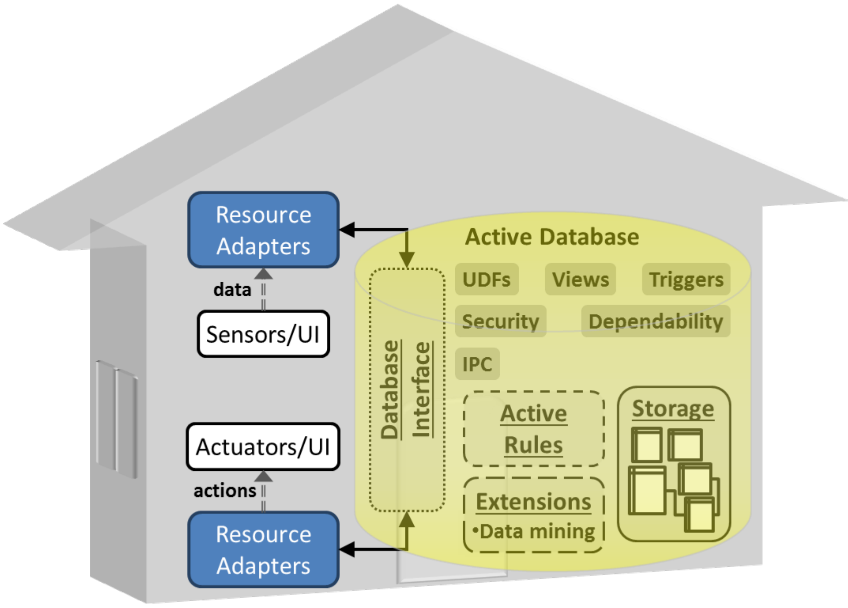

Figure 2.1: The proposed architecture of an active database in a smart home. Every resource is connected through a resource adapter. The active database components are examined in and the figure is taken from [dMW13].

The database allows to define UDFs and can manage access rights on functions and tables. Access management is based on user accounts which can have different roles assigned. Based on the roles a user can access different kind of data. For example an informal caretaker should not be able to access sensitive data, whereas a doctor should have access to health information.
A mobile robot can be a resource adapter for the database. In this context the robots acts both as actuator and sensor.

2.3.1 Smart Bedroom

An 'smart bedroom' has been developed as a demonstrator of this proposal. Based on the data of load cells in and under the bed, the active database can

- detect whether a person is in the bed
- calculate the heart rate and
- compute the breath rate.

Additional infrared sensors allow to estimate the position of the person in the room and actuators like the lamps can be turned on or off according to the situation.

Figure 2.2: A photo of the 'smart bedroom'

Chapter 3

Materials and Tools

This chapter introduces the blocks of the robot setup. It will give an overview of the existing hardware and software components used in this thesis. The chosen and developed components to solve the scenario are explained in chapter 4.

3.1 Robot Operating System (ROS)

The ROS is a software framework for the development of robot related software.

The roots of this robotic middleware go back to 2007 when the Stanford Artificial Intelligence Laboratory build a system for their robot STAIR [QBN07]. Today it is an open source project developed under the lead of the Open Source Robotics Foundation[1] with contributers all over the world. The software is released under Berkley Software Distribution (BSD) Licence which allows the integration in proprietary software projects. ROS comes with support for different operation systems[2] and a lot of robots[3] - a selection can be seen in table 3.1.

The ROS framework works as a distributed client server system. It works via network and allows to start client programs on every connected computer. The infrastructure is transparent and can be easily monitored by various debugging tools.

[1] Website of the Open Source Robotics Foundation `http://osrfoundation.org/`

[2] Supported operation systems: `http://wiki.ros.org/ROS/Installation`

[3] Supported robots: `http://wiki.ros.org/Robots`

Table 3.1: Selection of Operating Systems (OS) and hardware supported by ROS.

Operating Systems	**Hardware**
Official	Willow Garage PR2
Ubuntu Linux	Lego NXT
Experimental	Turtlebot 1 & 2
Mac OS X	Shadow Hand
Ubuntu ARM	AscTex Quadrocopter
Microsoft Windows	Care-O-bot

3.1.1 Building blocks of ROS

ROS Master and Parameter Server

Every ROS system needs one master. The master must be reachable from every node and acts as a nameserver. Every program in the ROS system registers at the server. If different nodes want to connect to each other, the master provides them with connection information and the nodes build a direct connection to each other. Multiple robots can share one master and therefore share information as long as the namespaces of the topics, for example the driver for the mobile base, are seperated.
The master is started together with the parameter server. This server allows the central storage and editing of attributes. For example in this project the connection information for the database is stored in the parameter server to allow modules to build up an own connection.

ROS Nodes

A node is a running program which is connected to the ROS environment. On startup every ROS node registers at the master using a unique name. Examples for nodes are:

- Robot Control Software
- Camera Driver
- Motor controller

A setup for navigation has usually about 30 registered nodes whereas a full setup can easily have 70 or more nodes interacting with each other. As it will be described in section 4.3.3.1 programs which implement the execution of a specific task like 'jump-on-the-table' are called modules. But these modules are still ROS nodes as they are programs which are connected to the ROS environment.

ROS Messages

Messages are objects sent over network allowing different nodes to provide Inter Process Communication (IPC). For example the camera driver publishes Red Green Blue (RGB) images in the message type 'sensor_msgs/Image.msg' which can be seen in listing 3.1.1. ROS comes with various common messages, but it is also easily possible to create an tailored message based on the specific needs. Messages are defined in a programming language agnostic way. On the compilation of the package defining the message ROS creates header files for every supported programming language. Together with the network transparency of ROS this allows to connect a sender written in C++ to a client written in Python and running on another machine.

```
Header header          # Header timestamp should be acquisition time of image
                       # Header frame_id should be optical frame of camera

uint32 height           # image height, that is, number of rows
uint32 width            # image width, that is, number of columns
string encoding         # Encoding of pixels -- channel meaning, ordering,
    size
                        # taken from the list of strings in include/
                            sensor_msgs/image_encodings.h
uint8 is_bigendian      # is this data bigendian?
uint32 step             # Full row length in bytes
uint8[] data            # actual matrix data, size is (step * rows)
```

Listing 3.1: ROS message for an image. One message of this type is sent for every captured frame.

ROS Topics

ROS follows a topic based publish/subscribe pattern. 'Topics are named buses over which nodes exchange messages.' [ros14h]. The topics names are usually composed of the sending part and the type of information. For example the cameras RGB-Image is by default published on the topic '/camera/rgb/image_color'. It is possible to have multiple publishers as well as multiple subscribers to one topic as long as they use the same type of message. During startup a publisher informs the master about the topics name. A subscriber queries the master for connection information to a specified topic name and builds up a direct connection to the publisher. The system of topics is very robust and supports subscribing before a publisher announced it as well as adding additional subscribers during runtime.

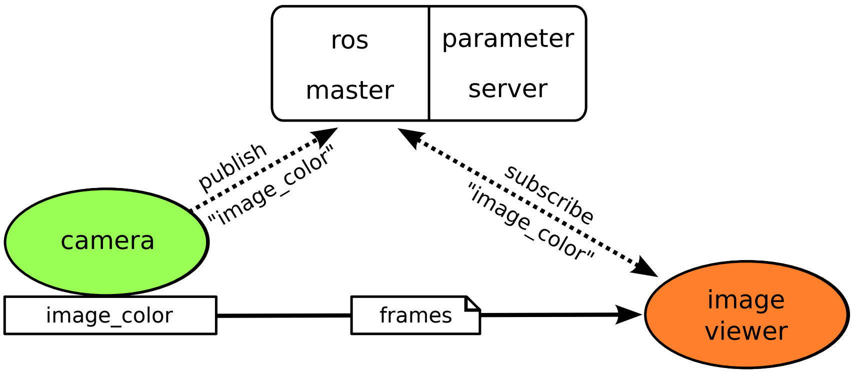

Figure 3.1: ROS Nodes for the camera publishes the topic 'image_color' and the image viewer subscribes to the topic. After the subscription the viewer receives frames directly.

ROS Service

In contrast to the unidirectional messages a ROS service provides request-reply interaction by Remote Procedure Call (RPC). Services can be defined in the same way as messages and are programming language agnostic. An example for a service is listing 3.1.1. A service is blocking on the client and based on the server implementation usually using the main thread of the server.

```
Header header          # Carying a sequence number and a timestamp
string[] name_array    # an array of possible names
---
bool successfull        # was the right name in the array?
bool answered          # did someone answer?
string label           # right name, if sucessfull
```

Listing 3.2: . The first part is the request. The second part the response of a call of that service.

Actionlib

The actionlib[4] is a widely used extension of the ROS core components. The concept of the actionlib is examined in [LPP+11] and is mainly used for long executing tasks like driving to a specific point. In addition to the request-reply concept of services, these tasks benefit from frequent feedback as well as from the possibility to abort a task. Both is not offered by the services.
The actionlib offers a standarized interface for a task state machine. The possible interaction between the server and the client can be seen in figure 3.2.

[4] The actionlib in the ROS wiki: `http://wiki.ros.org/actionlib`

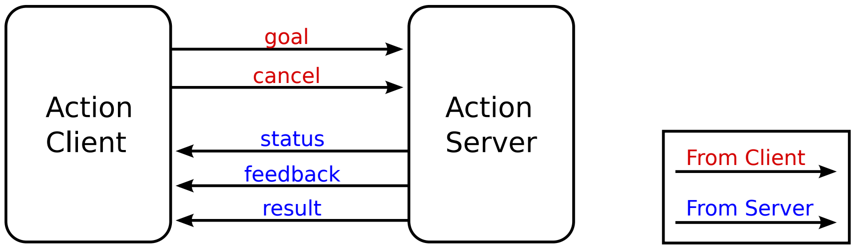

Figure 3.2: The interface between an actionserver and an actionclient

An action is a predefined object consisting of a goal, a result and a feedback section. An example for an action can be seen in listing 3.3. An action is sent by an actionclient to the corresponding actionserver and is non-blocking for the client.
Whenever a goal is sent to an actionserver two goalhandles are created:

- A server goalhandle on the server side and a
- A client goalhandle on the client side.

Goalhandles are objects, that allow to access a task in order check the state of the task, cancel the task or receive feedback. The state of a goal follows different state patterns on the client side (figure 3.3) and the server (figure 3.4) side in order to allow a clean implementation of the management.

```
# Define the goal
uint32 task_id
string task_name
uint32 priority
---
# Define the result
bool success
string end_result
---
# Define a feedback message
uint8 percentage
string intermediate_result
```

Listing 3.3: The action file of a task as an example

3.1.2 Capabilities of ROS

The ROS framework supports robot programmers with various implemented functionalities. The main ones are:

- Drivers for cameras, Intertial Measurement Unit (IMU), laserscanner and other sensors
- Image processing as well as a bridge to Open Source Computer Vision Library (openCV)
- 3D processing with pointclouds and depthimages
- Support for robot platforms
- Coordinate transformation and mangement of coordinate systems
- Motion planning for manipulators and navigation
- A robot simulation software

Programming paradigms and guidelines for enhancements of ROS are summarized in ROS Enhancement Proposals (REP).
The implementation of this system is based on the version 'hydro'.

3.2 Robot Turtlebot 2

The Turtlebot 2 is the second generation of a small and lowcost robot developing platform assembled from popular hardware components. The hardware specifications are released under the FreeBSD Documentation Licence and the software is fully open-source software. The robot is well integrated into ROS which allows rapid prototyping and gives the possibility to adjust the software to the projects needs. The delivered sensors can be seen in table 3.2 and mainly support navigation purposes. The Turtlebot can be used as an personal robot and can be purchased with a docking station. The robot can automatically connect to the docking station and charge itself as well as the delivered laptop. This allows continuous operation.
With the differential drive the robot can operate in smooth indoor environments but own experiences show, that it is not able to pass tresholds higher than 2 cm.
The platform offers an payload of 5 kg [Rob13] which is enough to cary a standard laptop and several sensors. With a maximum velocity of $0.65\,\mathrm{m\,s^{-1}}$ the robot can operate fast enough for real time applications.

3.2.1 Kinect Camera

The availability of Kinect cameras at the end of 2010 revolutionized the sensing in robotics. Although similar sensors like the Swissranger have been available before, the prices dropped to a tenth. In addition to an RGB image depth sensors offer a depthimage

Table 3.2: Sensors in the Turtlebot 2 robot [Rob13]

Type	**Model / Detail**
3D Vision Sensor	Microsoft Kinect
Wheel Encoders	11.5 ticks/mm
Gyrometer	factory calibrated, 100 deg/s
Bump Sensors	front, front right, front left
Cliff Sensors	front, front right, front left
Wheel Drop Sensors	one each wheel

allowing to locate every point in 3D space. This made it affordable to use this 3D data without the high effort of calibrating a stereo camera system or the use of a rotating laserscanner [SD03].
Within the ROS framework three kinds of raw data streams and another three processed streams are available. The streams are shown in figure 3.6. It has to be noted that the infrared image and the RGB image can not be accessed at the same time.

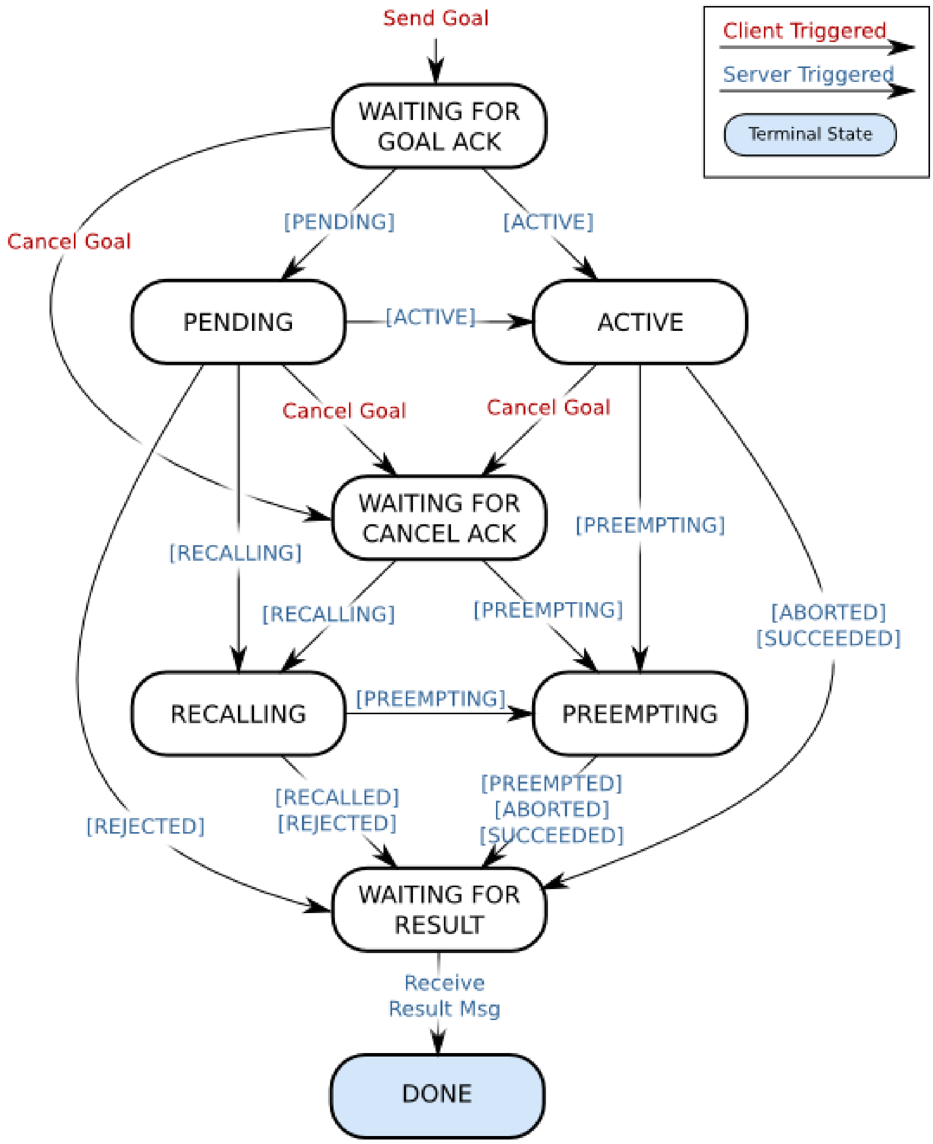

Figure 3.3: States and transitions of goals on the side of an actionlib client (taken from [ros14a])

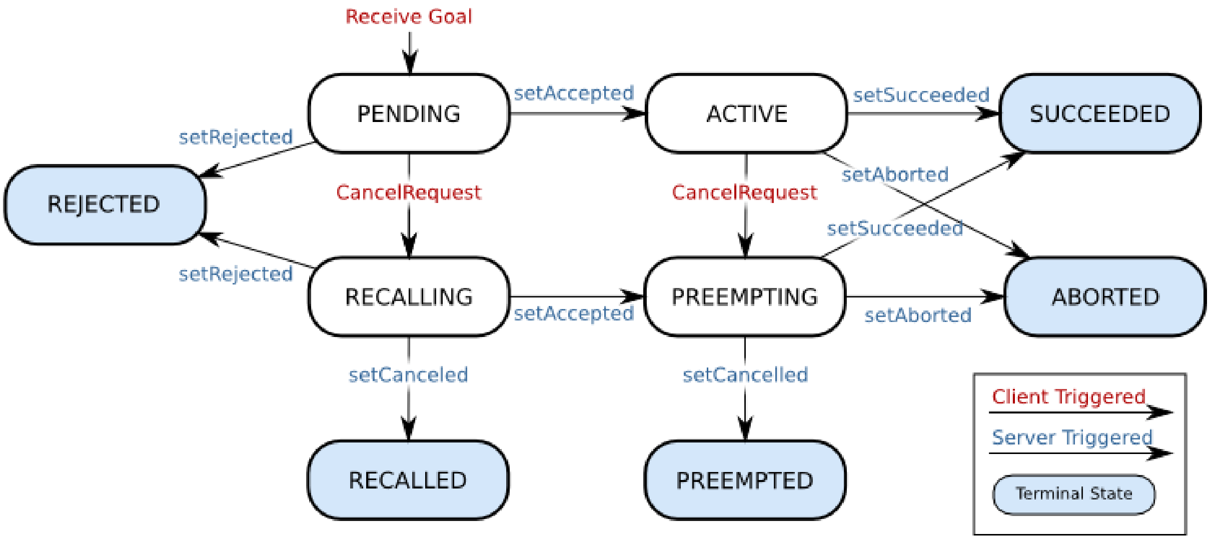

Figure 3.4: States and transitions of goals on the side of an actionlib server (taken from [ros14a])

Figure 3.5: Picture of the Turtlebot setup

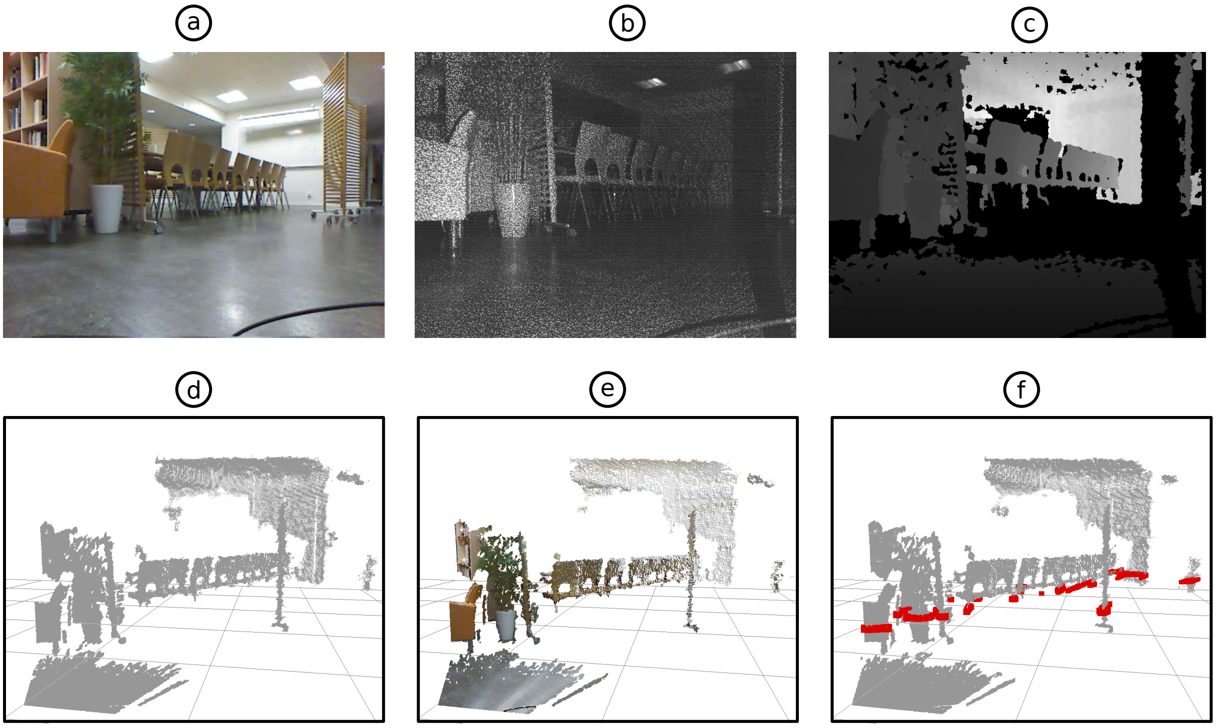

Figure 3.6: Available data streams using the Kinect camera

a: RGB image stream

b: Infrared image stream; the pattern of the infrared emitter can be seen

c: Depthimage; the grayscale represents the distance of the point

d: Pointcloud calculated from the Depthimage

e: Registered pointclound; every point has the color of the RGB image assigned

f: Calculated laserscanner data is shown as red dots

Chapter 4

Solution and Implementation

In this chapter different methods for a solution of the given assigment are evaluated and the implementation is presented. Each section starts with the requirements of the problem, leads to an overview over the possible methods to solve it and ends with the explanation of the implementation.

4.1 System Overview

The whole system consists of an existing active database, the hardware of the robot and the programs running on the robot. It can be seen in figure 4.1.
The resources of the robot are managed by the robot controller. This program implements the resource adapter (explained in 2.3) for the active database and grants modules access to the resources. Modules are programs which implement a specific task the robot can execute for the AmI. In the figure the component 'Coordination of the Search' is such a module and implements the task 'find_person'.
Furthermore a module needs utilities in order to be able to execute the task. Utilities do not implement a full task but can provide specific capabilities. For example 'Human Interface','Navigation' and the 'Person Detector' are utilities.
Modules and the robot controller can have an integrated database binding if it is necessary to exchange information with the AmI.

4.2 Navigation and Mapping

4.2.1 Requirements

The robot has to be able to reach points on a given map. It should be easily possible to create a new map in order to use the robot at different places. The map must have a

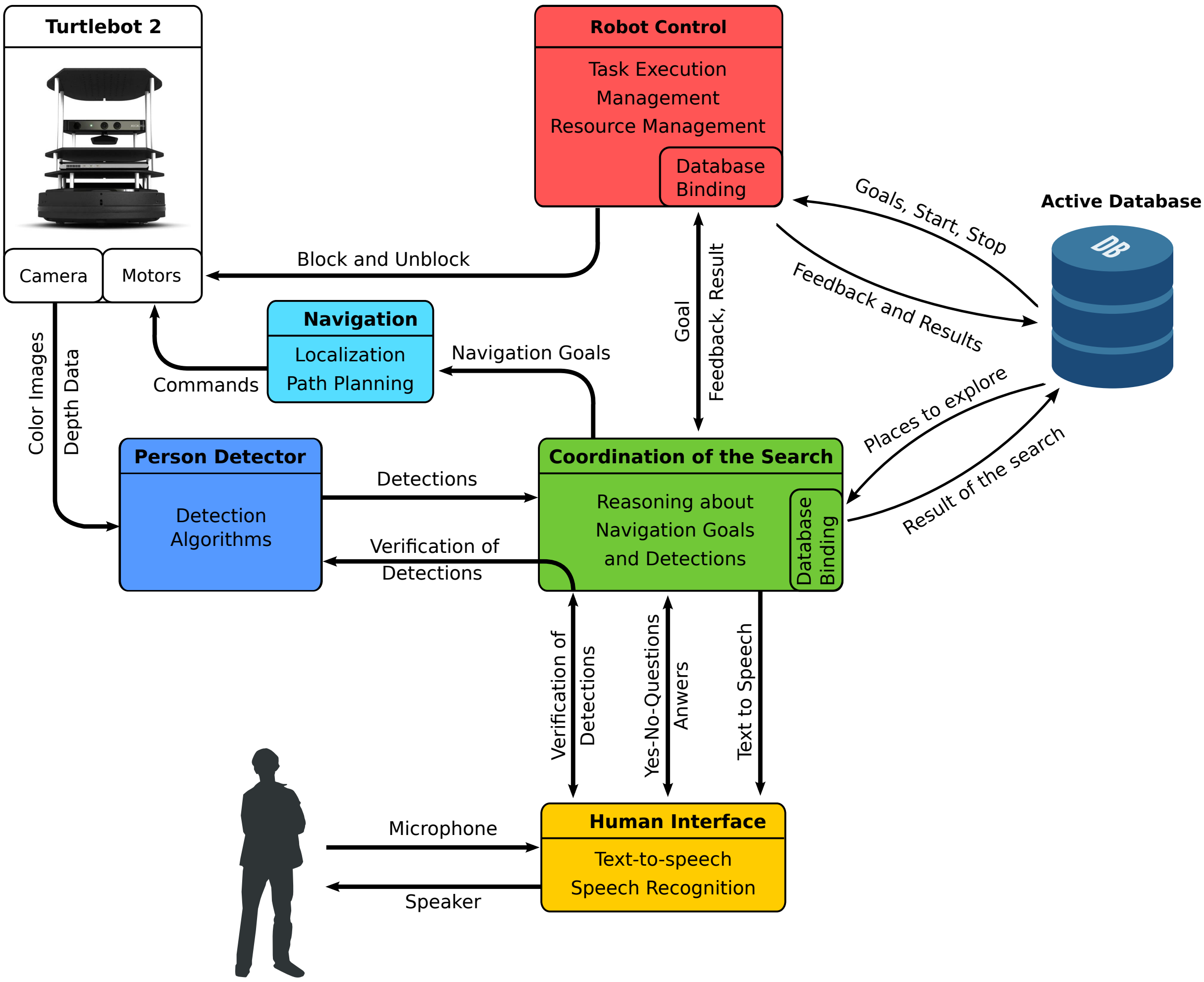

Figure 4.1: Simplified overview of the hardware and software components

coordinate system which makes it possible to exchange defined points on the map between the database and the robot. The map representation should furthermore support the calculation of a difference map between the current situation and the static map as a chosen method to find additional objects (shown later in section 4.5.2.3).

4.2.2 Solutions

4.2.2.1 2D Navigation and Mapping

Two dimensional maps are the traditional representation of map data. In ROS this type is represented as an occupancy map. The data is stored in a pixel image where every pixel can be

- occupied,

- free space
- or unknown.

Through this simple representation and the storage as a pixelgraphic these maps can be easily created from a lot of data sources. Figure 4.2 shows a map from a Simultanious Location and Mapping (SLAM) process and a converted map of a floor plan. Among the pixelgraphic a file with metadata about the scale and the origin is stored.

One drawback of the 2D navigation software is the missing possibility to update maps. Once a map is created the current implementation of the 2D navigation software in ROS can not update the map. The second drawback is mandatory expectation of a laserscanner as navigation sensor. As the Turtlebot does not come with a laserscanner this data is calculated by slicing out the horizontal line of a depthimage[1]. This means that every obstacle being lower than the mounting height of the Kinect of 32 cm (own measurement) is neither mapped nor recognized during runtime.

On the other hand, debugging this kind of map is far easier and the two dimensional representation does not need a lot of computation during runtime. As the system should be easily adaptable and a 2D representation allows an easier implementation of a map based approach evaluated in section 4.5.2.3, this method has been chosen.

4.2.2.2 3D Navigation and Mapping

In the last years the concept of OctoMap [HWB+13] has been introduced. This representation is using the memory saving Octree format. The approach comes with implemented packages for mapping, visualization and the use of maps[4]. A visualization of a 3D map can be seen in figure 4.3.

An advantage of this set of packages is the possibility to update an existing map during runtime. Furthermore the drawback of using the Kinect as a substitute for the laserscanner does not exist.

But OctoMap comes with a higher effort of computation . This is hard to fulfill with a small robot. Furthermore the implementation of map based approach to find additional obstacles becomes more difficult.

[1] By default the 10 middle rows of each column are taken and the closest point is used as the laserscanner result for that angle. See `http://docs.ros.org/hydro/api/depthimage_to_laserscan/html/classdepthimage__to__laserscan_1_1DepthImageToLaserScan.html`

[3] gmapping is one of the most used SLAM implementations. The gmapping website: `https://openslam.org/gmapping.html`

[4] OctoMap in the ROS wiki: `http://wiki.ros.org/octomap`

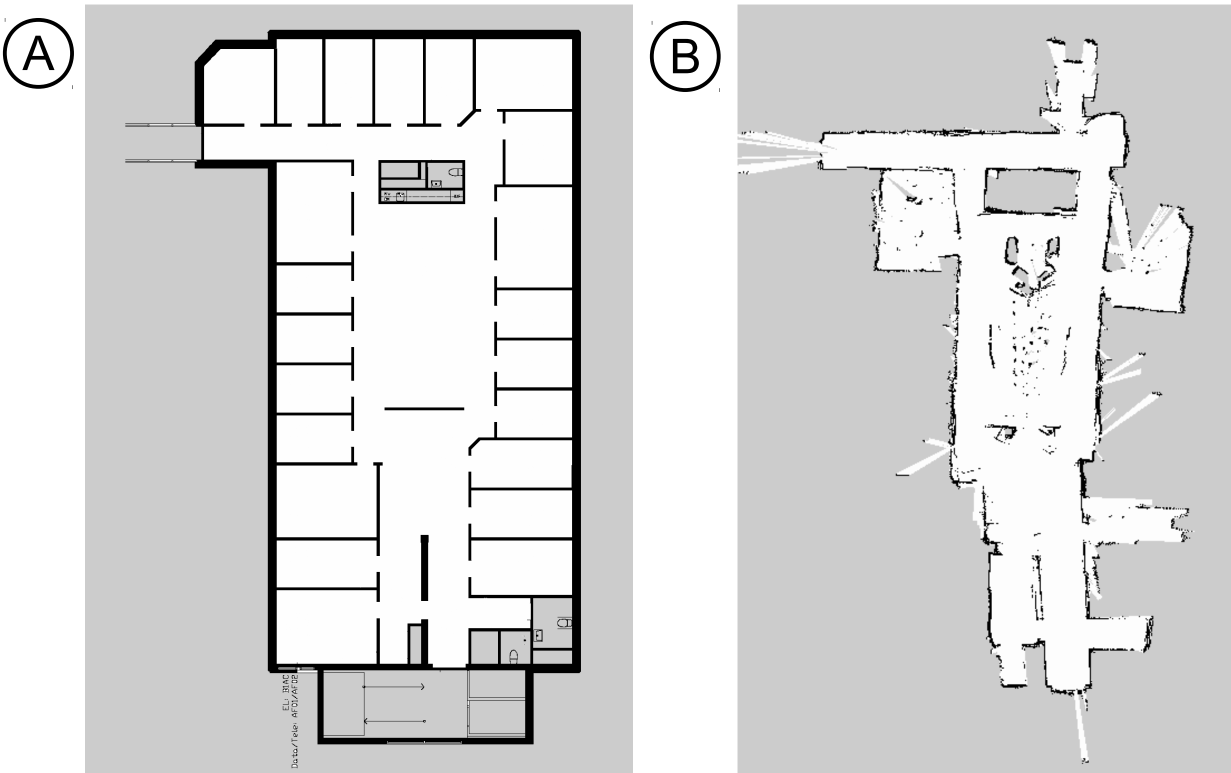

Figure 4.2: A: occupancy map based on the floor plan
B: a map created with the gmapping SLAM implementation[3]

4.2.3 Implementation

The 2D navigation software does not need any further implementation. For the test a SLAM based map has been used; it can be seen in figure 4.2.

The localization implements the adaptive Monte Carlo approach 'which uses a particle filter to track the pose of a robot against a known map' [ros14b]. The particles representing pose assumptions can be seen in the right picture of figure 4.10. The initial positioning can either be done manually or doing a global localization on the map. For a global localization particles are spread over the whole map and while the robot is moving some assumptions are discarded.

To navigate to a specific place it is enough to send a goal to the navigation software and monitor the state of the goal to know if it suceeded.

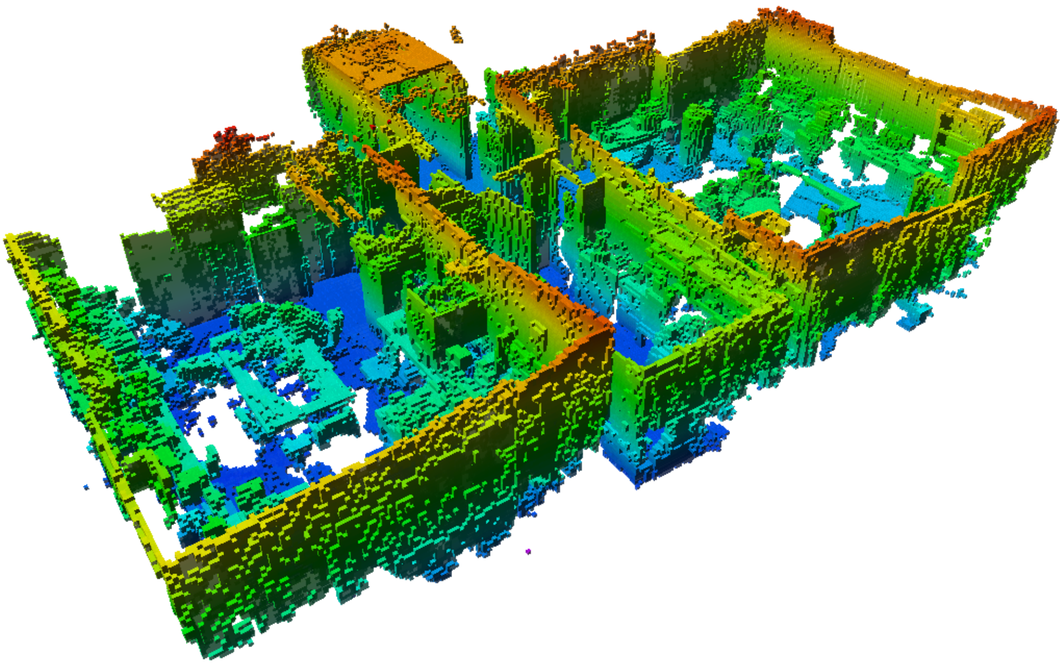

Figure 4.3: Three rooms in the OctoMap 3D representation (taken from [ros14e])

4.3 Main Robot Control

4.3.1 Requirements

A robot has a limited amount of resources and a management tool has to avoid that several programs access the same resources at the same time. The robot control software should run once on the robot and coordinate the access. As an agent for the database the robot should be able to handle discrete goals and manage their execution in order of the goal priorities. Moreover the cancelation of goals should be possible. A goal could be to execute a specific task like 'goto' or 'find_person'.
It should be possible to add new software modules for new kind of tasks during run-time. On startup the controller should build up a database connection and query the AAL-system for the robots configuration as well as supply other software modules with information for a database connection.

4.3.1.1 Triggers of the Active Database

In section 2.3 it is described that the active database can trigger resource adapters. In this setup the robot control is the only software component listening to the channels and is therefore implementing the resource adapter. It has to react to triggers on channels as shown in table 4.1.

Table 4.1: The database channels a robot listens to and the associated actions. The ID is a unique identifier for the robot.

Channel	**Action**
global_start	Unblock the motors and accept incoming tasks
global_stop	Abort all tasks and block the motors
start_robot[ID]	Unblock the motors and accept incoming tasks
stop_robot[ID]	Abort all tasks and block the motors
new_task	Query the database for task information
cancel_task	Query the database for task identifier of the canceled tasks

4.3.2 Solutions

There is no off the shelf solution for a whole robot control software. This is mainly due to the fact that the types of robots differ a lot. Furthermore, a robot control software usually has to fulfill special requirements and is depending on the skill set of the robot.

4.3.2.1 Goal Management

To pass goals to the task executing modules the ROS framework offers unidirectional messages, bidirectional services and the actionlib with its state pattern shown in figure 3.4. All are introduced in section 3.1.1.

A solution based on raw messages has a large implementation overhead, because delays in the message transmission and message collection have to be caught.

Services allow a direct evaluation of the result of a call. The drawback of this approach is the fact that services are thread blocking. This means that a single threaded robot control software can not react to any other events like other incoming goals. Furthermore there is no direct way to cancel a service call.

The actionlib implements a complete task coordination system. It allows to add, monitor, cancel and update goals as presented in [LPP+11]. Therefore an implementation based on the actionlib has been chosen.

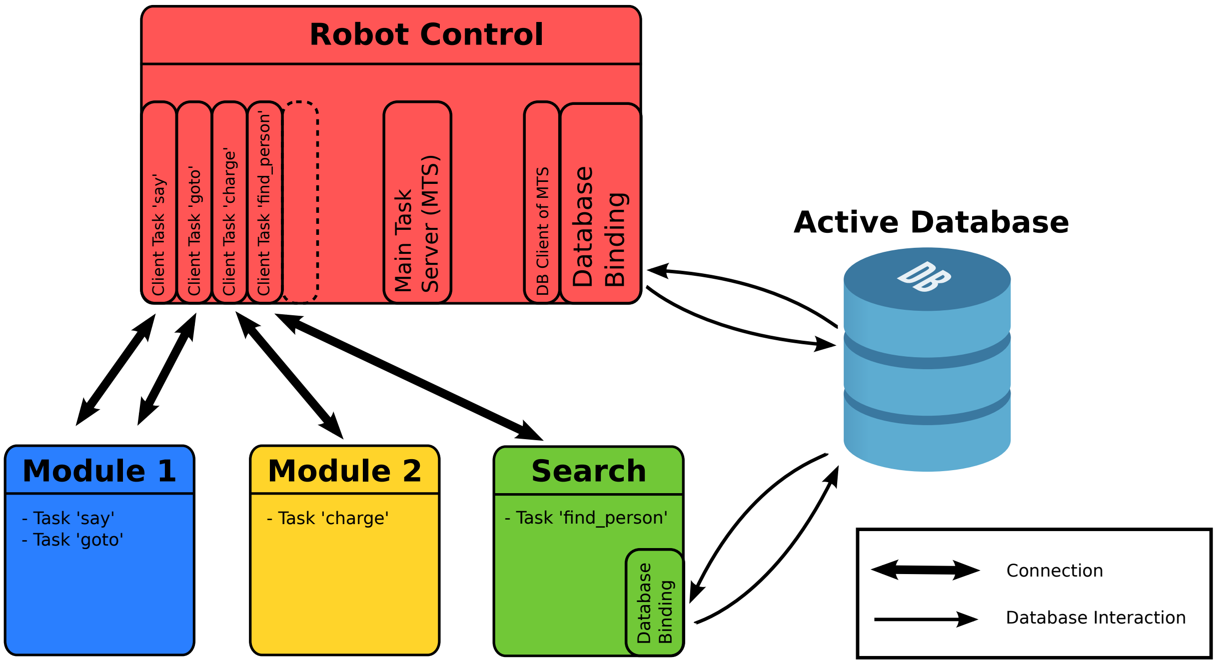

Figure 4.4: The design concept of the robot control software. Module 1 can execute two different tasks. Module 2 offers one task. The module 'search' can execute one task and has an own database connection.

4.3.3 Implementation

The overall concept as shown in figure 4.4 consists of one robot controller, one existing active database and several modules which can be started and terminated during runtime. The software of the robot controller is decoupled from the code of the modules. This makes it possible to load any module without a recompilation or a restart of the controller. A utility library is provided for the implementation of modules see section A.1.1.
The robot controller consists of the Main Task Server (MTS), the database binding and the client of the MTS for the database. It is enhanced by a client of every registered implementation of a task.
Every goal has to pass the Main Task Server in order to be executed. Whenever the robot is idle or done with the last goal the robot controller sets the next goal respecting the priorities of the pending goals.

4.3.3.1 Modules

A module is a program running on the robot and implementing the execution of a specific task. For example the module which can search for a person is called 'search' and implements the task 'find_person'.

A module can implement several tasks and has an own task server for every implemented task. These servers accept goal objects with the same definition as the Main Task Server. Modules that need a database connection can build up their own connection. The necessary connection information has been stored in the parameter server by the robot controller and can be accessed by every module.

4.3.3.2 Definition of the Generic Goal Object

The goal object is a part of an action of the actionlib which is explained in 3.1.1. The definition of the goal object can be seen in listing 3.3 and is generic in order to support all kind of tasks. It is not possible to deliver any information for the task execution by this object. The feedback message contains a percentage and an additional string for debugging. The result of a goal consists of a boolean representing the success and a string for debugging.

If an external module needs additional information or needs to report results, the database can be contacted using the unique ID of a task and the defined Application Programming Interface (API) between this module and the database. This design was chosen because it is not possible to cover the needs of strongly differing tasks in one object definition. For example a task 'charge' would need an integer to specify the aimed battery state the task 'say' needs information stored in a string and a more complex task like 'find_person' needs a full set of parameters.

Instead of trying to focus on a never sufficient design of a goal object it has been chosen to simplify the process of building a database connection with every module that needs to exchange information.

4.3.3.3 Goal Handling within the Controller

Goal Storage In section 3.1.1 it is explained that on submission of a new goal to a task server a server goalhandle and a client goalhandle for that goal are created. As these objects ensure the access to a goal they are always stored on the server and the client side.

If the database client of the Main Task Server submits a new goal, the goalhandle of the client side is stored in the list of 'all database goals' and the goalhandle on the server side is saved in the list of 'all submitted goals'. This can be seen in figure 4.5 for goal #1 and goal #2.

Whenever a stored goal of the Main Task Server is chosen as the next one to execute it is sent to the external task server of that task type. This creates another pair of goalhandles. The one on the external server is used to execute the goal. The client goalhandle which is created on the side of the robot controller is saved together with a pointer to the server goalhandle of the same goal at the Main Task Server. This can be seen in figure 4.5 for goal #1 whereas goal #2 is not running yet. This storage ensures that status feedback of the executing task can be passed to the client of the Main Task Server.

Goal State Policy of the Main Task Server In figure 3.4 the different states of goals can be seen. The most important ones are:

- PENDING - if a submitted task has neither been accepted nor been rejected
- ACTIVE - if the goal has been accepted
- REJECTED - if the server rejected the goal
- SUCEEDED - if the goal suceeded
- CANCELED or ABORTED - if either the client or the server terminates a goal

All incoming goals start in the state PENDING. When the Main Task Server receives a goal it is checked if this type of task, e.g. 'jump_on_table' can be executed by one of the modules. If that requirement is not fulfilled the goal is rejected, otherwise the goal stays pending and is added to the list of goals. When the goal is chosen for execution it is sent to the external task server of the corresponding module. The state is changed to ACTIVE and after that the state depends on the feedback and result of the external server.

4.3.3.4 Module Management

Modules can register at runtime at the robot controller using the message shown in listing 4.1. Based on the content of the message the robot controller builds up, updates or terminates a connection to an external task server. If the connection can be established the unique task name of the new task is added to the list of possible tasks and reported to the database. That way the AAL system always has a list of tasks which the robot can perform. If a deregistration is requested the controller terminates the connection, deletes the task type from the list of possible task. Furthermore the database is informed and all pending goals of that type are rejected.
In order to make sure that an external task server still responds are all connections frequently checked by the controller.

4.3.3.5 Developing new Modules

On the code level external modules always depend on the robot control package. Implementing a new module can either be done by fulfilling all requirements or using the class 'RobotControlSimpleClient' as parent class. This class as shown in A.1.1 implements and initializes all necessary structures to build a module which can offer the execution of one task type. It allows the fast creation of own modules without the overhead of knowing about the underlying structures and their initialization. Both existing modules use this class.

```
# The unique identifier of the task
string task_type
# The full name of the actionserver
string task_server_name
# true to register
# false to unregister
bool reg
```

Listing 4.1: Message definition for registering or deregistering of a task type at the robot controller

4.4 Database Connection

4.4.1 Requirements

The reference implementation of the active database architecture as explained in 2.3 is running in a PostgreSQL database. PostgreSQL is an open-source database software implementing the Structured Query Language (SQL) standard.
The robot as a resource adapter should be able to build a connection to the database and perform the following tasks:

1. Listen to channels
2. Receive notfications and extract information
3. Call functions in the database
4. Receive and process a table in return

Channels are unprotected tunnels and every connected client can subscribe to every channel. They can be triggered by any client of the database. If a channel is triggered every subscriber receives a notification consisting of:

- The name of the triggered channel
- The sending Process Identification Number (PID)
- An optional payload string

In this architecture the name of the channel and the notification event itself are used for IPC between the database and the clients.

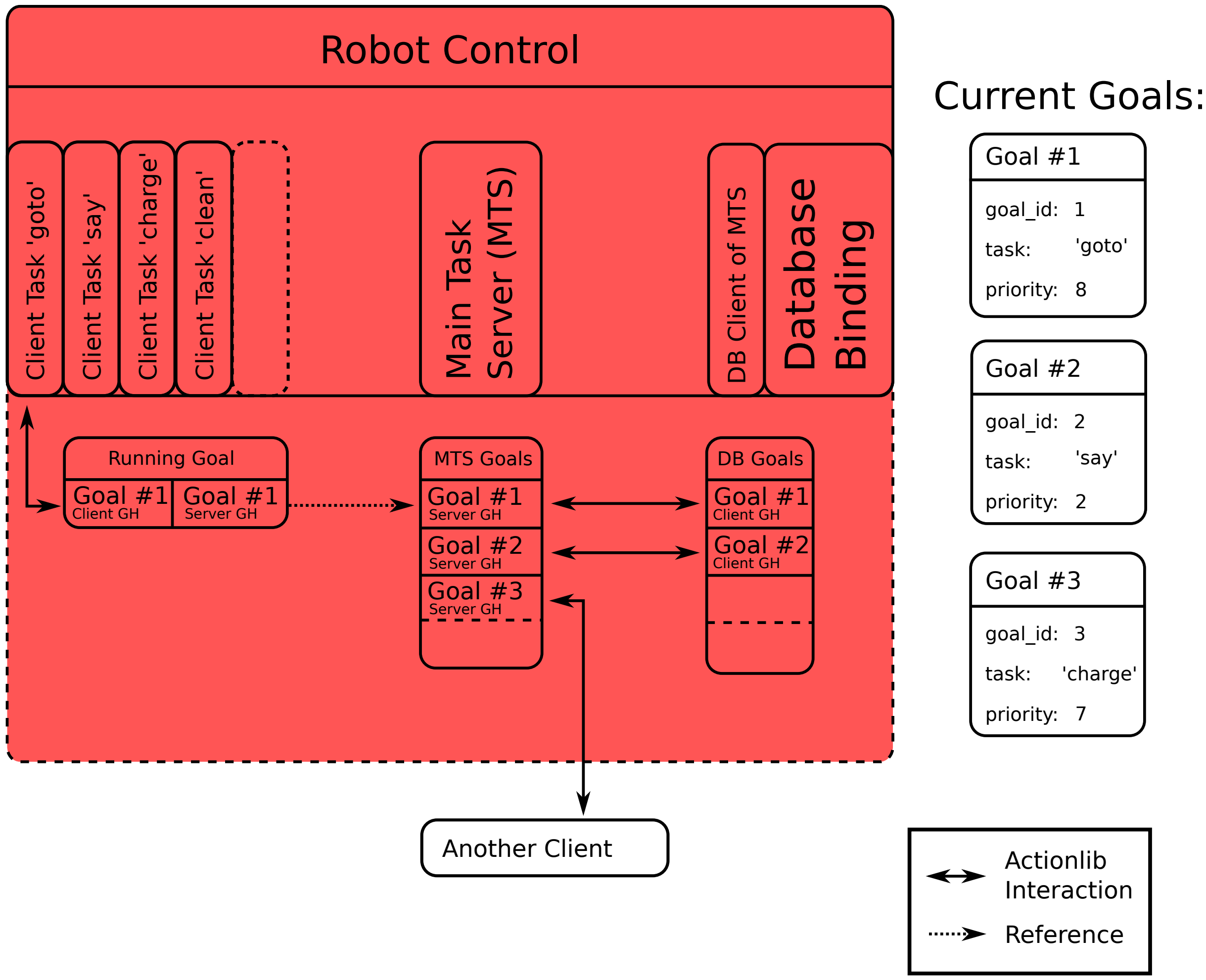

Figure 4.5: Detailed view on the robot control structure. The upper half shows the server and clients and the lower half the corresponding data structures for the goal storage. The database submitted two goals and another client the goal #3. Goal #1 is running on the task server of the task 'goto'.

4.4.2 Solutions

The robot software for this thesis is written in C++. Therefore just corresponding clients are evaluated.

4.4.2.1 Official PostgreSQL C Client

The PostgreSQL project provides a client for the programming language C called 'libpq' [lib14]. The package is well documented and supports a big set of database features. Its functionalities go far further than the requirements. The usage of the client would come with an additional learning overhead for everybody implementing modules.

4.4.2.2 ROS Package SQL-Database

The ROS package sql_database 'provides an easy to use and general interface between a SQL database and object-oriented C++ code, making it easy to encapsulate the conceptual 'objects' contained in the database as C++ classes.' [ros14g]

It is built on the official library 'libpq' and abstracts the functionalities. It is capable of inserting data into tables of a database and reading data of tables. It is well integrated in ROS and comes with tutorials.

Eventhough the functions for listening to channels, receiving notifications and calling functions are not yet implemented it has been chosen to use and enhance this package. It will allow module programmers an easy interaction with the database.

4.4.3 Implementation

The existing implementation of the ROS package 'sql_database' [ros14g] was lacking support for the following needed functionalities:

- Listen and unlisten to channels
- Retrieve notifications from channels
- Call functions in the database and process the results

The package was enhanced by these capabilities.

To receive notifications from an attached channel, incoming notifications have to be actively collected. The implementation allows to choose between two different functions for the collection of notifications.

1. **checkNotify** - checks for notifications and returns immediately
2. **waitForNotify** - waits for activity at the socket and exits on a connection error or a received notification

The capability to call UDF of the database has been derived from the existing functionality to read from databases. In listing 4.2 the code for a typical database query can be seen. The object 'tasks' of the type 'returnTasks' will store the returned table and allows an easy access.

For every call an object according to the expected columns has to be defined beforehand. Such a definition can be seen in listing 4.3

```
ROS_INFO("Getting new tasks");
std::vector< boost::shared_ptr<returnTasks> > tasks;
database_interface::FunctionCallObj call;
call.name = "get_tasks";
if (!database_->callFunction(tasks, call))
{
  ROS_WARN("Calling gettasks failed. Probably the connection dropped. Exiting.");
  return false;
}
```

Listing 4.2: Code to call a function of the database and receive a table of tasks in return. 'returnTasks' is the object type receiving the table (see A.5.3) and 'database_' is the object holding the connection (see A.5.2)

```
#include <string>
#include <vector>
#include <database_interface/db_class.h>

class returnTasks : public database_interface::DBClass
{
  public:
  database_interface::DBField<int> id_;
  database_interface::DBField<int> task_priority;
  ...

  returnTasks() :
  id_(database_interface::DBFieldBase::TEXT, this, "key_column", "places2", true),
  task_priority_(database_interface::DBFieldBase::TEXT, this, "task_priority", "places2", true),
  ...
  {
      primary_key_field_ = &id_;
      fields_.push_back(&task_priority_);
      ...
  }
};
```

Listing 4.3: Definition of the class 'returnTasks' used in 4.2 (shortened to the unique serial key and one column). The column name as well as the data type have to be mentioned for every column.

4.5 Person Detection

4.5.1 Requirements

The robot has to find a person in order to fulfill the scenario. As mentioned in the conventions in section 1.4 the search for a person should not rely on the existance of other external sensors. This means that the robot should be able to carry all sensors needed for the chosen methods and furthermore equip the computer performance to use them.
It can not be assumed that the person is standing and looking towards the robot. The case of a person lying on the ground should be covered as well.

4.5.2 Solutions

In figure 3.2 it can be seen that the depth camera of the Turtlebot has a low mounting height. This is necessary to avoid the navigation and mapping problems mentioned in 4.2.2.1. With the opening angle of the Kinect camera of 45.6 deg [ki-14] this means that the robot has to be more than 3 m away from a 1.90 m standing person in order to be able to detect a face or a full torso.

4.5.2.1 Method using 3D data

In [CMBV13] a fast detection process has been introduced. The method allows to detect humans based on depthimages even if the body isn't fully visible.
But eventhough the results are good it is not possible to integrate this method within the given timeframe.
Furthermore skeleton trackers like the 'openni_tracker' [5] or the 'Voodoo' tracker [KVSD06] need an intialization pose which does not fit the requirements of a surveilling robot.

4.5.2.2 Face Detection

Face detection can be a good identifier if the person is looking towards the robot. There exist two different packages for the ROS framework. The drawback of both packages is that they use a Haar detector to find regions of interest. These detectors are usually trained for a vertical face and can not detect the face of a lying person.

Care-o-Bot People Detection The Care-o-Bot project developed a complete face detection cascade presented in [BZF⁺13] and [det10]. It can do face detection and face recognition using a the 'Fisherface' or 'Eigenface' approach.

[5] openni_tracker on Github: `https://github.com/ros-drivers/openni_tracker`

As shown in figure 4.6, the software is using a Haar detector on depthimages [BZF+13] to find a head and passes areas of interest to a face detector. If a face is detected, the face identification assigns a name from the trained database for the label 'Unknown'. The additional usage of depthimages leads to a low false detection rate. The training of new persons is easy and can be done manually or even at code level. The package is integrated into ROS and ready to use.
Because it is interesting for an AAL system to address people personally and to distinguish and identify them this package has been chosen.

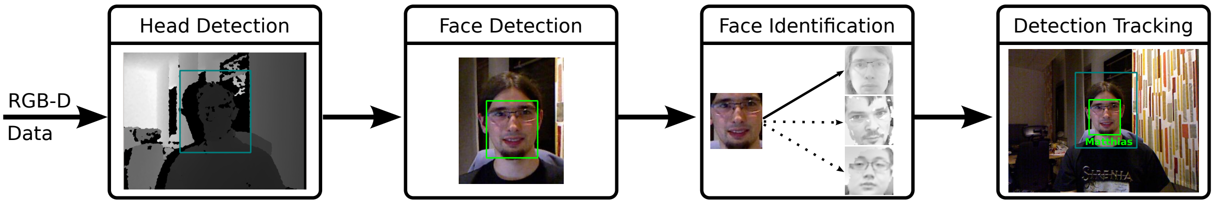

Figure 4.6: The simplified pipeline of the 'cob_people_detection'

Pi Face Tracker In this package a Haar detector for face detection [Goe] is used. Found faces are tracked using 'Good Features Track' and the 'Lucas Kanade Optical Flow' tracker. Additional depth data can be used to lower false detections and to improve the tracking.

4.5.2.3 Map based detection

If a person looses consciousness there is a high probability that the person falls or sits on the ground. When the robot is passing by it will see this spot as an additional obstacle. This information can be used to estimate if it could be a human person and use human robot interaction to verify the assumption.
This approach will cause false detections for replaced items. But in contrast to other methods it does not need a lot of computation and could have a high probability to recognize a person as an obstacle. As the robot is part of an AAL system it can pass information, for example images, about these spots if the search does not succeed. The system can transfer these pictures to relatives of the person or personal of the care facility to evaluate them.
As recognition of a fallen human is not a trivial task this method has been chosen to find a lying person. Furthermore it can be enhanced by an infrared sensor.

4.5.2.4 Infrared Sensors

Infrared sensors can measure the heat radiation emitted by objects. In [SSC+13] it is shown that an array of infrared sensors can significantly lower false detections of humans. An alternative to an array of sensors could be a single infrared sensor used to measure the temperature of interesting points like new objects.
In contrast to the results is the price of an infrared sensor array.Such an array would higher the price of a personal robot significantly. The prices for a single infrared sensor are reasonable, but the usage of these sensors comes with a high effort of integration. The sensors have to be connected to the analog or digital port of the robot and an own software has to process the incoming data.
Therefore it has been chosen not to use an infrared sensor.

4.5.3 Implementation

The implementation is designed to be independent from the robot specific and task specific parts in order to allow the reusage in other projects. This package does neither directly interact with the robot nor with a person. It observes and outputs information about detections.
The chosen approach to detect a person based on the additional appearence on the map must be fully implemented. The face recognition is a fully running external package. For this method the handling of the detections had to be implemented.

4.5.3.1 Map Approach

The goal of this approach is to identify and rate obstacles which do not appear on the static map used for navigation.

Incoming Data The software uses information from two sources:

1. Currently seen obstacles by the navigation software
2. The local costmap of the navigation software

A visualization of the data can be seen in figure 4.7. The currently seen obstacles are passed as points in absolute map coordinates. The local costmap is a small map anchored in the base frame of the robot. It is used to calculate the local path for navigation [ros14d]. On this map every point is either

- occupied,
- free space or

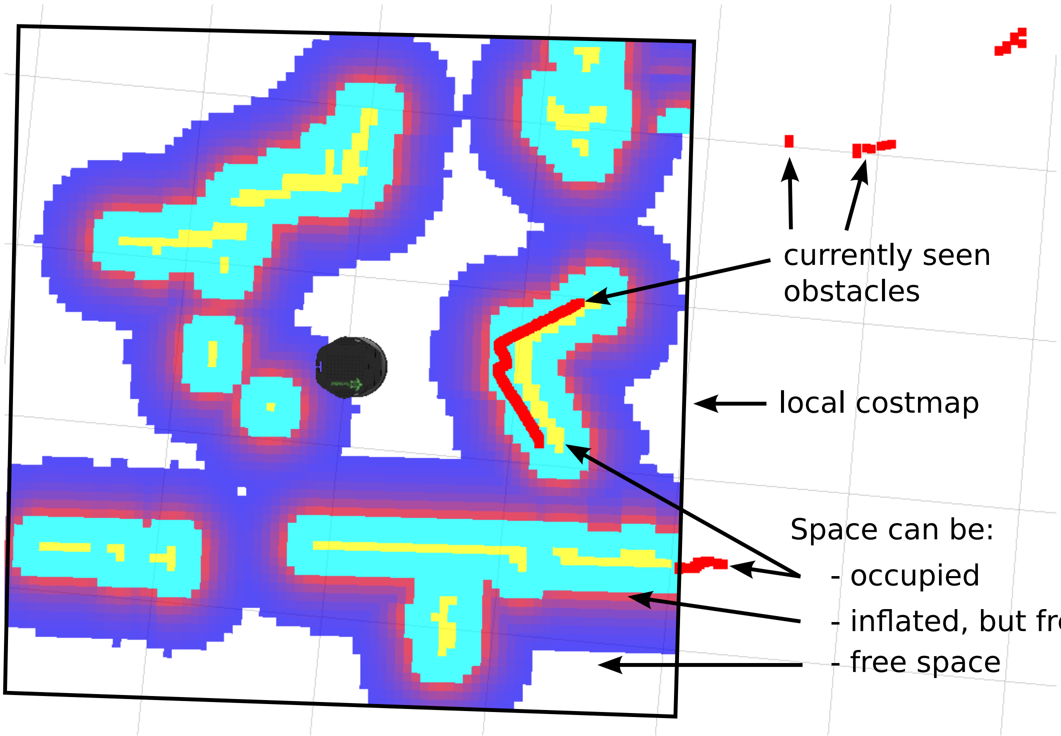

Figure 4.7: Visualization of the incoming data in RViz. The local costmap around the robot is marked by a rectangle. Occupied points are shown in yellow. Free space is white and all other colors are information for the navigation cost function but represent free space [ros14c]. The currently seen occupied points are marked in red and go further than the costmap.

- inflated by a cost function, but free space.

This means that input from the local costmap can be used to clear false detections while this is costly to compute with the incoming data of the currently seen obstacles.
Whereas the information of the local costmap can always be used, the quality of the information of seen obstacles is strongly depending on the distance and the twisting speed of the robot. Therefore data from this source is not used while the robot is turning and for points which are more far away than 4 m.

Data Storage Map data is stored in a costmap object provided by the navigation package of ROS [ros14c]. It provides an easy access to the stored occupancy information and can do the transformation between the internal array and map coordinates. For every point of the map an 8 bit integer is used to store occupancy information.
To find and rate obstacles five map objects are used:

- **Inflated Static Map:** A copy of the static map inflated by 10 cm

- **Updated Map:** A map only representing the actual situation
- **Count Map:** It stores how often a point has been marked as occupied
- **Distance Map:** A map storing the closest distance from which a point has been marked as occupied
- **Difference Map:** A map representing all occupied points of the updated map which are not occupied in the inflated static map

In the distance and count map the integer of the map representation has been used to store the amount of sightings and the closest distance in decimeter. The static map is the map used for navigation.

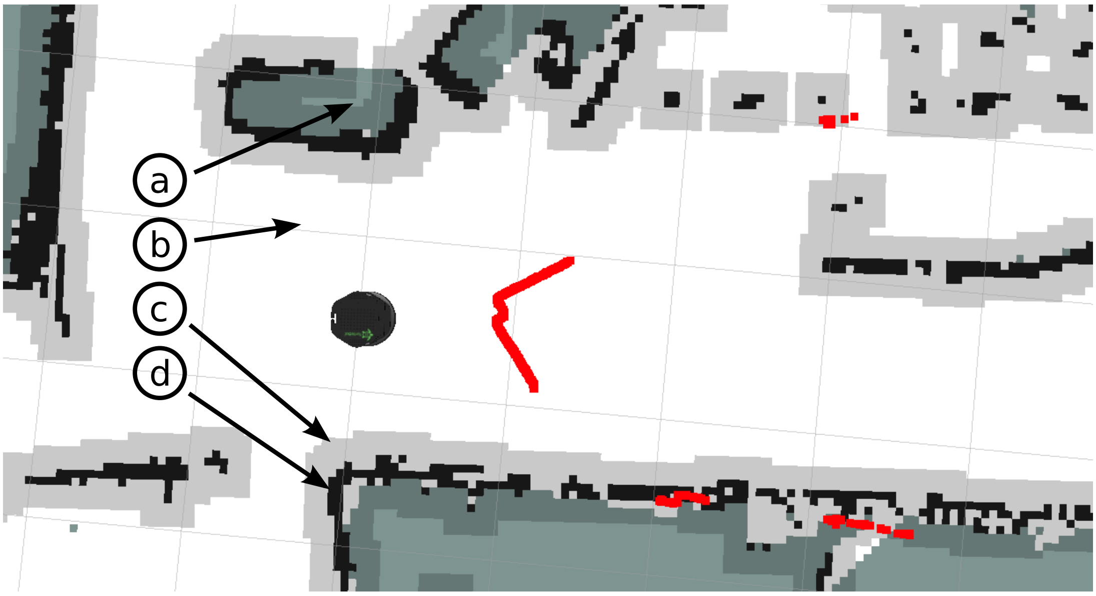

Figure 4.8: The static map and the inflated map overlayed in RViz.
a) Unknown space in the static map
b) Free space in both maps
c) Occupied space in the inflated static map
d) Occupied space in the static map

Process Flow In figure 4.9 the flow of one process cycle is shown. First the incoming data is inserted into the frequently updated map, the count map and the distance map. Then a difference map is calculated. Every point which is occupied in the updated map but marked as free in the inflated static map is an additional occupied point and set as occupied in the difference map.
At the end of every cycle the list of obstacles is published for other nodes.

Rating of Obstacles Every known obstacle is rated for its probability to be a human. This rating is based on three factors:

1. Average sighting distance
2. Average amount of sightings
3. Its size

In total an obstacle can score between 0 and 100 points. Each factor can influence one third.

Sighting distance The sensor data of the Kinect looses accuracy on higher distances. Therefore the average sighting distance for each obstacle is calculated and rated between 0 and 100 points using the function[6]

$$p_{rd} = p_d \cdot \left(-\frac{10}{3}\right) + \frac{400}{3}.$$

Whereas p_{rd} is the rating and p_d is the average distance of the obstacle in decimeter. As sensor data with a distance of more than 4 m is not used, the function never returns a negative value. The lowest distance a point can have is 1 m if the point appears on the local costmap.

Amount of sightings As mentioned in 4.5.3.1 the costmap can store values between 0 and 255. The function[6]

$$p_{ra} = \frac{100 \cdot \sum_{i=1}^{p_{size}} p_i}{255 \cdot p_{size}}$$

rates an obstacle between 0 and 100 points based on the amount of appearances. p_{ra} is the rating, p_{size} is the amount of points an obstacle includes and p_i is the number of sightings of each point stored in the count map.

Size The third factor in the rating of an obstacle is the size; respectively the amount of occupied points on the map. On the used map every point represents a 5 cm · 5 cm square. But it has to be noted that the robot usually just sees the border of an obstacle. For example the obstacle in front of the robot in figure 4.8 is an arm chair. The rating is

- $0 \leq p \leq 5$: 10 points,
- $5 \leq p \leq 10$: 50 points,

[6] Both function are self designed

- $10 \leq p \leq 20$: 75 points and
- $p > 20$: 100 points

whereas p is the number of points.

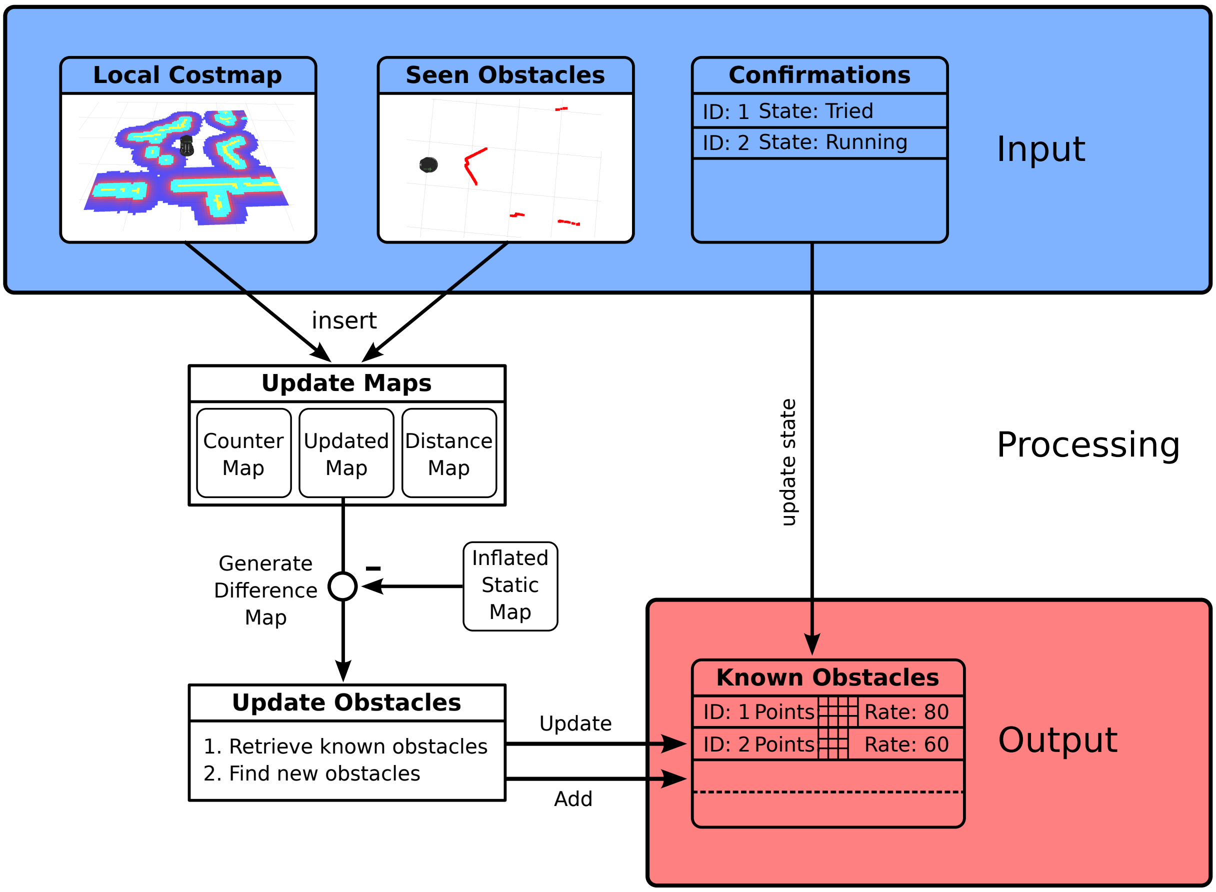

Figure 4.9: Process flow of one cycle in the obstacle approach

4.5.3.2 Integration of the Face Identification

The Care-o-Bot people detection software is a fully implemented face recognition. This package is developed in the older buildsystem 'rosbuild' whereas the new packages are developed in the newer buildsystem 'catkin'. As 'catkin' packages can not depend on 'rosbuild' packages, the header files for the messages had to be copied to an included header directory of the new package.

The face recognition software has several outputs of which the topic for tracked faces is used as input. The message is shown in listing 4.5.3.2.

The integration of the face recognition mainly implements four functionalities:

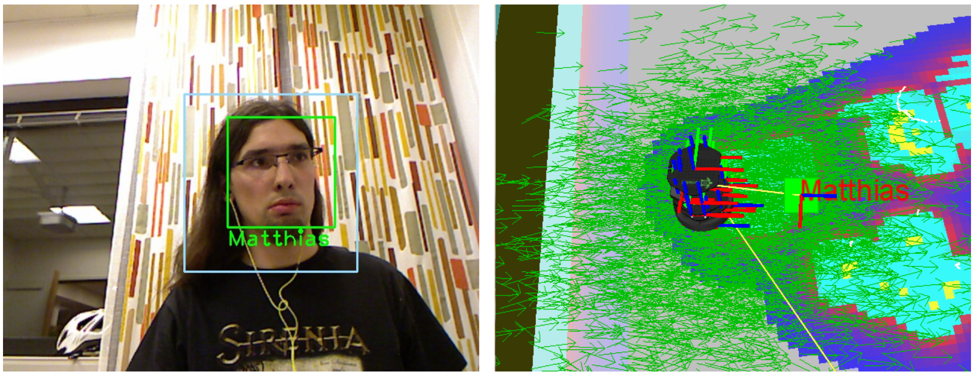

Figure 4.10: The left picture shows a detected head (outer blue rectangle) and the detected face within the head (inner green rectacle) with the assigned name. On the right picture the published coordinate frame as well as the visualization in RViz can be seen. (On the ground the local costmap as well as the localization assumptions are displayed)

1. Coordinate transformation to map coordinates
2. A simple tracking of recognized faces
3. The capability to assign several names to a recognition
4. The possibility to do an external verification e.g. by using human robot interaction

```
# Sequence number and timestamp
std_msgs/Header header
# Name of the person or 'Unknown'
string label
string detector
# The score of that label
float32 score
cob_people_detection_msgs/Mask mask
# the position and orientation
geometry_msgs/PoseStamped pose
```

Listing 4.4: Detection message of the cob-people-detection package (taken from [git14] and commented)

The pose represents the position of the face in the coordinate frame of the camera.

Coordinate Transformation First the position of every face is published as a coordinate frame in relation to the camera. This can be seen in figure 4.10. To allow easier debugging also the label of the recognition and a cube at this position are published for RViz.

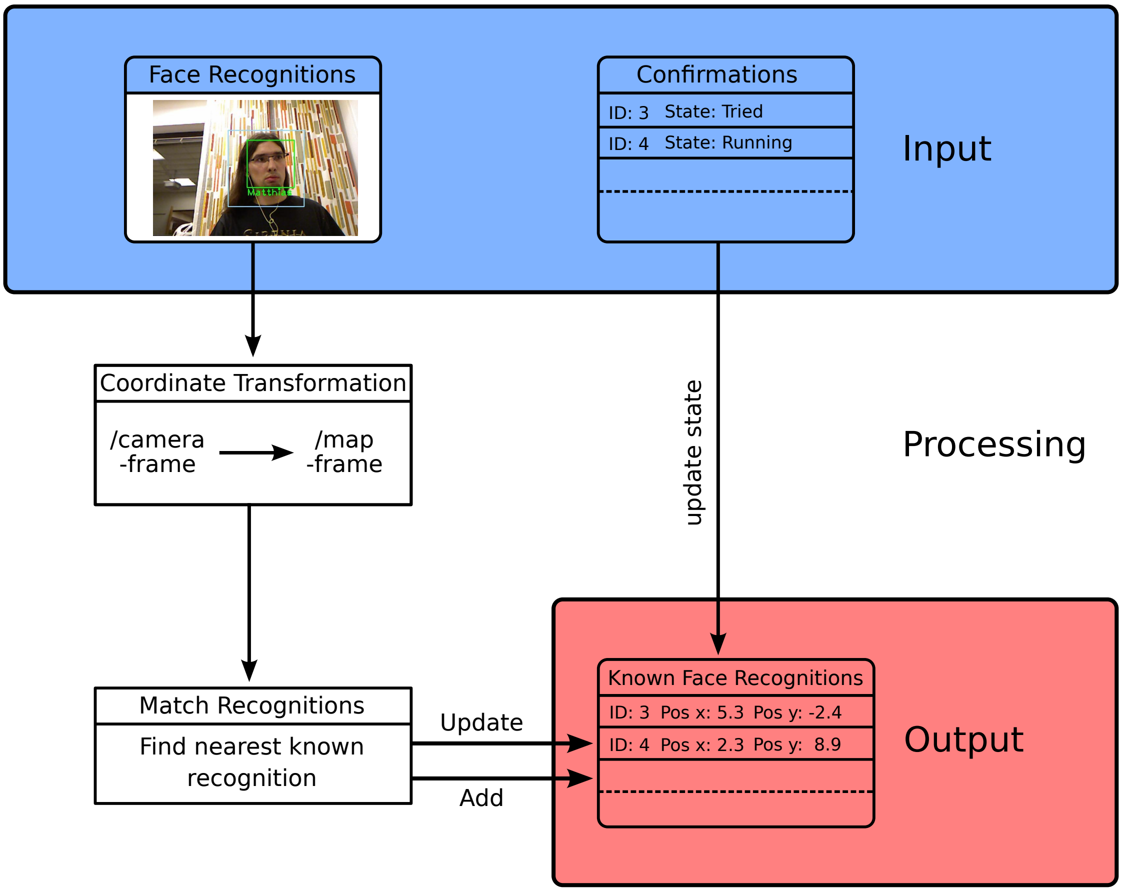

Figure 4.11: One cycle in the face detection.

Then the published coordinate frame is used to do a transformation between the detection and the map coordinate frame which can be seen in figure 4.12. The output of this transformation are the map coordinates of the detection.

Information Storage The assigned name of a face recognition should not depend on a single result. Furthermore a global storage on the whole map requires a tracking of recognitions. Therefore it is necessary to store and fuse information of recognitions. As shown in figure 4.11 incoming detections are used to update a known detection or add a new detection. The list of known detections covers all currently known detections.

Match Recognitions Every incoming detection is assigned to the nearest known recognition using a modification of the Hungarian Algorithm. If the nearest known detection is more than 0.5 m away, a new detection added. Otherwise the nearest one is updated with the name, position and time.

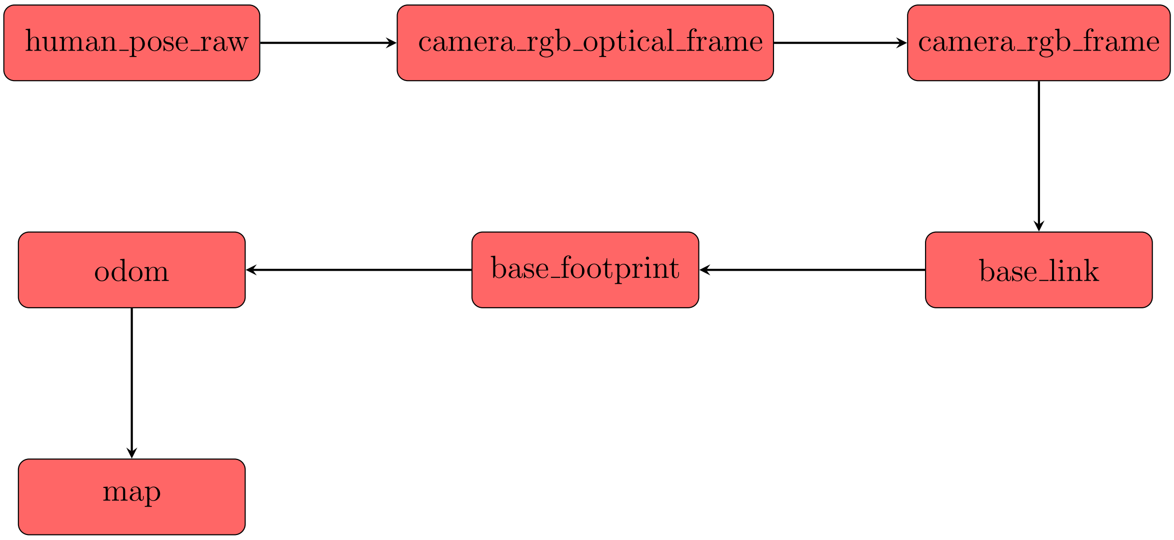

Figure 4.12: Transformation chain between a detection and the coordinate frame of the map.

Multiple Assigned Names A known detection can have multiple names assigned if the face identification outputs different names for the same person. Every time a known detection is updated with a name, a counter for that name is incremented. For example for the visualization in RViz the name with the most appearances will be displayed.
Furthermore, if a name, e.g. 'Matthias' is assigned to a detection, all other known detection on the whole map having the same name assigned loose one point on the counter for this name. This solves two issues:

1. older detections will loose points for this name and sooner or later be deleted
2. wrong identifications of that name will loose points

If a known detection does not have any more names assigned and is older than a specified time it is deleted by a garbage collector.

4.5.3.3 External Confirmation

The robot can use text to speech as well as speech recognition. This allows a confirmation of detected obstacles and face recognitions. As the person detection software is designed to be a passive component an external software has to carry out the confirmation and has to decide which detections are going to be confirmed.
If an external program start or updates a confirmation it can inform the person detector about the process using the message shown in listing 4.5.3.3. The person detector updates the known detection using the information of the confirmation. A detection can have the

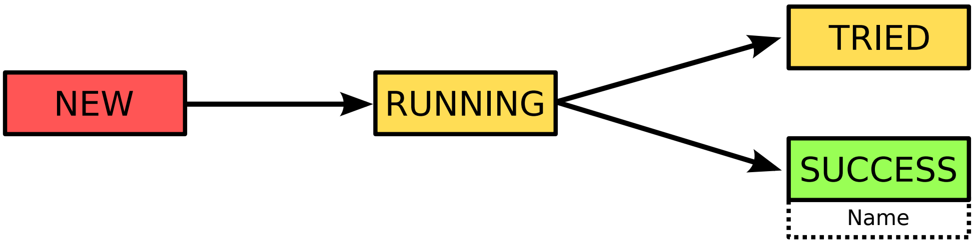

Figure 4.13: The states of obstacle and face detections. First all detections are unconfirmed, then an external process can report an ongoing confirmation. This can result in success or fail. A successfull confirmation can assign to a detection.

states shown in figure 4.13. If a confirmation was successful, this name is removed from all other known detections.

```
Header header
int32 id          # ID of the detection
bool running      # true if the confirmation has been started
bool tried        # true if the confirmation has failed
bool suceeded     # true if it suceeded
string label              # name to assign
time latest_confirmation  # time of the confirmation
```

Listing 4.5: Confirmation message used to inform the person detection software about external confirmation of a detection

4.6 Module for Search Coordination

This module executes the search for a person. Whereas the person detection software described in section 4.5 is collecting information, this module is reasoning about it.

4.6.1 Requirements

As shown in figure 4.1 this is connected to

- the database,
- the robot control,
- the human interface,
- the person detection and the
- navigation software

but also needs to take care of the exchange of information.
The reasoning about the order of the navigation goals and the detections should be done in this package. Furthermore it is deciding if a detection should be confirmed using the human interface component and takes care of the conversation with the person.
It is a module of the robot control software as described in section 4.3.3.1 and offers the task 'find_person'.

4.6.2 Solutions

During a search for a person, the robot has to perform a lot of different subtasks like doing a 360° turn or waiting for the photo to be taken. The different states can be defined in a state machine.

4.6.3 Implementation

The states and transitions can be seen in figure 4.14. The states are explained in table 4.2.
On startup the module registers the task 'find_person' at the robot control. If a new goal of this task is sent to the robot control it will be forwarded to this module. The module is inactive as long as there is not a running goal.

Initialization If a new goal comes in the database is queried for a list of places to explore. Each of the delivered places is added to a list of exploration goals and the state machine is started by setting the first goal.

Navigation Goal Storage The module has three different kind of goals:

1. Exploration goals
2. Face recognition goals
3. Obstacle goals

All are stored in the same format but lead to different behaviour which can be seen in figure 4.14 showing the state machine. Every goal can be marked as done to allow to exit the state machine if all goals have been reached.

Obstacle and Face Recognition Goals Whereas exploration goals are provided by the database, obstacle and face recognition goals are calculated from the known recognitions and known obstacles provided by the person detection package.

As described in 4.5.3.1 every obstacle is rated and can be present or unpresent. In this module a threshold is defined to distinguish between interesting obstacles and ignored obstacles. In order to be added as an navigation goal an obstacle has to be present and rated higher than the threshold. A second threshold allows to delete navigation goals if the corresponding obstacle gets a lower rating or if it was a false detection.
The navigation goals for both obstacle and face recognitions are calculated to be 1 m in front of the spot.

Ordering of Goals The goals are ordered following the rule:

1. all face recognition goals
2. all obstacle goals
3. all exploration goals

This way upcoming face recognitions and obstacle detection preempt exploration goals which usually leads to a faster search.

Visualization The whole process of a search can be monitored using the visualization tool RViz. In this tool

- the order of the goals
- the places of the calculated navigation goals and
- the state of goals

are shown. This can be seen in figure 5.6.

4.7 Human Interface

4.7.1 Requirements

The robot should be able to interact with a human in order to ask for its wellbeing. Therefore text-to-speech as well as speech recognition is required. But as the human robot interaction is not a main part of thesis, a basic setup is sufficient.
The system has to support to ask questions and to process the results. Furthermore, it should offer a simple interface to use text-to-speech.

Table 4.2: States of the search process. The state machine can be seen in figure 4.14

Shortcut	Explanation
START	Received a new goal from the robot control
FINISH	Done with the goal to search for a person
SET GOAL	Set the next goal in the list of ordered goals as navigation goal
FACE	Navigate to a detected face
OBS.	Navigate to a detected ostacle
EXPL.	Navigate to a place which should be explored
CONF.	Do a confirmation of an obstacle or detected face
PHOTO	Wait until a photo is taken and safed
PANO	Do a 360° turn
FOUND	Ask the person for its wellbeing

4.7.2 Solutions

4.7.2.1 Speech Recognition

A list of speech recognition software can be seen at [spe14]. From the large amount of different solutions, two got a closer evaluation.

Hark 'Hark' is an 'open-sourced robot audition software' [har14]. It has mainly three functionalities:

1. sound localization
2. sound seperation
3. speech recognition

The 'Hark' project offers source code and binaries for the used version of ROS. But as a simple setup is enough to solve the scenario, this software could be included in the next development step.

Pocketsphinx The ROS package pocketsphinx offers an integrated and easy to use speech recognition within the ROS framework. It is based on CMU Sphinx [7] and needs a dictionary file and a language model. Both are delivered in a default installation, but are limited to about 100 words [ros14f]. This package has been chosen, because it is a convenient solution for a ROS based system.

[7] Website of CMU Sphinx: `http://cmusphinx.sourceforge.net/`

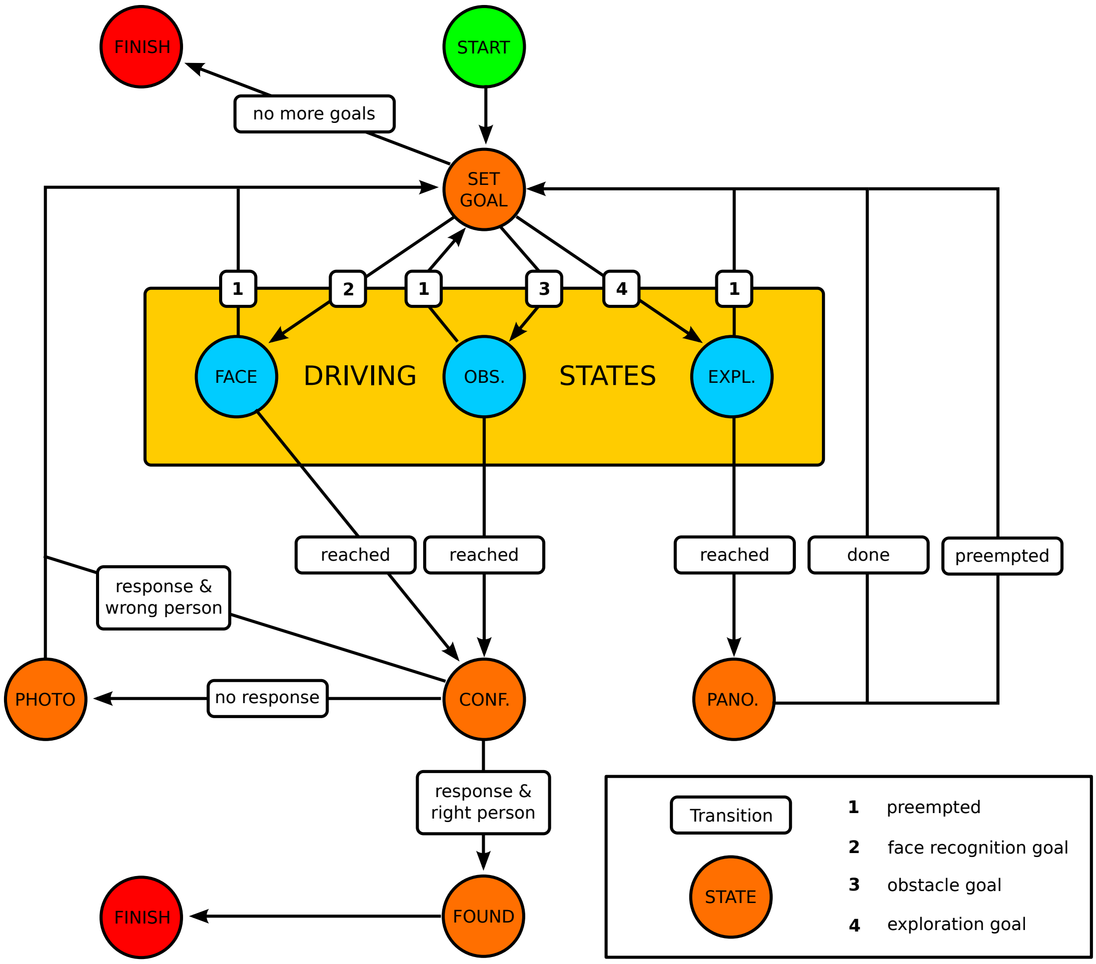

Figure 4.14: The state machine of the search process. The states are explained in table 4.2.

4.7.2.2 Text to Speech

Text to speech engines have a high price range starting from free open source solutions like Festival[8] going up to commercial products[9].

As a basic solution is sufficient for this thesis, the text to speech engine Festival has been chosen. It is well integrated into ROS by the 'sound_play' package[10]. A drawback of this solution is the lacking feature to track the end of an text to speech output. An advantage is the possibility to play soundfiles from the filesystem.

[8] Website of the Festival project: `http://festvox.org/festival/`

[9] Website of Ivona: `http://www.ivona.com/en/`

[10] 'sound_play' package in the ROS Wiki: `http://wiki.ros.org/sound_play`

4.7.3 Implementation

A node called 'human_interface' has been created to fulfill the requirements. It offers:

- text-to-speech handling
- a service for yes-no-questions returning the answer
- a service for the confirmation of several names

As the 'sound_play' package does not make sure that the text-to-speech output of 'Festival has ended before it transfers the next string, this capability had to be added.
Experiments showed that the average output of the default voice of Festival is 0.07 letter/s which is extended by a 1.5 s safety margin.

Yes-No-Questions The offered service of yes-no-questions is defined in the service file shown in listing 4.7.3. It outputs the question using the text-to-speech and then processes the input of the speech recognition.
The output of the speech recognition is a string for every sound snippet. Therefore these messages are enhanced by a timestamp refering to the time of arrival. After the question is pronounced the answers are processed. All sentences which have been recognized before the end of the text-to-speech output are ignored and the reamaining are searched for 'yes' or 'no'.
The person has 20 seconds to respond to the question. If there is not any valid answer, it is assumed that no one is present. This way was chosen, because even small noises are recognized as speech.

```
Header header
# the question
string question
# some things expire after some time
# mention the last time you want something to be said
time expires
---
# the status of the answer
# 0 = worked
# 1 = no answer
# 2 = wrong answers
# 3 = speaker blocked
int8 status
bool answer
```

Listing 4.6: Service definition of yes-no-questions

Confirmation of a face recognition Besides the yes-no-questions a complete confirmation of an array of names is offered. This is used if a detected face has multiple assigned names. The confirmation accepts an array of names and internally forms yes-no-questions for each of them. It returns information about the success and, if it was succesful, the right name.

Chapter 5

Results

This chapter describes the setup of the experiment for analysing the detection of persons in different positions.
It will give an idea of the working efficiency and explain the overall result of the approach to connect a robot to the AmI.

5.1 Setup of the Experiment

In order to show the abilities of the system and their behaviour in different situations a set of scenarios has been chosen to be tested.
The experiment has been performed in the facilities of the institute. The environment includes a long floor, two living room equivalent places as well as a big conference table. For the speech recognition software 'pocketsphinx' the robocub demo files were used. These files are part of the default installation of pocketsphinx.

5.1.1 Scenarios

Four different scenarios were evaluated. Their positions are shown in figure 5.1.1

1. A person sitting on a chair

2. A person lying flat on the ground

3. A person sitting at a wall

4. No person present

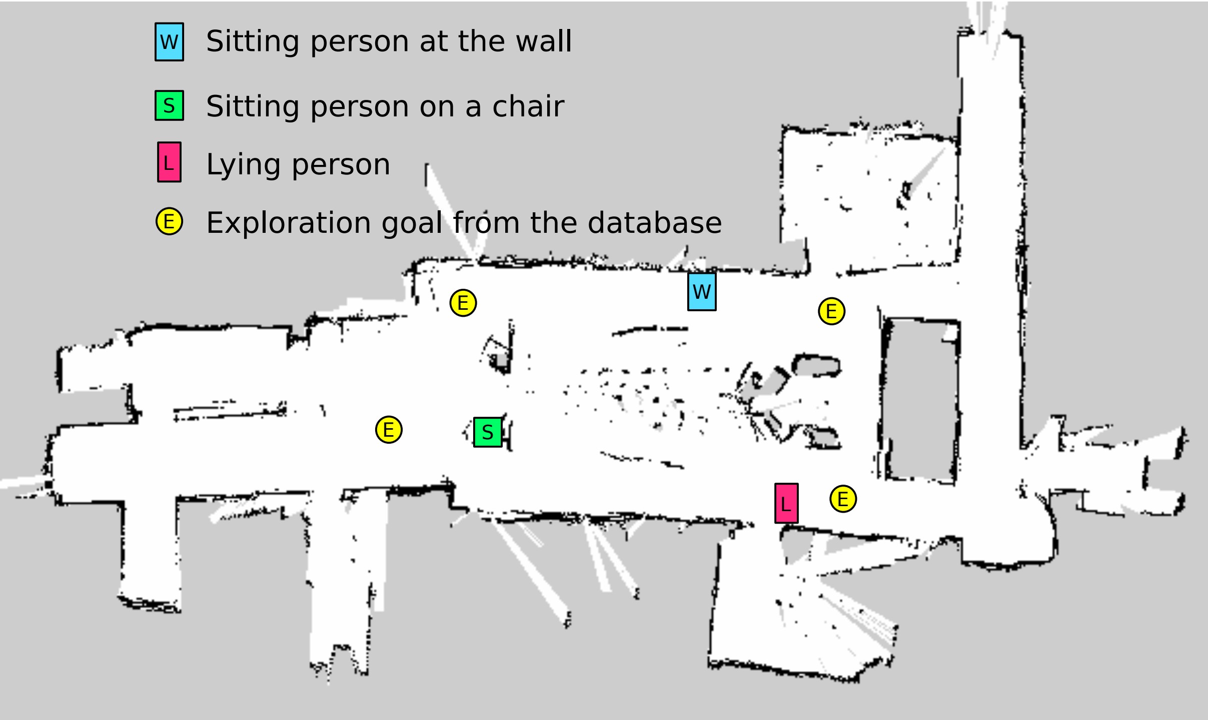

Figure 5.1: Positions of the person in the different scenarios

5.1.1.1 Chair Scenario

The setup of the scenario can be seen in figure 5.1.1.1. This scenario was chosen, because a person could have left the bed because he or she could not sleep and decided to read something before going back to bed. To fulfill this scenario the robot has to detect the person and do a confirmation.

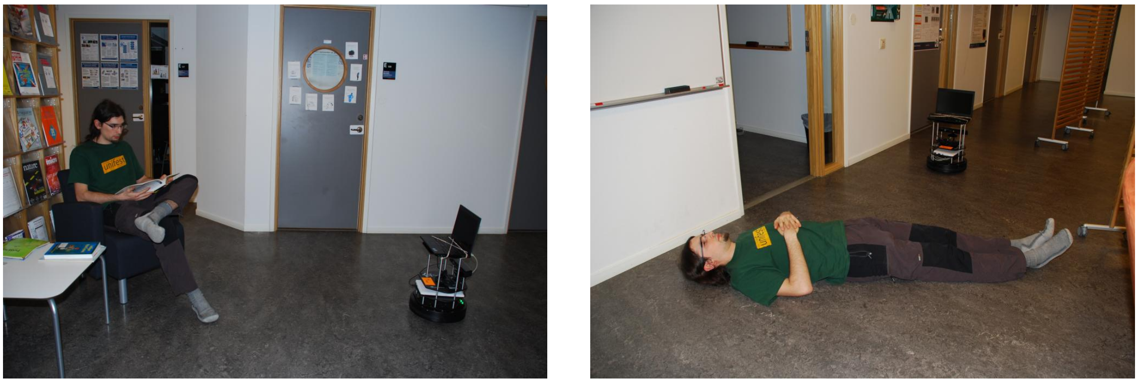

Figure 5.2: Person sitting at a chair (chair scenario) and a person lying on the ground (lying scenario)

To be able to find a person using the face detection the AAL system has to be aware of the low mounting point and the opening angle of the Kinect mentioned in section 4.5.2.

5.1.1.2 Lying Scenario

As the laserscanner can not recognize points below its mounting height as described in section 4.2.2.1 the detection in this scenario is especially difficult. The scenario is seen as fulfilled if a photo of the lying person is taken by the robot.

5.1.1.3 Sitting at a Wall Scenario

This scenario has been choosen for evaluation because the static map is inflated by 10 cm as explained in paragaph 4.5.3.1. Therefore the upper body next to a wall should not be detected by the robot. To fulfill this scenario the robot has to detect the person and do a confirmation.

Figure 5.3: A person sitting at a wall (wall scenario)

5.1.2 Hardware Setup

During the execution the whole software was running on the robot and the database was reachable all the time.
The robot was equiped with the two laptops shown in table 5.1.

Both laptops are connected with each other using an ethernet cable. As the camera is connected to the Dell laptop and the navigation software is running on the Asus laptop their clock times must concur **exactly**. On little divergence the navigation runs poorly, on slightly bigger divergence either the 'depthimage_to_laserscan'-node on the Dell or the

Table 5.1: Technical data of the laptops running the software [del14] [asu14]

Dell Latitude E4310	
Processor	Intel(R) Core(TM) i5 M 540
Operating System (OS)	Ubuntu Linux 12.04 32 bit
Ethernet	100/1000 mbit/s
Memory	4 GB DDR3

Asus eeePC X101CH	
Processor	Intel(R) Atom(TM) N2600
OS	Ubuntu Linux 12.04 32 bit
Ethernet	100 mbit/s
Memory	1 GB DDR3

Table 5.2: Splitting of the running software packages during the experiment

Dell Latitude E4310	**Asus eeePC X101CH**
openNI Kinect	ROS master
cob-people-detection	Turtlebot Software
Person Detection Software	Navigation Software
Human Interface	Search Software
Speech Recognition	Robot Control
Text-to-Speech	

'move_base'-node on the Asus notebook crashes. Synchronizing the clock times with the same timeserver before every start is strongly recommended.

5.1.3 Reasons for this Distribution

The Asus laptop supports 100 mbit/s network whereas the topic used as information source by the cob-people-detection software streams 50 mb/s [1]. Moreover is the processor of the Asus laptop not fast enough to calculate pointclouds from the depthimage with a

[1] Own measurement using the command "rostopic bw '/camera/depth_registered/points"' executed at the Dell laptop with the camera connected to the same laptop

frequency higher than 2-3 Hz [2]. Therefore the only useful setup is to connect the Kinect camera to the same laptop running the cob-people-detection software.

5.2 Results of the Experiment

The scenario without a present person has been tested 5 times and includes all exploration points shown in 5.1.1. The other scenarios just include the point before and after the person as the functionality of the whole system has already been tested in the first scenario. In the column 'Detected Obstacles' it is shown how many obstacles have been detected. The obstacles which were rated high enough to be considered as a person are shown in column 'Confirmed Obstacles'.

5.2.1 No Person Scenario

Table 5.3: Results of the scenario without a person

Nr.	Detected Obstacles	Confirmed Obstacles	Detected Faces	Confirmed Faces	Execution Time (s)	Panorama Waiting Time (s)	Scenario fulfilled
1	0	0	0	0	660	-	yes
2	10	3	0	0	1200	-	yes
3	10	2	0	0	1260	-	yes
4	12	4	0	0	1748	752	no[1]
5	14	3	0	0	1179	0	yes

[1]: One exploration point has not been reached.

5.2.2 Chair Scenario

To fulfill this scenario

1. the person has to be detected and
2. the answers have to be recognized.

This is shown in the column 'Scenario fulfilled'. The search has been aborted after the robot passed the person as there was not any chance for a success left.

[2] Own measurement using the command "rostopic hz '/camera/depth_registered/points"' executed at the Dell laptop with the camera connected to the Asus

Table 5.4: Results of the chair scenario

Nr.	Detected Obstacles	Confirmed Obstacles	Detected Faces	Confirmed Faces	Execution Time (s)	Panorama Waiting Time (s)	Scenario fulfilled
1	2	1	0	0	233	0	yes[1]/no
2	1	0	1	1	301	0	yes/no
3	2	1	1	1	290	0	yes/no
4	1	0	0	0	99	0	no[2]
5	0	0	0	0	86	0	no[2]

[1]: The legs have been recognized as an additional obstacle.

[2]: The face of the person was not completely in the field of view of the camera.

5.2.3 Lying Scenario

Table 5.5: Results of the lying scenario

Nr.	Detected Obstacles	Confirmed Obstacles	Detected Faces	Confirmed Faces	Execution Time (s)	Panorama Waiting Time (s)	Scenario fulfilled
1	5	2	0	0	407	0	yes
2	8	3	0	0	480	0	yes
3	10	3	0	0	294	0	no
4	5	2	0	0	245	0	no
5	7	0	0	0	120	0	no

The last exploration point has not been reached in any run of this scenario as the lying person blocked the whole way and was not recognized by the Kinect camera. Both detections were not based on incoming data from the laserscanner but from the obstacle detection based on the bump sensors at the front of the robot.

5.2.4 Wall Scenario

To fulfill this scenario

1. the person has to be detected and

2. the answers have to be recognized.

This is shown in the column 'Scenario fulfilled'.

Table 5.6: Results of the wall scenario

Nr.	Detected Obstacles	Confirmed Obstacles	Detected Faces	Confirmed Faces	Execution Time (s)	Panorama Waiting Time (s)	Scenario fulfilled
1	1	1	0	0	419	0	yes/no
2	2	1	0	0	170	0	yes/yes
3	2	1	0	0	374	0	yes/yes
4	1	1	0	0	137	0	yes/yes
5	2	1	0	0	207	0	yes/yes

5.3 Main Robot Control

The robot controll software offers a goal management and a capability management. It can accept goals from external sources like the database and forward them for the execution respecting the priorities of the goals. Futhermore modules can connect to this software and register tasks.

The external task executing server in a module can loose the received goals after a few seconds. This is still a task for further investigation, but can be bypassed by sending frequently feedback to the main task server.

5.4 Database Binding

The database binding is an easy to use interface which is able to build up a connection and interact with the database. The database connection is very stable and never dropped. The automatic reconnect of the robot controller works, but all interacting software is not yet able to stash queries until the connection is reestablished.

The integration of a database connection is abstracted to a high level and just needs some lines of code as it is shown in listing A.5.1.

5.5 Person Detection

The software to detect a person can perceive a person based on face identification and the appearance as an additional obstacle compared to the static map.

5.5.1 Obstacle Approach

This method causes a lot of false detections. The amount can be seen in the results of the experiments. The false detections are mainly

- replaced items or
- static elements like walls.

Static elements are recognized due to odometry differences and false localization.

5.5.2 Face Recognition

The face recognition software has been evaluated in [BZF+13]. The maximum distance is higher than shown in [TMV13] but not enough to recognize a standing person. Furthermore tilted heads are not detected.
The implemented tracking method is not able to track fast moving persons. But as they usually either

- move outside the field of view of the camera or
- move outside the maximum distance for a recognition

this setup is sufficient for the assignment.

5.6 Module for Search Coordination

The coordination of the search is a module of the robot controller and can manage the execution of the search using a state machine.
The implementation fulfills the requirements. But as this module does not do path planning by using the priority and the position of goals, the robot often follows a not optimal path.

5.7 Human Interface

The human interface offers basic functionalities for an interaction with a person. The text-to-speech always worked and spoken sentences are not recognized as an possible answer. Using the equiped dictionary file of the 'robocub' competition the speech recognition does often not detect the answer 'yes'. It often assigns 'get' or 'this' instead. Furthermore even small noises are recognized as spoken words.

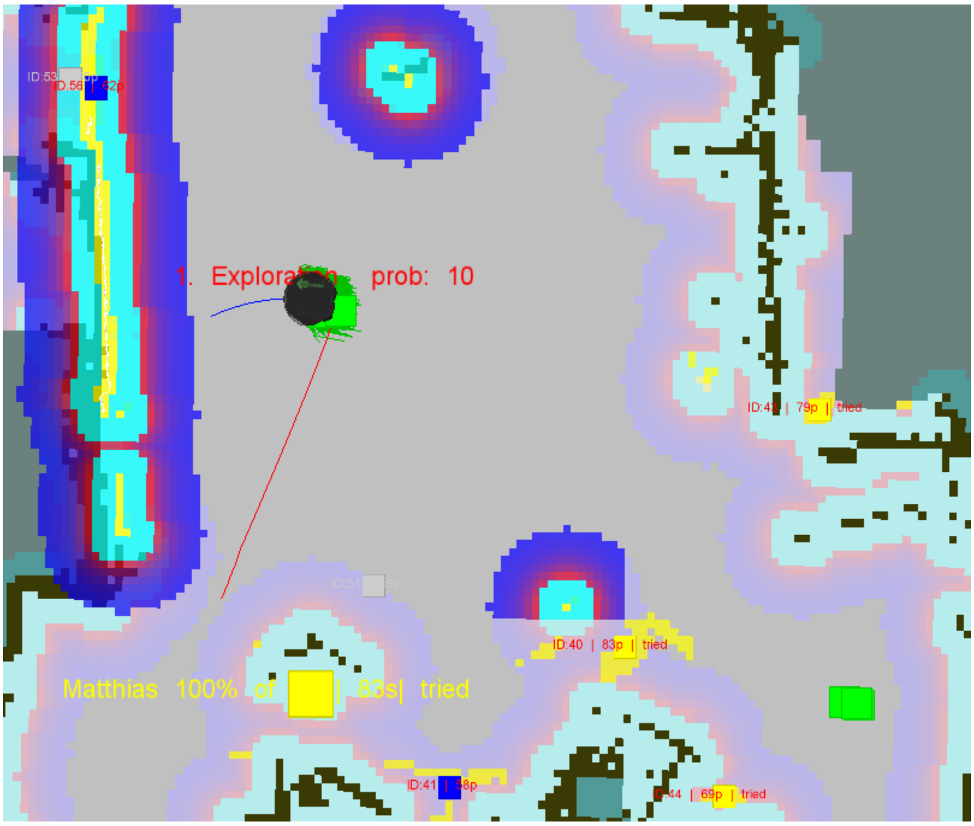

Figure 5.4: A running exploration task. The blue and red line are the planned navigation trajectories. In the bottom left the representation of a recognized face can be seen. In the bottom middle a detected but low rated obstacle can be seen.

5.8 Hardware

5.8.1 Turtlebot

Eventhough the Turtlebot offers a payload of 5 kg [Rob13] the navigation capabilities get worse if an additional laptop is attached to the robot. Rotating often fails, because the wheels are slipping. If the robot is stuck too long, the navigation software will abort the navigation goal. This limitation often leads to a not fullfilled scenario.

5.8.2 Laptops

The time on both laptops must accord exactly. Even slight time differences worsen the navigation capabilities a lot and lead to false detection of obstacles.

To be able to distribute tasks between several machines every computer should have a 1000 mbit/s ethernet connection. A slower connection or wireless network is not sufficient to transfer the required data for this setup.

5.9 Overall Result

The approach to connect a mobile robot to a database-centric AmI works. The shown setup allows to exchange information between both entities and fulfill goals. Based on this interaction the robot can search for a person in a known environment.
The system on the robot is able to accept several goals from the AAL-system as well as from robot internal clients and manage the execution. Furthermore it is possible to register new modules on runtime and therefore offer a flexible management system.
But whereas the design of the system works practical problems during the execution still accure. These problems are mainly focused on the hardware and the used existing software.

Chapter 6

Discussion and Conclusion

This chapter evaluates and interprets the results of the previous chapter. It summerizes the contributions and gives an outlook to further development.

6.1 Discussion

In this section the contributions are shown and the results are discussed. Furthermore the limitation of the approaches and the implementation are examined.

6.1.1 Main Robot Control

The main robot control is able to do

- a goal management
- a capability management and
- a resource management.

Furthermore it implements a resource adapter for the active database.
At the moment the registration of external task server is automatically canceled if the connection dropped. In the future, modules that do not respond anymore could be automatically restarted. Furthermore the code will be divided into a Turtlebot specific and a general part in order to allow reusage in other projects.

6.1.2 Database Connection

The database binding fulfills the requirements and does not need any improvements. It does not implement to resend a call if it did not succeed due to a dropped connection.

But it is better to react to a failed call in the code sending the call as the reaction can strongly differ.
During the work the existing package has been enhanced by the missing capabilities and is now an easy to use package to

- build up a connection and reconnect it if dropped,
- call function in the database,
- listen and unlisten to channels,
- receive notifications,
- insert into tables and
- read from tables.

6.1.3 Finding a Person

The mounting height of the Kinect camera is a problem in both approaches to detect a person. It is too low to detect a face of a standing person and can be too high to detect a lying person.
To get better and more reliable results for navigation, obstacle detection and face recognition it is strongly recommended either

- use two cameras
  - a low mounted for navigation and obstacle detection and
  - a camera on the top of the robot to do face or torso detection
- or to use a tilt module to switch between a navigation and a detection position.

6.1.3.1 False Detection Rate on Obstacles

In table 5.3 it can be seen that the false detection rate of obstacles is high. False detections have mainly two reasons:

- False localization
- Map inaccurancy

The false detection in a search could be significantly lowered by the usage of an infrared sensor as it is shown in [SSC+13].

False Localization The obstacle detection algorithm is not aware of localization inaccuracy. The position of the robot is always treated as the ground truth and data is added even if the localization algorithm replaces the robot. This problem could be remedled by an more advanced approach for the storage of seen obstacles; for example by using the variance of the robot position.

Map in inaccurancy The used map should be precise in order to avoid additional false detections. It is difficult to achieve a better map quality, because a copy of a floor plan does not include furniture and a more precise map based on SLAM is not easy to make.

6.1.3.2 Missing Detection of a Lying Person

The approach to find a person based on the navigation input is based on the appearance of that person in the sensor data. In section 4.2.2.1 it is described that the detected obstacles are limited by the mounting height of the Kinect camera. If a person is thin and lying totally flat, the navigation software will not always be able to perceive the person. This results in two things:

- The robot will hit the person
- The robot will is rarely detect the obstacle and will not try to interact with the person or take a photo

To overcome this, the Kinect camera could be mounted lower.

6.1.3.3 Face Detection

A detection of faces of persons far away as well as of persons with averting heads is not yet possible. Evenmore the detection of 'moving' faces for example while the robot is slowly rotating at an exploration point is unreliable. While the distance is a limitation of the camera averted heads could be detected by an additional set of detectors this would cause more computation.
The cob-people-detection package also offers head detections as an output stream. With this information the robot could go to the interesting place and ask the possible person to turn the head in order to be able to recognize the face. Furthermore that information could be used to to stop rotating in order to be able to recognize a face.

6.1.4 Module for the Search Coordination

The search software offers a working state machine to coordinate a search. Eventhough the design it does not yet do advanced path planning the chosen approach works and shows

good results. A drawback is, that the probability or priority of a goal is not considered yet. If the goals are not ordered by the database or the search is started with a lot of known detections the search will take long. A path planning could improve that.
Furthermore this software is not yet able to handle a cancelation of the navigation to a goal. In the current implementation this goal will be marked as 'done' although it has never been reached.

6.1.5 Human Interface

The implemented software offers basic functionalities for an interaction with a person. In the results it is shown that even the simple answer 'yes' is often not detected. Therefore it is recommended to use another speech recognition engine or improve the speech model and the dictionary.
To be able to distinguish speaking persons the speech localization software 'Hark' [1] could be used. Furthermore it could be asked if the recognized answer is correct and an avatar could displayed on the laptop screen.
Although the capabilities are limited this component can be a valuable utilitiy for other programmers.

6.2 Conclusion

In this thesis the approach of the extension of AmI with a mobile robot is examined. Furthermore the task to search for a person should be implemented to show that the combination can benefit from each others strenghts.
The robot hardware is a Turtlebot 2, the AAL system is implemented in a PostgreSQL database and the setup is based on the Robot Operating System (ROS).
To enable the robot to execute task the robot must be connected to the database, be able to accept and handle goals, detect a human, interact with a person and execute a search. The developed setup is able to execute a goal for the AAL systems and to report the result. The task to find a person can search in an apartment and can deliver the position of a found person or a set of pictures of possible spots. The implemented human robot interaction offers basic functionalities and the extendable robot control system is able to register new kind of tasks at runtime.
The detection of a lying person with this hardware setup is limited to the height of the Kinect camera. Furthermore the obstacle detection can have false detections if items are replaced or the localization of the robot is wrong. The perception of faces is limited to the opening angle of the camera.

[1] Website of the Hark project: `http://winnie.kuis.kyoto-u.ac.jp/HARK/`

These problems can be solved by attaching additional cameras and a more advanced obstacle detection based on the covariance of the robot position.
It is shown that a robot can be a useful and effective extension of an AAL system and can help to find a person. Both entities can benefit from the interaction. Whereas the mobile robot can provide information about the current situation the AmI is able to store long time information and to pass them to the robot if it is needed for the execution of a task. The proposed robot controller and its extensibility with modules that provide different capabilities has been shown to be well adapted to the concept of a database-centric AmI system.
The proposed system allows the further implementation of more types of tasks and can be used either as a personal robot or in a fleet of robots in a care facility. The implementation of new modules is easy as a parent class is provided for that purpose and the integration of a database connection is abstracted to a high level.

6.3 Outlook

The shown system of an expandable robot controller, modules that implement tasks and the connection to an active database can be the basis of a large quantity of scenarios and applications of mobile robots extending AmI.
If a personal robot becomes capable of performing more tasks and being realiable it can become an important helper for care facilities or at home. At home it could extend an AmI system by executing specific tasks. Furthermore these tasks can support AAL like the implemented task to search for a person. In care facilities a fleet of robots could be used to provide help for the care takers. An intelligent management system can distribute goals in the whole robot fleet. The possibilty to run a different set of modules on every robot can provide a new dimension of task assignment for those systems. Regarding the hardware and software capabilites, every robot can have an own skill set. For example all mobile robots are able to fulfill a task like 'goto' but just robots with manipulators can do a task like 'tidy_up'.
Moreover multiple robots could be used to execute the same task and exchange information about explored places using the AmI system.

Acronyms

AAL Ambient Assisted Living

AmI Ambient Intelligence

API Application Programming Interface

BSD Berkley Software Distribution (a software distribution licence)

IMU Intertial Measurement Unit

IPC Inter Process Communication

MTS Main Task Server (a part of the goal management in the robot controller)

openCV Open Source Computer Vision Library

OS Operating System

PID Process Identification Number (a unique number to access a process)

REP ROS Enhancement Proposals (ideas to enhance and guidelines to implement ROS[2])

RGB Red Green Blue (a color space)

ROS Robot Operating System

RPC Remote Procedure Call

RViz ROS Visualization (3D visualization tool for ROS[3])

SLAM Simultanious Location and Mapping (a robot based mapping approach)

SQL Structured Query Language (a programming language for relational database management systems)

---

[2] `http://www.ros.org/reps/rep-0000.html`

[3] `http://wiki.ros.org/rviz/`

UDF User Defined Function (a function in a database)

Bibliography

[ARA12] M.R. Alam, M. B I Reaz, and M. A M Ali. A Review of Smart Homes;Past, Present, and Future. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1190–1203, Nov 2012.

[asu14] Technical Data of the Asus X101 CH. `http://www.asus.com/Notebooks_Ultrabooks/Eee_PC_X101CH/#specifications`, 03 2014.

[BWB+] Wassilios Bentas, Marc Wolfram, Ronald Bräutigam, Michael Probst, Wolf-Dietrich Beecken, Dietger Jonas, and Jochen Binder. Da Vinci robot assisted Anderson-Hynes dismembered pyeloplasty: technique and 1 year follow-up. *World Journal of Urology*, 21(3):133–138.

[BZF+13] Richard Bormann, Thomas Zwölfer, Jan Fischer, Joshua Hampp, and Martin Hägele. Person Recognition for Service Robotics Applications . volume 13th. IEEE, 2013.

[CMBV13] B. Choi, C. Mericli, J. Biswas, and M. Veloso. Fast human detection for indoor mobile robots using depth images. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1108–1113, May 2013.

[CVBK11] Anand Chandrasekhar, Vineeth.P.Kaimal, Chhayadevi Bhamare, and Miss Sunanda Khosla. Ambient Intelligence: The Next Generation Technology. *International Journal on Computer Science and Engineering (IJCSE)*, 03:2491–2497, 06 2011.

[DB13] Torbjørn S Dahl and Maged N Kamel Boulos. Robots in Health and Social Care: A Complementary Technology to Home Care and Telehealthcare? *Robotics*, 3(1):1–21, 2013.

[del14] Technical Data of the Dell Latitude E4310. `http://www.notebookcheck.net/Review-Dell-Latitude-E4310-Subnotebook.45771.0.html`, 03 2014.

[det10] *Face Detection using 3-D Time-of-Flight and Colour Cameras*. VDE-Verlag, 2010.

[dMW13] W.O. de Morais and N. Wickstrom. A 'Smart Bedroom' as an Active Database System. In *Intelligent Environments (IE), 2013 9th International Conference on*, pages 250–253, July 2013.

[git14] Detection.msg of the cob-people-detection on Github. 03 2014.

[Goe] Patrick Goebel. Pi Face Tracker Package in the ROS Wiki. `http://wiki.ros.org/pi_face_tracker`.

[GRH+09] B. Graf, U. Reiser, M. Hägele, K. Mauz, and P. Klein. Robotic home assistant Care-O-bot - product vision and innovation platform. In *Advanced Robotics and its Social Impacts (ARSO), 2009 IEEE Workshop on*, pages 139–144, Nov 2009.

[har14] Hark in the ROS Wiki. `http://wiki.ros.org/hark`, 03 2014.

[HTK+05] Panu Harmo, Tapio Taipalus, Jere Knuuttila, José Vallet, and Aarne Halme. Needs and solutions-home automation and service robots for the elderly and disabled. In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 3201–3206. IEEE, 2005.

[HWB+13] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees . *Autonomous Robots*, pages 1–16, 2013.

[ki-14] Kinect Details on a Microsoft Homepage. `http://msdn.microsoft.com/en-us/library/hh855368`, 03 2014.

[KVSD06] S. Knoop, S. Vacek, K. Steinbach, and R. Dillmann. Sensor fusion for model based 3D tracking. In *Multisensor Fusion and Integration for Intelligent Systems, 2006 IEEE International Conference on*, pages 524–529, Sept 2006.

[lib14] libpq in the PostgreSQL Documentation. `http://www.postgresql.org/docs/9.1/static/libpq.html`, 03 2014.

[LPP+11] Ingo Lütkebohle, Roland Philippsen, Vijay Pradeep, Eitan Marder-Eppstein, and Sven Wachsmuth. Generic middleware support for coordinating robot software components: The Task-State-Pattern . *Journal of Software Engineering for Robotics*, 2011.

[mai] Website of the MAID project. `http://rob.ipr.kit.edu/projekte_1388.php`.

[MHN+10] T. Mukai, S. Hirano, H. Nakashima, Y. Kato, Y. Sakaida, S. Guo, and S. Hosoe. Development of a nursing-care assistant robot RIBA that can lift a human in its arms. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 5996–6001, Oct 2010.

[Nat01] United Nations. World Population Ageing: 1950-2050. Web, 2001.

[OFD+09] A.G. Ozkil, Zhun Fan, S. Dawids, H. Aanes, J.K. Kristensen, and K.H. Christensen. Service robots for hospitals: A case study of transportation tasks in a hospital. In *Automation and Logistics, 2009. ICAL '09. IEEE International Conference on*, pages 289–294, Aug 2009.

[QBN07] Morgan Quigley, Eric Berger, and Andrew Y. Ng. STAIR: Hardware and Software Architecture . Technical report, Association for the Advancement of Artificial Intelligence (www.aaai.org), 2007.

[RM13] P. Rashidi and A. Mihailidis. A Survey on Ambient-Assisted Living Tools for Older Adults. *Biomedical and Health Informatics, IEEE Journal of*, 17(3):579–590, May 2013.

[Rob13] Clearpath Robotics. *Turtlebot 2 Manual*, 0.5 edition, 2013.

[ros14a] actionlib in the ROS Wiki. 03 2014.

[ros14b] AMCL in the ROS Wiki. `http://wiki.ros.org/amcl`, 03 2014.

[ros14c] Costmap in the ROS Wiki. `http://wiki.ros.org/costmap_2d?distro=hydro#Inflation`, 03 2014.

[ros14d] Move Base in the ROS Wiki. `http://wiki.ros.org/move_base?distro=hydro`, 03 2014.

[ros14e] OctoMap Representations in the ROS Wiki. `http://wiki.ros.org/ccny_rgbd/keyframe_mapper`, 03 2014.

[ros14f] Pocketsphinx in the ROS Wiki. `http://wiki.ros.org/pocketsphinx`, 03 2014.

[ros14g] SQL-Database Package in the ROS-Wiki. `http://wiki.ros.org/sql_database/`, 03 2014.

[ros14h] Topics in the ROS Wiki. `http://wiki.ros.org/Topics`, 03 2014.

[SD03] Peter Steinhaus and Rüdiger Dillmann. Aufbau und Modellierung des RoSi Scanners zur 3D-Tiefenbildakquisition . Technical report, 2003.

[SFR11] Corry-Ann Smarr, Cara Bailey Fausset, and Wendy A. Rogers. Understanding the Potential for Robot Assistance for Older Adults in the Home Environment . Technical report, Human Factors & Aging Laboratory, 2011.

[spe14] List of Speech Recognition Software in the Wikipedia. `https://en.wikipedia.org/wiki/List_of_speech_recognition_software`, 03 2014.

[SSC+13] Loreto Susperregi, Basilio Sierra, Modesto Castrillón, Javier Lorenzo, Jose María Martínez-Otzeta, and Elena Lazkano. On the Use of a Low-Cost Thermal Sensor to Improve Kinect People Detection in a Mobile Robot. *Sensors*, 13(11):14687–14713, 2013.

[TMV13] C. Tonelo, A.P. Moreira, and G. Veiga. Evaluation of sensors and algorithms for person detection for personal robots. In *e-Health Networking, Applications Services (Healthcom), 2013 IEEE 15th International Conference on*, pages 60–65, Oct 2013.

Appendix A

Source Code

The source code can be found on Github and is licensed und GPL v2 as well as the BSD-licence.
Link to the Github repository: `http://www.github.com/matthiashh`

A.1 Robot Control Node

The robot control node is a completely self implemented component. The listing below shows the header file of the robot controller.

```
#ifndef ROBOT_CONTROL_H
#define ROBOT_CONTROL_H
#include <ros/ros.h>                                    // general ROS-header
#include <kobuki_msgs/ButtonEvent.h>                    // for kobuki button eventhandling
#include <kobuki_msgs/Led.h>                            // for kobuki led handling
#include <move_base_msgs/MoveBaseAction.h>              // for navigation_goals
#include <robot_control/RobotTaskAction.h>              // the generic task action
#include <actionlib/client/simple_action_client.h>      // for a move_base client
#include <actionlib/client/action_client.h>             // to use an actionclient
#include <actionlib/server/action_server.h>             // to use an actionserver
#include <std_msgs/Time.h>
#include <database_interface/postgresql_database.h>     // to connect to a postgresql database
#include <robot_control/RegisterTaskServer.h>           // the registration message

//needed for actionserver
typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;
typedef actionlib::ActionServer<robot_control::RobotTaskAction> TaskServer;


//general connection enom, used for database_connection
namespace robot_control
{
  //! Enum to support multiple states of a service robot
  /*! It is intended to allow users to shut off database support */
```

```
  enum robotMode
  {
    database,              //!< Accept tasks from the database
    speechControlled,      //!< Allows the user to control the robot by speech
    manual                 //!< For example for debugging or developing
  };

  //! Struct to store the connection and task executing capabilities
  /*! Every connection to an external task executing server is stored in element of this type  */
  struct externalServer
  {
    std::string task_name;                                          //!< Task identifying name
    std::string task_server_name;                                   //!< Topicname of the task server
    actionlib::ActionClient<robot_control::RobotTaskAction>* as;    //!< Access to the connection
    ros::Time last_contact;                                         //!< Frequenctly updated on every connection check
  };

  //! A struct for extern running tasks
  struct srvClGoalPair
  {
    TaskServer::GoalHandle* srv;                                          //!< Reference to the refering goal handle on the main_task_server
    actionlib::ClientGoalHandle<robot_control::RobotTaskAction> cl;    //!< Goalhandle of the extern running task
  };

  //! Struct for the database client
  struct GoalClientPair
  {
    RobotTaskGoal goal;                                                          //!< Goal of the task, because it can't be accessed through the goalhandle
    actionlib::ClientGoalHandle<robot_control::RobotTaskAction> client_gh;   //!< Corresponding goalhandle
  };
}


//! The main task running the whole robot controlling software

class RobotControl
{
private:
    //ros
    ros::NodeHandle n_;                //!< Mandatory nodehandle
    // Main ActionServer
    //! The main task server
    /*! The main task server is the central task server. Every task has to be sent to that server in order to be executed.*/
    actionlib::ActionServer<robot_control::RobotTaskAction> main_task_server_;
    actionlib::ActionClient<robot_control::RobotTaskAction> db_task_client_;             //!< The database client of the main task server

```

```
    std::vector<TaskServer::GoalHandle> all_goals_server_;
        //!< All goals of the main task server are stored here
    bool goal_active_;
        //!< True if an external goal is running

    // External Actionserver
    std::vector<robot_control::externalServer> all_external_as_;
        //!< All connected external taskserver
    std::vector<robot_control::srvClGoalPair> all_external_goals_;
        //!< All external running goals
    ros::Subscriber sub_reg_taskserver;
        //!< Subscriber for the registration topic

    // Database Actionclient

    std::vector<robot_control::GoalClientPair> all_db_goals_;
        //!< All database goals and goalhandles

    // Kobuki base
    ros::Publisher motors_;
        //!< Publisher to turn the motors on and off
    ros::Subscriber button_;
        //!< Subscriber to button input
    ros::Publisher led1_pub;
        //!< Publisher for the first LED
    kobuki_msgs::Led led1_;
        //!< Msg to see the last state
    ros::Publisher led2_pub_;
        //!< Publisher for the second LED
    kobuki_msgs::Led led2_;
        //!< Msg to see the last state
    ros::Publisher kob_sound_;
        //!< Can make sounds on the turtlebot
    bool button0_;
        //!< Latest state of button 0
    bool button1_;
        //!< Latest state of button 1
    bool button2_;
        //!< Latest state of button 2

    // Robot Control
    robot_control::robotMode robot_mode;
        //!< Storing the state of the robot (not used yet)
    bool running;
        //!< To see if the database told the robot to stop
    // Database

    database_interface::PostgresqlDatabase* database_;
        //!< The connection to the database
    // FUNCTIONS
    // Callbacks

    //! Callback for turtlebot buttons
    /*! \param button the received button message */
    void buttonCallback(const kobuki_msgs::ButtonEvent button);

    //! Callback for the registration process
```

```
    /*! \param reg Received registration message */
    void registerCallback(const robot_control::RegisterTaskServer reg);

    //! Callback if an external goals state is changed
    /*! \param cgh Received goalhandle of the goal running externally */
    void transitionCallbackExternalGoals(actionlib::ClientGoalHandle<robot_control::
        RobotTaskAction> cgh);

    //! Callback if an database goals state is changed
    /*! \param cgh Received goalhandle of the goal running on the main task server */
    void transitionCallbackDatabaseGoals(actionlib::ClientGoalHandle<robot_control::
        RobotTaskAction> cgh);

    //! Callback for feedback of external goals
    /*! \param cgh Received goalhandle delivering the information */
    void feedbackCallbackExternalGoals(actionlib::ClientGoalHandle<robot_control::
        RobotTaskAction> cgh);

    //! Callback for feedback of database goals
    /*! \param cgh Goalhandle carying the feedback */
    void feedbackCallbackDatabaseGoals(actionlib::ClientGoalHandle<robot_control::
        RobotTaskAction> cgh);

    // Database
    //! Reacts based on information in the notification
    /*! \param no Notication delivering the information
        \return False if the notfication doesn't fit to the known ones */
    bool processNotification(database_interface::Notification no);

    //! Get new tasks if the database triggers
    /*! \return fails if the call failed */
    bool getTasks();

    //! Checks the database connection, sets the LED and checks for notifications
    /*! \return false if there's no connection */
    bool checkDatabaseConn();
    // Main Actionserver

    //! Called if a new goal is receieved
    /*! \param gh The new goal */
    void TaskServerGoalCallback(TaskServer::GoalHandle gh);

    //! Called if a goal is cancelled
    /*! \param gh The cancelled goal */
    void TaskServerCancelCallback(TaskServer::GoalHandle gh);

    //! Finds the goal with the highest priority and sets it
    bool setNewMainGoal();

    // External Actionserver
    //! Checks contact to all external actionserver
    /*! \param max_time Maximum time a connection can stay dropped until the external
        server is deleted and the corresponding goals are aborted */
    void checkContact(ros::Duration max_time);

    // Robot Control
    //! Starts robot services if the database asks devices to run
    void dbStart();
```

```

    //! Stop robot services if the database asks for a stop
    void dbStop();

    //! Deprecated function to do a full turn
    int fullTurn();

public:
    //! Constructor of the main class
    /*! \param name Topic name of the actionserver */
    RobotControl(std::string name);
    // Database

    //! Connect to the database
    /*! \return true if it worked */
    bool connectDb();

    //! Gets the config from the database
    /*! \return  */
    bool getConfig();

    //! Runs the controller and just returns on critical errors
    int run();
};

#endif // ROBOT_CONTROL_H
```

Listing A.1: The header file of the robot controller

A.1.1 Robot Control Simple Client

The robot control simple client is a utility class to allow an easy implementation of modules.

```
 #include <ros/ros.h>                              // mandatory ros header
#include <string>                                 // to process strings
#include <robot_control/RegisterTaskServer.h>     // to register an task_server
#include <robot_control/RobotTaskAction.h>        // the action for the goals
#include <actionlib/server/action_server.h>       // the header for the server

typedef actionlib::ActionServer<robot_control::RobotTaskAction> _task_server;

//! A utility class to allow an easy implementation of modules

class RobotControlSimpleClient
```

```
{
private:
  ros::Publisher pub_registration_;          //!< Publisher to register at robot_control
  std::string task_name_;          //!< The name of the offered task
  std::string task_server_name_;          //!< The address/topic name of the task server
  void TaskServerGoalCallback(_task_server::GoalHandle gh);          //!< The callback for incoming goals
  void TaskServerCancelCallback(_task_server::GoalHandle gh);          //!< The callback for canceled goals
protected:
  ros::NodeHandle n_;          //!< Mandatory ROS nodehandle
  _task_server::GoalHandle task_goal_;          //!< The current goal
  robot_control::RobotTaskResult result_;          //!< A template result for easier usage
  robot_control::RobotTaskFeedback feedback_;          //!< A template feedback for easier usage

  //! Function to register at robot_control
  /*! \return true if success*/
  bool registerServer();

  //! Function to deregister at robot_control
  /*! \return true if success */
  bool deregisterServer();

  //! A possibility to check the incoming goal before it gets accepted
  /*! \param goal The new goal
      \param res  The returned result if the goal is rejected
      \return true if accepted; false if rejected */
  virtual bool checkIncomingGoal(robot_control::RobotTaskGoalConstPtr goal,
      robot_control::RobotTaskResult &res);

  //! Draft: A possibility to prepare before a new goal is running
  /*! \param goal The new goal
      \param res  The result for the old goal */
```

```
  virtual void prepareForNewGoal(robot_control::RobotTaskGoalConstPtr goal,
      robot_control::RobotTaskResult &res);

  //! A possibility to clean up the canceled old goal
  /*! \param res Result of the old goal */
  virtual bool cleanupCancelledGoal(robot_control::RobotTaskResult &res);

  //! Executes a ros::spinOnce
  void spinServer();

  //! Easy identifier if a goal is currently active
  bool goal_active_;
public :
  //! The task server object
  /*! Has to be public as this is the only way to get it work.*/
  actionlib::ActionServer<robot_control::RobotTaskAction> task_server_;

  //! The constructor
  /*! \param task_server_name The address/topic name of the task server
   *  \param task_name        The name of the implemented task */
  RobotControlSimpleClient(std::string task_server_name, std::string
     task_name);
  ~RobotControlSimpleClient();
};
```

Listing A.2: The head file of the 'RobotControlSimpleClient' utility class

A.1.1.1 Example Code for a Simple Module

This module implements the task 'say'. This task does not have a database connection. The additional code to build up a database connection can be found in section A.5.1.

```
#include <ros/ros.h>                               // mandatory ROS_header
#include <human_interface/SpeechRequest.h>         // to perform the
    speech request
#include <robot_control/RobotControlSimpleClient.h> // to connect it to
    robot controll
#include <string>

class say : public RobotControlSimpleClient
{
private:
  // publisher for text-to-speech
  ros::Publisher pub_speech_;
```

```
  // overloading function to check a goal before it is accepted
  bool checkIncomingGoal(robot_control::RobotTaskGoalConstPtr goal, robot_control::RobotTaskResult &res);
public:
  // constructor; 'task_server_name' specifies where to reach the server; 'task_name' specifies the identifier/name of the task
  say(std::string task_server_name, std::string task_name);
  void run();
};

bool say::checkIncomingGoal(robot_control::RobotTaskGoalConstPtr goal, robot_control::RobotTaskResult &res)
{
  // we could check the database here
  return true;
}

say::say(std::string task_server_name, std::string task_name) : RobotControlSimpleClient(task_server_name,task_name)
{
  // initialize the publisher for text-to-speech
  pub_speech_ = n_.advertise<human_interface::SpeechRequest>("/human_interface/speech_request",10);
}

void say::run()
{
  ros::Rate r(3);
  while (ros::ok())
  {
    // 'goal_active_' is switched to true if a new goal arrives
    if (goal_active_)
    {
      // send text to the human_interface
      human_interface::SpeechRequest req;
      req.text_to_say = "Yeah - it worked. This is goal ";
      int id = (*task_goal_.getGoal()).task_id;
      req.text_to_say += boost::lexical_cast<std::string>(id);
      pub_speech_.publish(req);
      // report success to the robot controller
      result_.success = true;
      result_.end_result = "Everything worked";
      ros::Duration(5).sleep();
      task_goal_.setSucceeded(result_,"Everything worked");
      goal_active_ = false;
    }
```

```
    r.sleep();
    spinServer();
    ros::spinOnce();
  }
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "say_sentence");
  say say_object("/robot_control_basics/say_task_server","say");

  ROS_INFO("Finished initialization, now running in the loop");
  //This loop is supposed to run endless
  say_object.run();
  return 0;
}
```

Listing A.3: Example code for a module implementing a simple task

A.2 Person Detector Node

The header file of the person detection implementing a handling of face recognition and an approach to find additional obstacles in the environment.

```
 #ifndef PERSON_DETECTOR_H
#define PERSON_DETECTOR_H
#include <ros/ros.h>                                   // general ROS-functionalities
#include <cob_people_detection_msgs/DetectionArray.h>  // message type for the cob_people_detection_topic
#include <queue>                                       // used to store the detections
#include <vector>                                      // used to store amcl-poses
#include <tf/transform_listener.h>                     // currently unused
#include <tf/transform_broadcaster.h>                  // used to broadcast detections
#include <visualization_msgs/Marker.h>                 // display markers on rviz
#include <visualization_msgs/MarkerArray.h>            // advanced display of marker on rviz
#include <person_detector/DetectionObjectArray.h>      // our detections
#include <person_detector/DetectionObject.h>           // used for a single detection
#include <person_detector/SpeechConfirmation.h>        // for speech confirmations we receive
#include <person_detector/ObstacleArray.h>             // to store found obstacles
#include <nav_msgs/OccupancyGrid.h>                    // the map format
#include <costmap_2d/layer.h>                          // to use a costmap
#include <costmap_2d/costmap_2d_ros.h>                 // to use a costmap
#include <sensor_msgs/Imu.h>                           // to get information about the rotation
#include <geometry_msgs/PoseWithCovarianceStamped.h>   // for amcl
#include <geometry_msgs/Pose.h>                        // to save the center of an obstacle
```

```
namespace person_detector {
  //! This struct is used to store the obstacle map points
  /*! The obstacle points in the Obstacle.msg are stored in metric map coordinates. This would require a lookup metric map coordinates to the corresponding fields in the costmap array each time the obstacle has to be found again. Therefore the arrays positions of all points of an obstacle are also stored in this struct.*/
  struct ObsMapPoints
  {
    unsigned int id;                                          //!< The unique ID of the obstacle
    std::vector<geometry_msgs::Point> points;                 //!< A vector points on the costmap array. They x and y values are pointing to a field on the costmap and aren't metric
  };
}

/*! The person_detector class manages the whole process of detections based on face recognitions and found obstacles */

class person_detector_class
{
private:
  //ros-stuff
  ros::NodeHandle n_;                                         //!< The mandatory ROS nodehandler
  ros::Subscriber sub_face_recognition_;                      //!< Subscriber to cob_people_detection face recognitions
  ros::Publisher pub_all_recognitions_;                       //!< Publisher of all face recognitions
  ros::Publisher pub_all_obstacles_;                          //!< Publisher of all obstacles
  ros::Subscriber sub_map_;                                   //!< Subscriber to the map topic
  ros::Subscriber sub_local_costmap_;                         //!< Subscriber to the local costmap of move_base
  ros::Subscriber sub_obstacles_;                             //!< Subscriber to the published obstacles by move_base
  ros::Subscriber sub_imu_;                                   //!< Subscriber to the robots gyro data
  ros::Subscriber sub_confirmations_;                         //!< Subscriber to the confirmations published by other nodes
  ros::Subscriber sub_amcl_;                                  //!< Subscriber to the robots positions published by amcl
  ros::Subscriber sub_reset_all_;                             //!< Subscriber to reset the obstacles and the detections
  ros::Subscriber sub_reset_obstacles_;                       //!< Subscriber to reset the detected obstacles
  ros::Subscriber sub_reset_detections_;                      //!< Subscriber to reset the face detections
  //transformations
  tf::TransformListener tf_listener_;                         //!< Transformation listener to get transformations between a face recognition and the map
  tf::StampedTransform transform_li_;                         //!< Resuable transformation object for the transformation listener
  tf::TransformBroadcaster tf_human_local_broadcaster_;       //!< Transformation broadcaster to announce transformations between the camera and a detected faces
  tf::Transform transform_br_;                                //!< Resuable
```

```
transformation object for the transformation broadcaster
tf::TransformBroadcaster tf_map_human_broadcaster_;          //!< Transformation broadcaster to announce transformations between the map and the detected faces
tf::Transform transform_br_map_;                             //!< Reusable transformation object for the transformation broadcaster
ros::Duration tf_cache_;                                     //!< Storing the information about the length of the tf_listener_ cache

//markers for rviz
ros::Publisher pub_human_marker_raw_;                        //!< Publisher of the raw received face detections as cubes in rviz. \sa showAllRecognitions
visualization_msgs::Marker heads_raw;                        //!< Reusable Marker for the pub_human_marker_raw_. Avoids long initialization. \sa showAllRecognitions
ros::Publisher pub_human_marker_raw_text_;                   //!< Publisher of the raw received face detections as text in rviz. \sa showAllRecognitions
visualization_msgs::Marker text_raw_;                        //!< Reusable Marker for the pub_human_marker_raw_text_. Avoids long initialization. \sa showAllRecognitions
ros::Publisher pub_human_marker_;                            //!< Publisher of all face recognitions stored in all_detections_ as cubes in rviz. \sa showAllRecognitions
visualization_msgs::Marker heads_;                           //!< Reusable Marker for the pub_human_marker_. Avoids long initialization. \sa showAllRecognitions
ros::Publisher pub_human_marker_text_;                       //!< Publisher of text to all face recognitions stored in all_detections_ in rviz. \sa showAllRecognitions
visualization_msgs::Marker heads_text_;                      //!< Reusable Marker for the pub_human_marker_text_. Avoids long initialization. \sa showAllRecognitions
ros::Publisher pub_obstacle_points_text_;                    //!< Publisher for information to every occupied point in the difference_map_ \sa showAllObstacles_ \sa difference_map_
visualization_msgs::Marker obstacle_points_text_;            //!< Reusable Marker for the pub_obstacle_points_text_. Avoids long intializiation. \sa showAllObstacles_ \sa pub_obstacle_points_text_
ros::Publisher pub_obstacle_borders_;                        //!< Publisher for a line connecting all points of an obstacle in rviz. \sa showAllObstacles_
visualization_msgs::Marker obstacle_boarder_marker_;         //!< Reusable Marker for the pub_obstacle_boarders_. Avoids long initialization. \sa showAllObstacles_ \sa pub_obstacle_boarders_
ros::Publisher pub_obstacle_cubes_;                          //!< Publisher for a cube representing an the middle of an obstacle in rviz. \sa showAllObstacles_
visualization_msgs::Marker obstacle_cubes_;                  //!< Reusbale Marker for the pub_obstacle_cubes_. Avoids long initialization. \sa showAllObstacles_ \sa pub_obstacle_cubes_
ros::Publisher pub_obstacle_info_text_;                      //!< Publisher for text information to each obstacle in rviz. \sa showAllObstacles_
visualization_msgs::Marker obstacle_info_text_;              //!< Reusable Marker for the pub_obstacle_info_text_. Avoids long intialization. \sa showAllObstacles_ \sa pub_obstacle_info_text_


//callbacks
//! Callback for face recognitions of the cob_peoble_detection.
/*! \param received_detections detection array from cob_peoble_detection*/
void faceRecognitionCallback_(const cob_people_detection_msgs::DetectionArray received_detections);

//! Callback for the occupancy map used by the robot.
/*! \param received_map The received map */
void mapCallback_(const nav_msgs::OccupancyGrid received_map);
```

```
//! Callback for the localCostmap provided by move_base
/*! \param received The received occupancy grid
    \sa updated_map_ \sa obstacleCallback_ \sa updated_dm_ \sa updated_counter_ */
void localCostmapCallback_ (const nav_msgs::OccupancyGrid received);

//! Callback for the occupied points provided by move_base
/*! \param pcl The received PointCloud
    \sa updated_map_ \sa updated_dm_ \sa updated_counter_  */
void obstaclesCallback_ (const sensor_msgs::PointCloud pcl);

//! Callback for the gyrometer output.
/*! \param imu The received imu data */
void imuCallback_(const sensor_msgs::Imu imu);

//! Callback for confirmations information about an obstacle
/*! \param conf The received confirmation information */
void confirmationCallback_(const person_detector::SpeechConfirmation conf);

//! Callback for the position of the robot provided by amcl
/*! \param pose The received pose with covariance */
void amclCallback_(const geometry_msgs::PoseWithCovarianceStamped pose);

//! The callback to reset all detections
/*! \param trig The trigger */
void resetAllCallback(const std_msgs::Empty trig);

//! The callback to reset obstacles
/*! \param trig The trigger */
void resetObtaclesCallback(const std_msgs::Empty trig);

//! The callback to reset all face detections
/*! \param trig The trigger */
void resetDetectionsCallback(const std_msgs::Empty trig);

//detections
//! Storage for detection arrays provided by cob_people_detection waiting to be
    processed
/*! The detections are just saved here until they are processed and matched with
    transformation information. A warn is sent out if this array becomes too big.
    \sa processDetections */
std::queue<cob_people_detection_msgs::DetectionArray> detection_temp_storage_;

//! The array of all current detections with all information.
/*! This array is frequently updated with new detections and new incoming data and sent
     out by the publisher.
    \sa processDetection \sa pub_all_recognitions_ */
person_detector::DetectionObjectArray all_detections_array_;

//! A counter for the unique IDs assigned to all recognitions.
/*! It has to incremented every time a new detected obstacle or a new detected face is
    is created. Obstacles and face recognitions share the same counter.*/
unsigned int recognition_id_;

//! Holds a copy of the static map received from the /map topic.
/*! If a new map is received, the map information is copied into this costmap. After
    that it will be inflated by 10cm.
    \sa inflateMap_ \sa mapCallback_ */
```

```
costmap_2d::Costmap2D static_map_;

//! This map is frequently updated with occupancy information.
/*! It stores the information about currently occupied and free points and receives
    information from the localCostmap of move_base and the published obstacles. It is
    used to calculate the difference map.
    \sa localCostmapCallback_ \sa obstaclesCallback_ \sa calcDifferenceMap \sa
        difference_map_ */
costmap_2d::Costmap2D updated_map_;

//! This map stores information about the distance from which the obstacle has been
    seen.
/*! The depthdata of the Kinect is very inaccurate on higher distances. The closest
    distance of a detection for each point is saved in this map. Please note, that it
    is stored in decimeter. A value of 10 results in 100cm closest distance. Obstacle
    seen by the localCostmap are always stored with 100cm distance. This map is used by
     rateObstacle.
    \sa localCostmapCallback \sa obstacleCallback \sa rateObstacle */
costmap_2d::Costmap2D updated_dm_;

//! This map stores information about the number of appearances of an obstacle.
/*! Sometimes obstacles are just seen a few times because it were walking people or
    wrong sensor information. This map counts the appearances of each occupied point.
    The maximum value is 255 according to the limit of unsigned char. If a occupied is
    marked as FREE_SPACE by the localCostmap, 10 points are substracted each time.
    \sa localCostmapCallback \sa obstaclesCallback_ \sa rateObstacle_ */
costmap_2d::Costmap2D updated_counter_;

//! This map is the difference between the updated map and the inflated static map
/*! Every point which is occupied on the updated map but not occupied in the inflated
    static map is occupied in this map. This map is the base for the detection of
    obstacles.
    \sa updatedMap_ \sa static_map_ \sa generateDifferenceMap \sa findObstacles */
costmap_2d::Costmap2D difference_map_;

//! This map is used in the process of finding and rematching of obstacles and is copy
    of difference_map_
/*! The usage of this map is not really sure. Each run it is a copy of the
    differenceMap_ and every processed occupied point is marked as FREE_SPACE.
    \todo Check if the dmap_new_ is really necessary. Probably not.*/
costmap_2d::Costmap2D dmap_new_;

//! The seen obstacles during a panorama turn are marked in this map.
/*! This map isn't used yet, but it should help to report which obstacles have been
    seen during a panorama turn.*/
costmap_2d::Costmap2D dmap_pano_;

//! Marks if the maps have been intialized by retrieving the map from the map topic.
/*! In order to start working it is very important to retrieve the map from the map
    topic first. The most functions of this node don't work if the maps haven't been
    initialized yet.
    \todo Add a warn if map_initialized is false */
bool map_initialized_;

costmap_2d::Costmap2DPublisher *pub_static_map_;                              //!<
    Publisher for the static map.  \sa static_map_
costmap_2d::Costmap2DPublisher *pub_updated_map_;                             //!<
    Publisher for the updated map representing the current obstacles \sa updated_map_
```

```
costmap_2d::Costmap2DPublisher *pub_difference_map_;                        //!< Publisher for the difference map \sa difference_map_
costmap_2d::Costmap2DPublisher *pub_dmap_new_;                              //!< Publisher for the difference map used to find obstacles \sa dmap_new_
costmap_2d::Costmap2DPublisher *pub_dmap_pano_;                             //!< Publisher for the dmap_pano_ \sa dmap_pano_

//! Storing the rotation velocity provided by the gyrometer
/*! The latest angular velocity in z-direction (rotation of the robot) is stored here. This is necessary because the occupied points received by the obstacle publisher of move_base are too noisy during a turn.
    \sa obstacleCallback_ */
double imu_ang_vel_z;

//! A queue storing all confirmations until they are processed
std::queue<person_detector::SpeechConfirmation> conf_queue_;

//! A vector storing the latest 30 amcl poses of the robot
/*! The poses have to be stored, because the storage of information from localCostmap needs the position of the robot.
    \sa findAmclPose_ */
std::vector<geometry_msgs::PoseWithCovarianceStamped> amcl_poses_;

//! All information about all current tracked obstacles are stored in here.
/*! All information about obstacles are stored in this array. It is frequently updated and later published. Each obstacle entry must have an corresponding entry in all_obs_map_xy_. So the size and the order of the obstacle vector in all_obstacles_ must be the same as in all_obs_map_xy_
    \sa findObstacles_ \sa all_obs_map_xy_ */
person_detector::ObstacleArray all_obstacles_;

//! This vector holds the corresponding points of an obstacle in the array of the costmap
/*! In order to avoid frequent conversion from metric map coordinates to the corresponding points in the array of the costmap, all occupied points of an obstacle are stored in here as well. The size and the order of this vector must be the same as the number of obstacle in all_obs_map_xy_
    \sa ObsMapPoints \sa all_obstacles_*/
std::vector<person_detector::ObsMapPoints> all_obs_map_xy_;
//global helperpoint
geometry_msgs::Point p_;                                                    //!< This point is globally used to avoid the creation of temporary ones. Always reset unused attributes!
geometry_msgs::Point p_map_xy_;                                             //!< This point is globally used to avoid the creation of temporary ones. Always reset unused attributes!
double x_map_;                                                              //!< This variable is globally used to avoid the creation of temporary ones.
double y_map_;                                                              //!< This variable is globally used to avoid the creation of temporary ones.

//functions
//! This function processes incoming detections to the right coordinate frame, rates and adds them to the global array.
/*! \returnÂ¸Sucess of the processing */
int processDetections();

//! This function takes incoming detections and calculates the distance to known ones.
```

```
  /*! \param detection_array The incoming detections
      \return 0 on success */
  int classifyDetections( cob_people_detection_msgs::DetectionArray detection_array );

  //! This function adds a incoming detection to the array of all detections
  /*! \param new_detection The new incoming detection which should be added
      \return 0 on success */
  int addNewDetection(cob_people_detection_msgs::Detection new_detection);

  //! This function updates a known detection with new information,
  /*! \param new_detection The incoming detection delivering information for the update
      \param pos The position in the all_detections_array_ for array access
      \return 0 on success*/
  int updateDetection(cob_people_detection_msgs::Detection new_detection, unsigned int
      pos);

  //! This function finds the closest known detection to a incoming detection
  /*! \param distances An array of distances
      \param win_id The ID of the closest known detection to each incoming detection. The
           same ID can appear several times!
      \param win_dist The winning distance for each pair in meter.
      \param detection_array_size The amount of incoming detections
      \return 0 on success
*/
  int findDistanceWinner(std::vector< std::vector <double> > &distances, std::vector<
      unsigned int> &win_id, std::vector<double> &win_dist, unsigned int
      detection_array_size);

  //! Checks if two incoming detections want to assign to the same known detection
  /*! \param distances The array of all distances between a incoming and a known
      detection
      \param win_id The ID of the closest known detection to an incoming detection
      \param win_dist  The shortest distance between the incoming detection and the
          known detection specified in win_id
      \param detection_array_size The amount of known incoming detections
      \return 0 on success*/
  int clearDoubleResults(std::vector< std::vector <double> > &distances, std::vector<
      unsigned int> &win_id, std::vector<double> &win_dist, unsigned int
      detection_array_size);

  //! Substracts an hit of a name on every other DetectionObject
  /*! \param label The newly detected name
      \param leave_id The ID the name was newly assigned.
      \return 0 on success */
  int substractHit (std::string label, unsigned int leave_id);

  //! Deletes old face recognitions and detected obstacles
  /*! \param oldness The maximum lifetime and old object can have
      \return 0 on success */
  int garbageCollector (ros::Duration oldness);

  //! Prepares and sends visualization of the face recognitions to rviz
  void showAllRecognitions();

  //! Generates the difference map from the static map and the updated map
  /*! \return 0 on sucess*/
  int generateDifferenceMap();

```

```
  //! Updates known obstacles and finds new obstacles
  /*! \return 0 on success */
  int findObstacles();

  //! Recursive helper function searching for more occupied points around a specified
      point
  /*! \param orig_x The starting x position on the costmap
      \param orig_y The starting y position on the costmap
      \param costmap The costmap used for to search. Every found occupied point is going
          to be marked as FREE_SPACE on this costmap
      \param points Vector storing all found points in metric map coordiantes to update
          or create an obstacle object
      \param points_map_xy Vector storing all found point in costmap array coordinates
      \return sucess */
  bool searchFurther(unsigned int orig_x, unsigned int orig_y, costmap_2d::Costmap2D*
      costmap, std::vector<geometry_msgs::Point> *points, std::vector<geometry_msgs::
      Point> *points_map_xy );

  //! Helper function rating an obstacle on a scale from 0 to 100
  /*! \param obs Pointer to the obstacle that should be rated
      \param map_points Pointer to the corresponding points in the costmap array
          coordinates
      \return sucess */
  bool rateObstacle(person_detector::Obstacle *obs, person_detector::ObsMapPoints *
      map_points);

  //! Helper function to calculate the geometric center of a set of points
  /*! \param points The points used for calculation
      \param pose   The returned pose
      \return success  */

  bool calculateCenter(std::vector<geometry_msgs::Point> points, geometry_msgs::Pose &
      pose);

  //! Prepares and sends visualization of the obstacles to rviz
  void showAllObstacles();

  //! Inflates occupied points on a received static map by 10cm
  int inflateMap();

  //! Updates known obstacles and known face recognitions with incoming confirmation
      information
  int processConfirmations();

  //! Finds the best fitting robot position to a specified time
  /*! \param pose The returned pose
      \param stamp The time the pose should match
      \result False if no poses are stored. True if a pose could be found*/
  bool findAmclPose (geometry_msgs::PoseWithCovarianceStamped &pose, ros::Time stamp);

public:
  //! Constructor initializing subscriber, publisher and marker
  person_detector_class();
  //! Runs endless and manages the whole detection process
  int run();
};

#endif // PERSON_DETECTOR_H
```

Listing A.4: Header file of the person detection package

A.3 Exploration Node

The header file of the module which is coordinating the search.

```
#ifndef EXPLORATION_HH_H
#define EXPLORATION_HH_H

#include <ros/ros.h>                                  // needed for general ROS-support
#include <robot_control/RobotControlSimpleClient.h>   // class inherits from the simpleclient
#include <vector>                                     // needed to store the goals
#include <std_srvs/Empty.h>                           //
#include <std_msgs/Empty.h>                           //
#include <exploration_hh/ExplorationGoal.h>           // exploration messagetype
#include <move_base_msgs/MoveBaseAction.h>            // to make a move-base-client
#include <actionlib/client/simple_action_client.h>    // --
#include <person_detector/DetectionObjectArray.h>     // to process and store the detections
#include <person_detector/ObstacleArray.h>            // to process and store the obstacles
#include <visualization_msgs/Marker.h>                // to show state in rviz
#include <sensor_msgs/Image.h>                        // to store the panorama images
#include <image_transport/image_transport.h>          // to subscribe to image topics
#include <cv_bridge/cv_bridge.h>                       // to save pictures
#include <opencv/cv.h>                                // to save pictures
#include <opencv/highgui.h>                           // to save pictures
#include <human_interface/RecognitionConfirmation.h>   // for confirmation of a person
#include <database_interface/postgresql_database.h> // to be able to use the database

//just here for non permanent purposes
namespace human_interface {

  struct speechRec {
    ros::Time time;
    std::string sentence;
  };

  enum yes_no_result {
    ANSWERED = 0,
    UNANSWERED = 1,
    WRONG_ANSWER = 2,
    BLOCKED_SPEAKER = 3
  };
}

namespace exploration_hh
{
  //! Enum to describe the states of the state machine
  /*! This node is a state machine switiching between these states */
  enum state
```

```
  {
    IDLE,                                      //!< Nothing to do. No more goals
    EXPLORATION,                               //!< The current goal is an exploration goal
    FACE_RECOGNITION,                          //!< The current goal is a goal for a recognized face
    CONFIRMATION,                              //!< The confirmation of an obstacle or a recognized face takes place
    OBSTACLE,                                  //!< The current goal is an obstacle goal
    PANORAMA,                                  //!< A panorama picture is taken
    PHOTO,                                     //!< Needed to wait a short moment for the picture
    FOUND                                      //!< Found the right person
  };
  //! Enum to distinguish different kind of goals
  /*! Every goal has to be of one kind.*/
  enum goal_type
  {
    EXPLORATION_GOAL = 0, /*!< A goal sent by the database */
    RECOGNITION_GOAL = 1, /*!< A goal for a recognized face */
    OBSTACLE_GOAL = 2     /*!< A goal for a recognized obstacle */
  };
  //! A struct to save an panorama image with metainformation
  /*! This can be used for later purposes. So this may be transfered into a ROS message.
   */
  struct img_meta
  {
    int id;                                    //!< A unique ID
    sensor_msgs::Image img;                    //!< The picture
    geometry_msgs::Pose robot_pose;            //!< The pose of the robot when picture started to be taken
    int obstacle_id;                           //!< The ID the visible obstacle
    geometry_msgs::Pose obstacle_pose;         //!< The pose of the visible obstacle
    int face_detection_id;                     //!< The ID of the visible face recognition
    geometry_msgs::Pose face_detections_pose;  //!< The pose of the visible face detection
  };
}
/*! The Exploration class is able to coordinate and influence the search for a person */


class Exploration : public RobotControlSimpleClient
{
private:
  //ROS and Markers
  //ros::NodeHandle n_;                        //!< Mandatory ROS-Nodehandler
  ros::Subscriber sub_exploration_goals_;      //!< Subscriber for database-given exploration goals
  ros::Subscriber sub_detections_;             //!< Subscriber for person_detector face detections
```

```
    ros::Subscriber sub_obstacles_;                                     //!< Subscriber
        for person_detector obstacles
    ros::ServiceClient confirmation_client_;                            //!< Client for
        human_interface confirmation requests
    ros::ServiceClient yes_no_client_;                                  //!< Client for
        human_interface yes-no-questions
    ros::Publisher pub_speech_;                                         //!< Publisher
        for human_interface text-to-speech requests
    ros::Publisher pub_confirmations_;                                  //!< Publisher
        for person_detector confirmations
    ros::Publisher pub_pano_start_;                                     //!< Publisher
        to start a panorama picture
    ros::Publisher pub_pano_stop_;                                      //!< Publisher
        to stop a panorama picture
    image_transport::Subscriber *sub_pano_;                             //!< Subscriber
        for the panorama image topic
    image_transport::Subscriber *sub_img_;                              //!< Subscriber
        to the image topic
    actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>* ac_; //!< Client for
        navigation goals
    ros::Publisher pub_point_marker_;                                   //!< Publisher
        for rviz goal-cubes
    visualization_msgs::Marker point_marker_;                           //!< Internal
        storage for rviz goal-cubes. This is used to avoid the initialization.
    ros::Publisher pub_text_marker_;                                    //!< Publisher
        for rviz goal-text
    visualization_msgs::Marker text_marker_;                            //!< Internal
        storage for rviz goal-text

    //Database
    //! A database object for the connection to the PostgresqlDatabase
    /*! It holds the connection, can report about its state and do queries.*/
    database_interface::PostgresqlDatabase* database_;

    //own-variables
    std::vector<exploration_hh::ExplorationGoal> exploration_goals_;    //!< Internal
        storage for exploration goals
    std::vector<exploration_hh::ExplorationGoal> recognition_goals_;    //!< Internal
        storage for face recognition goals
    std::vector<exploration_hh::ExplorationGoal> obstacle_goals_;       //!< Internal
        storage for obstacle goals

    //! The ordered goals after they have been ordered by calcGoals()
    /*! The first item of this vector is always the next goal. If it changes, the new first
         item will be set as the next goal and the state changes according of the state of
        that goal.
    */
    std::vector<exploration_hh::ExplorationGoal*> ordered_goals_;
    exploration_hh::ExplorationGoal* current_goal_;                     //!< Holds a
        pointer to the current goal
    move_base_msgs::MoveBaseGoal move_base_goal_;                       //!< The
        navigation goal sent to the movebase

    //! Holds all detections of the person_detector
    /*! This object is frequently replaced by a new array of detections sent from the
        person_detector. Don't change anything or store any data in there. Use the
        extracted ExplorationGoal s or send a person_detector::confirmation message to the
        person detector if you need to update anything.*/
```

```
person_detector::DetectionObjectArray detections_;

//! Holds all detected obstacles of the person_detector
/*! This object is frequently replaced by a new array of detections sent from the
    person_detector. Don't change anything or store any data in there. Use the
    extracted ExplorationGoal s or send a person_detector::confirmation message to the
    person detector if you need to update anything.*/
person_detector::ObstacleArray obstacles_;
int goal_counter_;                                              //!< Counter to
    give every new goal a unique ID

//! The name of the person, we're searching right now
/*! If the search doesn't have a name, this string is empty */
std::string name_;

//! Saves the current state of the node
/*! The node has several states defined in the exploration_hh::state enum. Whenever the
     state is changed, this variable has to be updated.*/
exploration_hh::state node_state_;
int speech_confirmation_id_;                                   //!< Counter to
    give human_interface confirmations unique IDs

//! The threshold for accepting obstacle goals
/*! Obstacle goals get on a scale from 0 to 100 points. It is possible to set this
    threshold to accept just interesting goals. Be careful setting this variable. If it
    's too low the search process will drown in obstacle goals. It it's too high it
    will cause false negatives. A reasonable value for quite a lot of goals is 40. A
    balanced value could be 50. A high value with some false negatives could be 60.
    \sa erase_threshold_ */
int accept_threshold_;

//! The threshold for deleting obstacle goals
/*! If an obstacle turns out to be a wrong detection or if its size shrinks the
    corresponding obstacle goal should be deleted. It is possible to set a treshold for
     that, but be carefull setting it. It is recommended to keep a distance (e.g. 10
    points) from Exploration::accept_treshold_ in order to avoid goals appearing and
    disappearing all the time.
    \sa accept_treshold_ */
int erase_threshold_;

//! A counter for the panorama images we take
/*! This counter also affects the filenames */
int image_counter_;
sensor_msgs::ImageConstPtr tmp_picture;
std::vector<exploration_hh::img_meta> images_;                //!< Storage of
    all images with metainformation
image_transport::ImageTransport imageTransport_;              //!< Needed to
    connect to an sensor_msgs::Image stream

bool image_taken_;
bool image_running_;
bool panorama_taken_;
bool panorama_running_;

//own-functions are documented in the cpp
//! The callback for a new exploration goal sent by the database
/*! \param received_goal The new goal received from the database */
void explorationGoalCallback(const exploration_hh::ExplorationGoal received_goal);
```

```
//! The callback for the person_detector face detections
/*! \param rec The received array of detections */
void detectionsCallback(const person_detector::DetectionObjectArray rec);

//! The callback for the person_detector obstacle detections
/*! \param obs The received array of obstacles */
void obstacleCallback(const person_detector::ObstacleArray obs);

//! The callback for the panorama image
/*! \param img The received image */
void panoramaCallback(const sensor_msgs::Image::ConstPtr& img);

//! The callback for the color image
/*! \param img The received image */
void imageCallback(const sensor_msgs::Image::ConstPtr& img);

//! This function orders all goals and creates a new ordered_goals_ vector
/*! \return sucess or not */
bool calcGoals();

//! This functions prepares and publishes the information vor rviz
void showGoals();

//! Goes through the latest face detection array and updates and add goals
/*! \sa recognition_goals_ detections_ */
bool processDetections();

//! Goes through the latest obstacle array and updates and adds the goals
/*! \sa obstacle_goals_ obstacles_ */
bool processObstacles();

//! Sets the next goal of the ordered_goals vector and changes the state
/*! \sa ordered_goals_ current_goal_ */
void setGoal();

//! Called function in the state CONFIRMATION with current_goal_ = OBSTACLE_GOAL
int confirmation_face();

//! Called function in the state CONFIRMATION with current_goal_ = RECOGNITION_GOAL
int confirmation_obstacle();

//! Called function in the state RECOGNITION
int recognitionGoal_();

//! Called function in the state EXPLORATION
int explorationGoal_();

//! Called function in the state OBSTACLE
int obstacleGoal_();

//! Called function in the state PANORAMA
int panorama_();

//! Called function in the state FOUND
void found();

//! Called function if we are in PHOTO
```

```
  void photo();

  //! Used to get a list of places to a task
  /*! \return true on success */
  bool getPlaces();

  //! Used to calculate where the robot should go in order to speak with the person
  /*! \param robot_pose The place from which the robot saw the interesting point
      \param int_place  The interesting place
      \param goal       Where the robot should go to for a conversation
      \return success*/
  bool calcGoalPlace(geometry_msgs::Pose* robot_pose, geometry_msgs::Pose* int_place, geometry_msgs::Pose &goal);

  //! Check for incoming goals if database data is available
  /*! \return true if the new goal will be accepted */
  bool checkIncomingGoal(robot_control::RobotTaskGoalConstPtr goal, robot_control::RobotTaskResult &res);

  bool cleanupCancelledGoal(robot_control::RobotTaskResult &res);

  //! Cleanup after finishing the task
  void finishTask(bool success, std::string res);

public:

    //! Constructor - intializes the class
    Exploration(std::string task_server_name, std::string task_name);

    //! Run loop which runs endless
    int run();
};

#endif // EXPLORATION_HH_H
```

Listing A.5: Header file of the search coordination

A.4 Human Interface Node

The human interface node offers basic functionalities for the interaction with a human. It is self implemented and uses the 'sound_play' package for text-to-speech and 'pocket-sphinx' for speech recognition.

```
#ifndef HUMAN_INTERFACE_H
#define HUMAN_INTERFACE_H
#include <ros/ros.h>            // general ros functionalities
#include <string>
#include <std_msgs/String.h>
#include <human_interface/SpeechRequest.h>
#include <human_interface/RecognitionConfirmation.h>
#include <human_interface/YesNoQuestion.h>
```

```
#include <queue>                                    //to store speech recognition results

//be carefull to modify the ./include/human_interface/enums.h as well
namespace human_interface {
  //! Used to add a time to incoming sentences of the speech recognition
  struct speechRec {
    ros::Time time;
    std::string sentence;
  };
  //! Defines the states an answer according to the message file
  enum yes_no_result {
    ANSWERED = 0,
    UNANSWERED = 1,
    WRONG_ANSWER = 2,
    BLOCKED_SPEAKER = 3
  };
}

//! Stand alone node to allow text-to-speech, process yes-no-questions and do confirmations
class human_interface_class
{
private:
  //ros-stuff
  ros::NodeHandle n_;                               //!< Mandatory nodehandle
  ros::Publisher pubRobotSounds_;                   //!< Publisher for the text-to-speech soundplay-node
  ros::Subscriber subSpeechRequests_;               //!< Subscriber to receive text-to-speech requests
  ros::ServiceServer yesNoServer_;                  //!< Server for yes-no-questions
  ros::Subscriber subSpeechRecog_;                  //!< Subscriber to the recognized speech outputs of pocketsphinx
  ros::ServiceServer confirmationServer_;           //!< Server for confirmations

  //speech
  bool speakers_in_use_;                            //!< True if the speakers are used (probably not need on a single thread program
  std::queue <human_interface::speechRec> speech_q; //!< A queue for all recognized sentences with timestamps

  //! Forms the string to the tts-message and waits until its said
  /*! \param text_to_say Text which should be sent */
  void say_(std::string text_to_say);

  //! Server function for confirmation requests
  /*! \param req Received request
      \param res Returned result to the requesting client
      \return Passed to the client of the request */
  bool recognitionConfirmation(human_interface::RecognitionConfirmation::Request &req, human_interface::RecognitionConfirmation::Response &res);

  //! Server function for the yes-no-questions
  /*! \param req Received request
      \param res Returned result to the asking client
      \return Passed to the client of the request */
  bool yesNoQuestionService(human_interface::YesNoQuestion::Request &req, human_interface
```

```
        ::YesNoQuestion::Response &res);

  //! Implementation of the yes-no-question
  /*! \param question Question string
      \param answer   True if the answer is yes
      \param status   Following the message definition */
  void yesNoQuestion(std::string question, bool &answer, int &status);

  //! Callback for received speech recognitions
  /*! \param speech The recognized sentence  */
  void speechRecognitionCallback_ (const std_msgs::String speech);

  //! Callback for speech requests
  /*! \param req Received request object */
  void speechRequestCallback_(human_interface::SpeechRequest req);

  //! Function to get access to the speaker
  /*! \todo Probably useless in a single threaded system
      \param max Maximal duration to wait
      \return true if access is granted */
  bool getSpeakers(ros::Duration max);
public:
  //! Constructor initializing subscriber and publisher
  human_interface_class();

  //! Runs the class and never exits except in critical errors
  int run();
};

#endif // HUMAN_INTERFACE_H
```

Listing A.6: Header of the human interface

A.5 SQL Database Client

A.5.1 Integration of a Database Binding

To integrate a database binding into a module the ROS package 'sql_database' [1] must be included.

```
#include <database_interface/postgresql_database.h> // to be able to use
    the database

std::string host,port,user,passwd,db;
```

[1] The latest modified package can be found on: https://github.com/matthiashh/sql_database
The latest upstream package can be found on: https://github.com/ros-interactive-manipulation/sql_database
The upstream package does not yet include all necessary code

```
if (!n_.getParam("/database/hostname",host) ||
    !n_.getParam("/database/port",port) ||
    !n_.getParam("/database/db_user",user) ||
    !n_.getParam("/database/db_passwd",passwd) ||
    !n_.getParam("/database/db_name",db))
{
    ROS_ERROR("Couldn't get all parameters for the database connection. Did
        you start robot_control and got a connection?");
    ROS_ERROR("All database related tasks won't work.");
}
else
{
  ROS_INFO("Trying to connect with host %s, port %s, user %s, passwd %s, db
      %s",host.c_str(), port.c_str(),user.c_str(),passwd.c_str(),db.c_str()
      );
  database_interface::PostgresqlDatabase database(host,port,user,passwd,db)
      ;
}
```

Listing A.7: Code to integrate a database binding into a module. 'n_' is the ROS nodehandle of that executable

A.5.2 PostgreSQL Database Header

The PostgreSQL database header is part of the ROS package 'sql_database' and has been modified and enhanced in order to fit the requirements of the project.

```
/*********************************************************************
* Software License Agreement (BSD License)
*
*  Copyright (c) 2009, Willow Garage, Inc.
*  All rights reserved.
*
*  Redistribution and use in source and binary forms, with or without
*  modification, are permitted provided that the following conditions
*  are met:
*
*   * Redistributions of source code must retain the above copyright
*     notice, this list of conditions and the following disclaimer.
*   * Redistributions in binary form must reproduce the above
*     copyright notice, this list of conditions and the following
*     disclaimer in the documentation and/or other materials provided
*     with the distribution.
*   * Neither the name of the Willow Garage nor the names of its
*     contributors may be used to endorse or promote products derived
*     from this software without specific prior written permission.
*
*  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

```
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *********************************************************************/

// Author(s): Matei Ciocarlie

#ifndef _POSTGRESQL_DATABASE_H_
#define _POSTGRESQL_DATABASE_H_

#include <vector>
#include <string>
#include <list>
#include <boost/shared_ptr.hpp>

//for ROS error messages
#include <ros/ros.h>
#include <yaml-cpp/yaml.h>

#include "database_interface/db_class.h"
#include "database_interface/db_filters.h"

//A bit of an involved way to forward declare PGconn, which is a typedef
struct pg_conn;
typedef struct pg_conn PGconn;

namespace database_interface {

//this is passed over to a function called holding all the informations which should be
    submitted
struct FunctionCallObj {
  std::string name;
  std::vector<std::string> params;
};

//this is used to pass the information stored in a received notification event
struct Notification {
  std::string channel;
  int sending_pid;
  std::string payload;
};

class PostgresqlDatabaseConfig
{
private:
  std::string password_;
  std::string user_;
  std::string host_;
  std::string port_;
  std::string dbname_;
```

```
public:
  PostgresqlDatabaseConfig() {  }

  std::string getPassword() const { return password_; }
  std::string getUser() const { return user_; }
  std::string getHost() const { return host_; }
  std::string getPort() const { return port_; }
  std::string getDBname() const { return dbname_; }

  friend void operator>>(const YAML::Node &node, PostgresqlDatabaseConfig &options);
};

/*!
 *\brief Loads YAML doc into configuration params. Throws YAML::ParserException if keys
    missing.
 */
inline void operator>>(const YAML::Node& node, PostgresqlDatabaseConfig &options)
{
  node["password"] >> options.password_;
  node["user"] >> options.user_;
  node["host"] >> options.host_;
  node["port"] >> options.port_;
  node["dbname"] >> options.dbname_;
}

class PostgresqlDatabase
{
 protected:
  void pgMDBconstruct(std::string host, std::string port, std::string user,
                      std::string password, std::string dbname );

  //! The PostgreSQL database connection we are using
  PGconn* connection_;

  //! Helper class that acts like an auto ptr for a PGresult, with a little more cleanup
  class PGresultAutoPtr;

  // beginTransaction sets this flag. endTransaction clears it.
  bool in_transaction_;

  //! Stores all channels, which the instance listens
  std::list<std::string> channels_;

  //! Gets the text value of a given variable
  bool getVariable(std::string name, std::string &value) const;

  //! Issues the "rollback" command to the database
  bool rollback();

  //! Isses the "begin" command to the database
  bool begin();

  //! Issues the "commit" command to the database
  bool commit();

  //! Retreives the result of a function call in a certain type
  template <class T>
```

```
  bool callFunction(std::vector< boost::shared_ptr<T> > &objVec, const T& example,
      FunctionCallObj paramVec) const;

  //! Helper function for callFunction, separates SQL from (templated) instantiation
  bool callFunctionRawResult(const DBClass *example, std::vector<const DBFieldBase*> &
      fields,
                            std::vector<int> &column_ids, FunctionCallObj paramVec,
                            boost::shared_ptr<PGresultAutoPtr> &result, int &num_tuples)
                                const;

  //! Retreives the list of objects of a certain type from the database
  template <class T>
    bool getList(std::vector< boost::shared_ptr<T> > &vec, const T& example, std::string
        where_clause) const;

  //! Helper function for getList, separates SQL from (templated) instantiation
  bool getListRawResult(const DBClass *example, std::vector<const DBFieldBase*> &fields,
      std::vector<int> &column_ids, std::string where_clause,
      boost::shared_ptr<PGresultAutoPtr> &result, int &num_tuples) const;

  //! Helper function for getList, separates SQL from (templated) instantiation
  bool populateListEntry(DBClass *entry, boost::shared_ptr<PGresultAutoPtr> result, int
      row_num,
        const std::vector<const DBFieldBase*> &fields,
        const std::vector<int> &column_ids) const;

  //! Returns the 'curval' for the database sequence identified by name
  bool getSequence(std::string name, std::string &value);

  //! Helper function for inserting an instance into the database
  bool insertIntoTable(std::string table_name, const std::vector<const DBFieldBase*> &
      fields);

  //! Helper function that deletes a row from a table based on the value of the specified
        field
  bool deleteFromTable(std::string table_name, const DBFieldBase *key_field);

 public:
  //! Attempts to connect to the specified database
  PostgresqlDatabase(std::string host, std::string port, std::string user,
          std::string password, std::string dbname);

  //! Attempts to connect to the specified database
  PostgresqlDatabase(const PostgresqlDatabaseConfig &config);


  //! Closes the connection to the database
  ~PostgresqlDatabase();

  //! Returns true if the interface is connected to the database and ready to go
  bool isConnected() const;

  //! Reconnects to the database. For example if the connection is lost
  void reconnect();

  //------- general queries that should work regardless of the datatypes actually being
      used ------

```

```
//——— calling a user defined function ———
template <class T>
bool callFunction(std::vector< boost::shared_ptr<T> > &objVec, std::string func) const
{
  T example;
  FunctionCallObj paramVec;
  paramVec.name = func;
  return callFunction<T>(objVec, example, paramVec);
}

template <class T>
bool callFunction(std::vector< boost::shared_ptr<T> > &objVec, FunctionCallObj paramVec
    ) const
{
  T example;
  return callFunction<T>(objVec, example, paramVec);
}


//——— retrieval without examples ———
template <class T>
bool getList(std::vector< boost::shared_ptr<T> > &vec) const
{
  T example;
  return getList<T>(vec, example, "");
}
template <class T>
bool getList(std::vector< boost::shared_ptr<T> > &vec, const FilterClause clause) const
{
  T example;
  return getList<T>(vec, example, clause.clause_);
}
template <class T>
bool getList(std::vector< boost::shared_ptr<T> > &vec, std::string where_clause) const
{
  T example;
  return getList<T>(vec, example, where_clause);
}

//——— retrieval with examples ———
template <class T>
bool getList(std::vector< boost::shared_ptr<T> > &vec, const T &example) const
{
  return getList<T>(vec, example, "");
}
template <class T>
bool getList(std::vector< boost::shared_ptr<T> > &vec, const T &example, const
    FilterClause clause) const
{
  return getList<T>(vec, example, clause.clause_);
}

//! Counts the number of instances of a certain type in the database
bool countList(const DBClass *example, int &count, std::string where_clause) const;

//! templated implementation of count list that works on filter clauses.
template <typename T>
bool countList(int &count, const FilterClause clause=FilterClause()) const
```

```
  {
    T example;
    return countList(&example, count, clause.clause_);
  }

  //! Writes the value of one particular field of a DBClass to the database
  bool saveToDatabase(const DBFieldBase* field);

  //! Reads the value of one particular fields of a DBClass from the database
  bool loadFromDatabase(DBFieldBase* field) const;

  //! Inserts a new instance of a DBClass into the database
  bool insertIntoDatabase(DBClass* instance);

  //! Deletes an instance of a DBClass from the database
  bool deleteFromDatabase(DBClass* instance);

  //! Enables listening to a specified channel
  bool listenToChannel(std::string channel);

  //! stop listening to a specified channel
  bool unlistenToChannel(std::string channel);

  //! Checks for a notification
  bool checkNotify(Notification &no);

  //! Checks for a notification, but idles and exits when we have one
  bool checkNotifyIdle(Notification &no);

};

template <class T>
bool PostgresqlDatabase::callFunction(std::vector< boost::shared_ptr<T> > &objVec,
                                      const T &example, FunctionCallObj paramVec) const
{
  //we will store here the fields to be retrieved retrieve from the database
  std::vector<const DBFieldBase*> fields;
  //we will store here their index in the result returned from the database
  std::vector<int> column_ids;
  boost::shared_ptr<PGresultAutoPtr> result;

  int num_tuples = 0;

  if (!callFunctionRawResult(&example, fields, column_ids, paramVec, result, num_tuples))
    {
      return false;
    }

  objVec.clear();
  if (!num_tuples)
  {
    return true;
  }

  //parse the raw result and populate the list
  for (int i=0; i<num_tuples; i++)
  {
    boost::shared_ptr<T> entry(new T);
```

```
    if (populateListEntry(entry.get(), result, i, fields, column_ids))
    {
      objVec.push_back(entry);
    }
  }

  return true;
}


/*! The datatype T is expected to be derived from DBClass.

  The example is used only to decide which fields should be retrieved from the database.
  Note that the primary key field is ALWAYS retrieved; you can expect the returned list
  to have the primary key set. Any other fields are retrieved ONLY if they are marked
  with syncFromDatabase in the example.

  Note that the example is not used to decide which instanced to retrieve (but only which
  *fields* of the instances). To retrieve only certain fields, you must use the where_clause.
  This is not ideal, as much functionality is hidden from the user who is not exposed
  to SQL syntax. For those functions where the external user needs the where_clause (even if
  he does not know it) we are currently providing public wrappers, but that  might change in
  the future.

  The significant difference between this function and the version that reads a
  certain field is that this function creates new instances of the DBClass and gives them
  the right values of the primary key. The function that reads a certain field expects the
  instance of DBClass to already exist, and its primary key field to be set correctly
  already.
*/
template <class T>
bool PostgresqlDatabase::getList(std::vector< boost::shared_ptr<T> > &vec,
         const T &example, std::string where_clause) const
{
  //we will store here the fields to be retrieved retrieve from the database
  std::vector<const DBFieldBase*> fields;
  //we will store here their index in the result returned from the database
  std::vector<int> column_ids;
  boost::shared_ptr<PGresultAutoPtr> result;

  int num_tuples;
  //do all the heavy lifting of querying the database and getting the raw result
  if (!getListRawResult(&example, fields, column_ids, where_clause, result, num_tuples))
  {
    return false;
  }

  vec.clear();
  if (!num_tuples)
  {
    return true;
  }

  //parse the raw result and populate the list
```

```
  for (int i=0; i<num_tuples; i++)
  {
    boost::shared_ptr<T> entry(new T);
    if (populateListEntry(entry.get(), result, i, fields, column_ids))
    {
      vec.push_back(entry);
    }
  }
  return true;
}


}//namespace

#endif
```

Listing A.8: Header of the modified database binding

A.5.3 Return object for tasks

This is the full object definition of the object which is needed to get the new tasks from the database. The returned table of a call for new tasks is stored in a vector of this object. For every kind of database call such an object has to be defined according to the returned columns and the expected variable types.

```
#include <string>
#include <vector>
#include <database_interface/db_class.h>

class returnTasks : public database_interface::DBClass
{
public:
  database_interface::DBField<int> id_;
  database_interface::DBField<int> task_id_;
  database_interface::DBField<std::string> task_name_;
  database_interface::DBField<int> priority_;

  returnTasks() :
    id_(database_interface::DBFieldBase::TEXT,this,"key_column","places2",
        true),
    task_id_(database_interface::DBFieldBase::TEXT,this,"task_id","places2"
        ,true),
    task_name_(database_interface::DBFieldBase::TEXT,this,"task_name","
        places2",true),
    priority_(database_interface::DBFieldBase::TEXT,this,"priority","
        places2",true)
  {
```

```
    primary_key_field_ = &id_;
    fields_.push_back(&task_id_);
    fields_.push_back(&task_name_);
    fields_.push_back(&priority_);
  }
};
```

Listing A.9: Definition of the object returned by the database binding on a call for new tasks