Construction Mobile Robots - WS19

Final Report

Hugo Leurent - 11917024, Arnav Bansal - 11926152, Fabian Hirmann - 01530583,
Gerald Strommer - 01530543, Michael Geigl - 01530327

1 Task description

The project for the Construction of Mobile Robots course is to have a robot automatically stacking blocks. The robot used is the Baxter robot by Rethink Robotics, and is programmed via ROS, in C++ or Python. Baxter's task is to be able to stack blocks of different shape and color depending on the scene that's in front of him. This might seem a simple task on the first glance, but this is way more developed than it looks.

For this task, 4 main parts could be distinguished : The robot needs to handle speech recognition to understand and analyze the vocal commands. It also outputs spoken messages. The second part is perception : by using one of Baxter's implemented cameras, the robot has to detect the scene. This means knowing exactly which blocks are in front of him, their color, shape, and positions. Thanks to those informations, the manipulation part can move the arm of the robot accordingly and perform the stacking of the blocks. Finally, the fourth part is at the center of it all. This is the reasoning and planning part, whose role is to analyze every incoming information from speech and perception, to compute the plan that the robot will follow to execute his actions. This is the part that links all the other elements together.

After explaining the install instructions to make the robot properly work, we will talk precisely about every tasks in each part of the report. We will explain how are the nodes working and how they are communicating to each other.

2 Install instructions

2.1 Perception

- Install ROS Kinetic
- Install Openni2 with
  - sudo apt-get install libopenni0 libopenni-sensor-primesense0 ros-kinetic-openni2-camera ros-kinetic-openni2-launch
- PCL should already be installed with ROS
- connect the Asus Xtion to a normal USB port on the PC (no USB3!!)

- lsusb should show you a device called ID 1d27:0601 ASUS
- calibrate the intrinsic parameters with the method from [4]
- calibrate the extrinsic parameters following the steps from [2]

2.2 Interaction per speech

- Set up the access key for the google speech API by writing the following into the .bashrc file:
  - export GOOGLE_APPLICATION_CREDENTIALS=$HOMEcatkin_wssrckmr19speech_reco
- Set up the Google Speech environment by following this steps from [3]
- Install python3 and pip3.
- Install the speech-to-text python library
  - pip3 install –upgrade google-cloud-speech
- If not automatically install, also install the portaudio, pyaudio, pyyaml, rospkg libraries.
- Install the nltk library
  - pip3 install -U nltk
- Nodes should be already set as executables. If not, go in the nodes folder and do :
  - chmod +x speech_request_node.py
  - chmod +x parser_client_node.py
- install java
  - sudo apt-get update
  - sudo apt-get install default-jre
  - sudo apt-get install default-jdk
- Download the stanford parser archive from [13]
- In a terminal go inside the StanfordNLP folder and launch the NLP server in the background with the command
  - java -mx4g -cp ”*” edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9000 -timeout 15000
- Install sox
  - sudo apt-get install sox soxlib-fmt-mp3
- Install google text to speech for python
  - pip3 install gTTs

2.3 Reasoning & planning

- install ROSPlan by following this steps [11]
- install mongoDB store for scene DB with
  - sudo apt install ros-kinetic-mongodb-store

2.4 Manipulation

To install the MoveIt Planning Framework and the configurations for the Baxter Robot, follow the instructions on
`https://github.com/RethinkRobotics/sdk-docs/wiki/MoveIt-Tutorial`. The *moveit_robots* folder contains all configurations for the robot.

The Baxter SDK can be installed by following the instructions in `https://github.com/RethinkRobotics/sdk-docs/wiki/Installing-the-Research-SDK`. Just use **catkin build** instead of **catkin_make**. The other program parts need **catkin build** and if you use **catkin_make** here, you have to initialize your catkin workspace again.

Finally build everything with catkin build. Do not use catkin_make!

3 Run instructions

If everything is correctly installed simple call:
`roslaunch reasoning_planning kmr19_baxter.launch`

4 Modules description

The block diagram in figure 1 shows the overall system architecture and interactions between components.

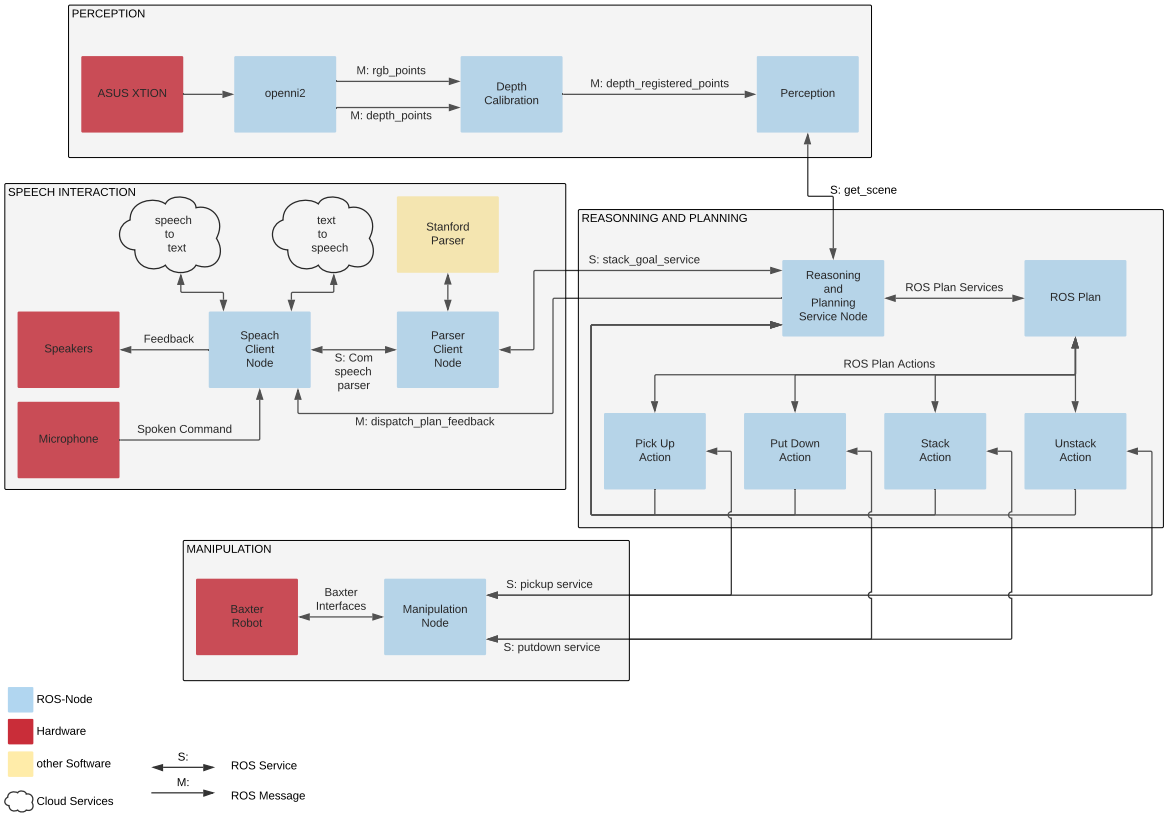


Figure 1: Overall System Architecture and Interaction.

The following chapters contain details about each module.

4.1 Perception

4.1.1 Key-Facts

- Responsible:
  - Michael Geigl
- Task:
  - The goal of the perception is to correctly detect and represent all blocks in the scene. In order to achieve this goal the task is split into many subsequent steps which are explained in the perception pipeline.
- Used libraries/3rd party modules

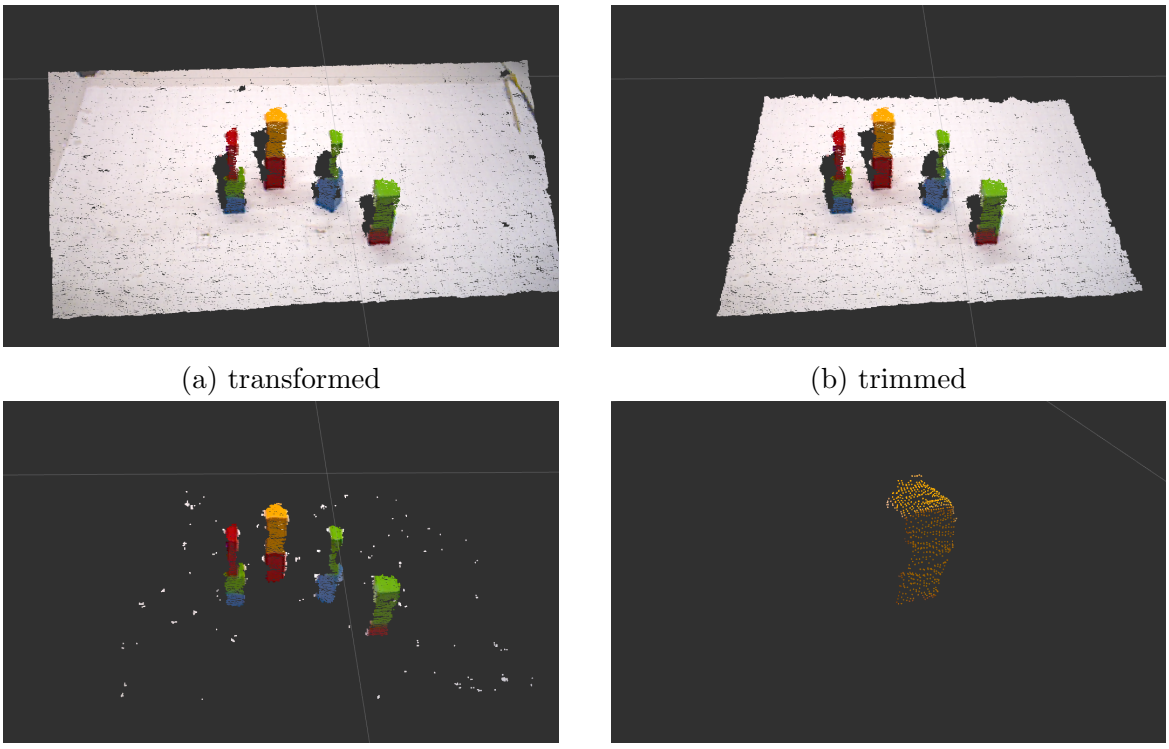
  - Depth calibration ROS-Package from [9], which is based on the work of [14] and [5]
  - Point Cloud Library (PCL) [8]

- Working hours:
  - around 100 hours

4.1.2 Perception pipeline

The first step is to take a "picture" of the current scene and get a useful point-cloud with a plane table out of it. The ASUS Xtion Live camera is used for taking the picture and the provided driver-ros-package, which is called openni2, delivers an RGBD point-cloud. Although the intrinsic parameter where correctly calibrated, in the outputted point-clouds the planes where curved instead of flat. Therefore, the depth image output from openni2 is send through the depth calibration ros-package and then gets overlaid with the RGB image. The output then is a RGBD point-cloud with a flat table plane.
This point-cloud is the input into the perception ros-package where at first the positions of the points get transformed via an tf transform to the table coordinate frame. The output after this transform can be seen in figure 2a.
The next step is to trim the whole picture to the area of interest which is a cuboid on the table in front of the robot where the blocks are allowed to be placed. This is implemented by simply using a pass-through-filter from the pcl which removes all points which lay outside of the given parameters. The result of this is pictured in figure 2b.
In the following step we use another filter from the pcl to remove all points which correspond to the table surface. It is the SACsegmentation filter with a plane model and RANSAC as method. This filter tries to fit the biggest possible plane model into the given point cloud and then removes all points which correspond to this model. Again the result is shown in figure 2c.
After the table is removed the remaining point-cloud gets split into a point-cloud for each block by grouping the points with the same color. To prepare the point-cloud for this segmentation the colorspace of the points gets changed from RGB to HSV because the color values for each block color were much easier to separated in the HSV colorspace. The separation process itself is realized with a conditional euclidean clustering filter from the pcl which uses a flood fill algorithm. This algorithm groups points together which fulfill the condition that the distance between two points is smaller then a given threshold and the condition that the points have the same color.
Although allot of effort got into the correct calibration the points of the point-cloud where distributed too far in all three axis. To compensate this behavior the x, y and z-axis values get scaled around the center point of the picture. This is simple math and is attached to the color segmentation function because there was already an access to each single point given. The output of this segmentation and scaling is shown in figure 2d.
The last step in the perception pipeline is the information extraction from this block point-clouds. The needed information is the color, the position, the orientation and the size. The color information is straight forward because the point-cloud got split by the

color value. For the other three informations different cases need to be considered. The blocks can stand alone which would deliver the top and front facing plane of the block. Another block can be stacked on top of this block which only leafs the front facing plane in the point cloud and the block can be rotated which can leaf two or one front facing planes. To handle this cases each block gets split into his planes with the same algorithm which was already used at the table remove step. The orientation is always calculated from the coefficients of the biggest front facing plane because each block needs to have one otherwise the camera would not see the block. Depending on how many planes got extracted and the coefficients of these planes a different method to get the position and size is used. If the orientation says that the block is straight to the camera and the front facing plane and the top plane is given then the wide is the delta of the points along the y-axis, the depth is the delta along the x-axis, the height is the delta along the z-axis and the position is delta half plus the minimum value along each axis. If only the front facing plane is available and the orientation is straight to the camera, the width and the height are calculated the same way as before and for the depth we look up in the block types we have defined. If the width and height matches one type it returns the corresponding block depth and with the size of the block and one plane given also the position can be calculated. If the block is rotated the block widths and depths take the rotation angle into account and multiply it, using sine and cosine functions, with the delta along the x and y-axis.



(a) transformed

(b) trimmed

(c) table removed

(d) block extracted

Figure 2: filter sequence

4.1.3 Problems

The basic idea was simple and looked as the implementation would be straight forward but during the process we faced many problems. The biggest of them are shortly introduced here.
The most time consuming problem was the correct calibration of the camera. We first calibrated the RGB and Depth sensors intrinsic parameters by using the ros camera calibration package [10]. Generally the ASUS Xtion Live has a mode where the depth registration of the points, the camera corrects the depth distortions and overlays the RGB image, is calculated on the hardware. This mode should already deliver a point cloud with no lens distortion and the RGB image perfectly aligned. The second part is true but the first part was not which delivered a bended table instead of a flat one. The openni2 driver also delivers the option to deactivate the hardware depth registration and use the built in software registration. The problem with this software registration method was that the range of the parameters was limited before the result was good enough. I guess thats because the driver was built for the Kinect sensor. Finally, I found the already mentioned ros calibration package which needed one round with an empty table to calculate the distortion parameters and now delivers a flat table. But the RGB image was not good aligned which cost some extra search to find the RGB link transformer and trim it to fit at least sufficiently.
And thats not enough from the calibration side, because the points where still to far distributed in the space which leaded to too big block sizes and drifting position approaching the edges of the scene. This was countered with a scaling of the axis which was also already mentioned above.
Another problem was that the planes of the blocks where noisy planes and sometimes had randomly holes in it. With a smoothing filter from the pcl it good a bit better but at some cases also this filter was not able to improve it. I guess this is caused by the not optimal mounting position of the camera. If the camera gets lower and closer to the scene this would improve but that would have meant to redo the calibration and that was too much for this project.

4.2 Interaction by speech

4.2.1 Key-Facts

- Responsible:
  - Hugo Leurent and Arnav Bansal
- Task:
  - Input vocal commands from a user, parse them and send them to the reasoning and planning module. In the opposite way, getting back the status messages from reasoning and planning, and interpreting them into speech.
- Used libraries/3rd party modules
  - Google Speech-To-Text API for Python

  - Google Text-To-Speech API for Python
  - Stanford parser library, in Java
  - Multiple libraries to handle audio (pyaudio, portaudio, pyyaml)

- Working hours:
  - around 60 hours (Hugo Leurent)
  - around 40 hours (Arnav Bansal)

4.2.2 The speech pipeline

The main parts of our package can be launched through the baxter_speech_recognition.launch roslaunch file . This file will run 3 necessary nodes to our program:

The first node, called speech_request_node.py will handle everything directly related to speech. This means it has two tasks : capturing the audio data from the microphone, but also handling the text-to-speech. At first, the Speech-to-Text part is started, and captures every surrounding sound. Once the command *Hello Baxter* is heard by the node, it will orally indicate us that the next sentance will be the request. When the sentance is captured by the program, it will open a client-server connection with our second node, our node, as a client will send him his request sentance. As a response, we will receive a list of errors given by the reasoning-planning nodes. If the list is empty, this means that the request was understood and is coherent with the scene detected by the perception. Else, we will know which kind of error we're facing, and which block it is about (the top block, below block, or the scene in general). This error codes will be translated into sentances and sent to the text-To-Speech module also handled by this node. This way, the Baxter robot can give us an oral information about the nature of our request. Finally, we also opened a listener in this node, directly communicating with the dispatch_plan_feedback talker opened by the reasoning and planning part. This message is being sent after the robot performed his actions, to tell us how it went. This way, we can know if the robot executed the action correctly or if errors happened during the manipulation. Again, this is sent into the text-To-Speech to inform the user of the situation. After that, the user will be able to perform a new request by triggering the program once more with the keywords *HelloBaxter*.

The second node called parser_client_node.py will be in charge of communicating with the Stanford Parser to extract important information from the request. As said before, it is being triggered once receiving the request from our previous node. The message will be sent into the Stanford Parser and a tree-shaped request will be given back to us. This way, we can easily extract information concerning the two blocks we want. We then convert the important info (color and positioning) into the correct semantics, and send this to the reasoning and planning part by also using a client-service connection. As a response, we get the error codes from the scene. We send them back to our first node to be handled.

Finally, the third node is launching a bash script. This one will be in charge of launching the Stanford Parsing Server, that allows us to perform the conversion of the request into a Tree-shaped parsed message. Adding this script allows us not to launch the stanford server separately every time.
To understand better the whole pipeline, 3 is a schematic summarizing the process.

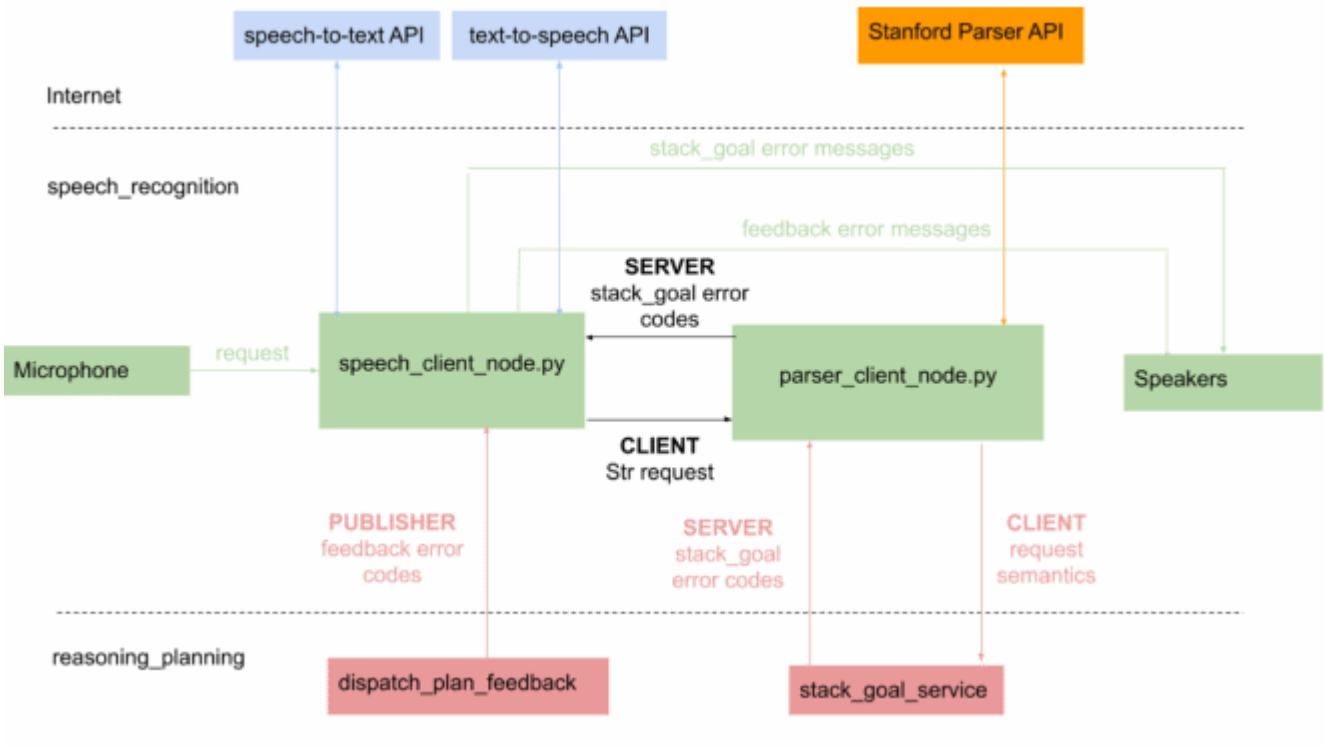


Figure 3: Schematic of the communications in the speech module

4.2.3 Problems and possible improvements

Problems encountered Multiple problems have been encountered during the whole projects. Some could have been solved, some others not. The first one is about which speech recognition module to use. The base idea was to use the functionnalities given by the Amazon Web Services. But after trying different methods and following tutorials, we couldn't manage to get anything out of it. It took us a few weeks to give up and start using the Google API, which was way more easy to use and documented.

Another problem we encountered later was the need of accessing Internet while communicating with the Baxter robot. Speech recognition is the only part that needs Internet in the project, because of the constant communication with the Google and Stanford APIs. However, the lab computer couldn't handle being connected to Internet and to Baxter at the same time. We managed to connect both but we still couldn't listen to the topics sent by Baxter. After investigating into the problem, we noticed it was an issue with the Baxter hostname that wasn't the same everywhere. We finally could solve this

problem by changing the /etc/hosts file.

Some annoying unsolved bugs are still hidden in our program. The Google API does not allow the program to be up more than 305 seconds in a row, which will lead to rebooting it often. Also, the speech recognition can't really be deactivated when we want, which sometimes leads to the microphone multiple requests, sometimes even hearing the output from the text-to-speech module. This creates an awful loop and we have to restart the program. The problem is that the speed of the speech processing is strongly dependant of the quality of the internet connection. If it works too poorly, the user will tend to think the program crashed when it's just taking a long time to process. Apart from that, the nodes work well in general, but that could obviously use some improvements

Possible improvments The main improvements that could be handled are stabilizing the program in general. Deactivating the speech-to-text tool when this is not needed, to avoid hearing useless requests. Also, finding a quicker way to implement the requests, since we need everytime the robot to tell us when he is ready after we said the keywords.Also, this could be better not to have to tell the whole request again when an error is displayed, but to talk only about the concerned block for the next request. To summarize : make the conversation more fluid.

Some would tend to say Baxter is a dude. However, the output voice is a girl's voice. This can't be changed with the way we're handling text-to-speech actually. We would need to use a different library that gives us more freedom when it comes about changing the voice.

4.3 Reasoning & planning

4.3.1 Key-Facts

- Responsible:
  - Fabian Hirmann
- Task:
  - Fetch goal from speech module, interpret scene from perception module, plan simple actions to reach goal, communicate with manipulation module to perform the simple actions and return feedback of goal execution to speech module
- Used libraries/3rd party modules
  - ROSPlan [1, 12]
    - Complete framework to define planning domain, goal definition and execution of atomic actions
  - MongoDB [6] + mongodb_store [7] as interface to ROS

        * MongoDB used as scene database to store the blocks from the perception

- Needed working hours:
  - 135 hours (accurate counting because of always stopping time for lectures with a tool)

4.3.2 Own topics/services

- stack_goal_service
  - Type: reasoning_planning/StackGoalService
  - Function: Service for goal to stack to blocks on each other. Inputs are semantics to define the blocks (like red, left, ...) and output is success if the stacking of the blocks has been started and if not, multiple error codes.
- dispatch_plan_feedback
  - Type: reasoning_planning/DispatchPlanFeedback
  - Function: Provide feedback message after dispatching of plan has started. Gives back success when goal was successfully reached or appropriate error codes if not.
- action_feedback
  - Type: reasoning_planning/DispatchPlanFeedback
  - Function: Provide feedback message from simple action executions to reasoning & planning service when an action was not successful with an appropriate error code

4.3.3 Description

The description of the module is showed by a simple usecase of stacking two blocks on each other.

Suppose there is a scene with 4 different colored blocks (red, green, yellow, blue) where the green block is stacked on the yellow block and the blue block is stacked on the red block. The user wants to stack the red block on the yellow block.

First, the speech module sends a request to stack_goal_service with the correct semantics set (top_block_semantics = [RED]; below_block_semantics = [YELLOW]). Then, the service fetches the actual scene from the perception via the service get_scene and stores it in the scene database. Next, the service has to connect the high-level semantics of the speech module with the lower-level representations of the blocks in the scene database and infer actually which blocks are wanted to be stacked. In this case, this is rather simple because all blocks have different colors. However, it could also be that there are multiple red blocks. Then the service returns that there are multiple possible

blocks at the top position. The user can rephrase its request with multiple semantics like top_block_semantics = [RED, LEFT_RELATIVE]. This means it should take the red block which is left from the other block (in this case the other block is yellow). The spatial relations like left, right, far, close, higher, lower are always as seen from the user and not from the baxter. For the spatial relations there is always relative or absolute possible. Relative means relative to the other block (of course the other block must be unique) and absolute means absolutely seen for the whole scene of blocks). We continue with the original scene and request which means we can infer the correct blocks which should be stacked.

Next, there is based on the scene the high-level knowledge created to plan the scene. ROSPlan uses as language for domain definition `PDDL`. The planning domain is defined statically by the programmer. There are two types, a block and a location, and both have the same base class place. It also contains so called predicates which work like true or false statements, and the simple actions which work on the defined types and blocks when conditions of predicates are fulfilled and then sets the predicates as defined. Therefore, the high-level knowledge of the existing blocks and locations, and fulfilled predicates at the stored scene is created. The complicated part is that (as always in practice) there are measurement uncertainties at the blocks from the perception so it can be challenging to infer if two blocks are really stacked on each other or if they are on separate locations on the table or not or if the top face of the block is clear to stack another block on it or not. A high accuracy of the perception is therefore needed. Additionally to the block locations there is also a free location stored in the knowledge and scene database. This free location is randomly in a defined area on a table where there are no other blocks. It is used as a temporary location to put other blocks. Otherwise it could be that there is no plan found because the wanted blocks are below other blocks and those blocks on the top cannot be put somewhere else because there is no free location known.

After the high-level knowledge of the blocks, locations and predicates are set in the knowledge database (this is different than the scene database!), the goal definition is set. In this case it is that the predicate "on block_red block_yellow" shall be fulfilled. This predicate is set as goal and then the service of ROSPlan to create a plan is called. Because we defined a temporary location, as showed before, a plan should always possible to find. The plan is parsed and after parsing a separate thread is created to actually dispatch the plan. After the thread is created, the service stack_goal_service returns with success=True.

Now follows the actual plan execution. There are 4 different actions defined in the planning domain: `pick up`, `put down`, `stack` and `unstack`. All of those actions have a separate class inferred from a base class of ROSPlan which is called when ROSPlan want to execute them. All classes share that first they need to fetch the actual block and location representation from the scene database with the name of the high-level knowledge. `Pick up` and `unstack` are similar. Both pick up a block but `pick up` picks the block from a location on the table where `unstack` picks up the block from

another block. On both actions based on the name the block is picked up with the service kmr19_manipulation_pick_up of the manipulation module by using its name in the scene database. `Put down` and `stack` are also similar. `Put down` puts the holding block down on a location on the table whereas `stack` stacks the holding block on another block. The wanted pose to set the holding block is inferred from the location or below-block pose and is used at the service kmr19_manipulation_put_down. Additionally at `put down` and `stack`, if the action is executed with no error from the manipulation module, there is the scene database updated with the new position. After it, the scene database is checked against the scene as seen from the perception. If the two scenes are different, this means the robot has collided any block, or putting down or stacking was not successful. The challenging parts here are again measurement uncertainties and inaccuracies from the perception and this time also manipulation inaccuracies. If there are no errors returned from every action the plan was successfully dispatched and the topic dispatch_plan_feedback publishes that the dispatching was successful. If there is an error, then the action publishes on the topic action_feedback and returns false to ROSPlan. ROSPlan stops the plan execution and it returns back to the separate thread started from the reasoning & planning module (see before). This separate thread subscribes to the topic action_feedback and uses this information to set the appropriate error code at the topic dispatch_plan_feedback which can be later used at the speech module to provide feedback to the user.

4.3.4 Faced problems and found solutions

At beginning of work the main problem was to decide on a basic framework or structure for reasoning & planning. The first idea was to use TMKit [15] which provides a full framework for not only task but also even already the motion planning. However, first tests (only at simulation) showed that the framework is not mature and does not perform as expected. Even at the basic tutorial in the simulation there were collisions between the objects and arms visible. Additionally, the installation was very error prone. Therefore, we decided to use ROSPlan [12] which is mature and well documented.

The next faced problem was that the reasoning & planning highly depends on all other modules but at earlier points of development the other modules were (as planned) not ready for productive usage. Therefore many nodes, scripts and other stuff had to be created for testing that module. Just starting at a later point with the reasoning & planning would not have been doable because the overall time is too short to wait until the other modules are finished.

At a later point where we wanted to provide error feedback from the single actions back to the reasoning & planning service node, we faced the issue that ROSPlan's standard way of implementing actions is not built such that there are error codes/messages possible but only return true or false. It is possible to create a custom action interface but there is less documentation about it and at that late point, it was also no more feasible. Therefore, we decided to send the error code via a separate topic called `action_feedback` (see Figure 1 and Section 4.3.2).

Another huge faced problem which cost much time was that ROSPlan provided a full framework for planning but not for reasoning of the scene and interpreting the semantic representations of speech commands.
First, the blocks returned by the perception are just having some poses without any relation to each other. From that it must be inferred if they are e.g. actually standing on the table or stacked on another. Those predicates form the basics for the planning. This was especially hard because (as always in real world) there are measurement uncertainties and even two consecutive requests to the perception can lead to different returned blocks. Second, the command from the speech just provided semantics of the blocks like "red", "blue", "absolute left", "relative right", "relative far" and so on. Based on that the actual meant block must be found. For humans this is rather easy but for robots not because it is not clear and precise what actually mean those semantics. In this case it was solved such that for every single semantic a filter was created which sorts out not matching blocks. First there were the color and absolute spatial relations filtered out. After that the relative spatial relations were used because they work on the spatial relations relative to the other block. Of course this is only possible if there is exact one other block left after filtering the absolute spatial and color semantics. After that the planning was simple as the IDs used by the predicates to describe the goal were clear and ROSPlan does correctly the rest of the planning as long as the high-level representation of the blocks and their relations were correct.

4.3.5 Known issues and further improvements

One known issue is that reasoning & planning highly depends on the ID of the blocks. If there are blocks with the same ID, the planning or action execution will fail. The ID was meant to be unique and even when blocks are moved they shall have the same ID as before. Therefore, object tracking would have been necessary. Due to lack of time, this was not implemented in the perception and the ID is just a combination of the color code and the block type code (big cube, small cube, etc.). Therefore, it is not allowed to have multiple blocks of the very same shape and color in the scene. Still it can make a problem if a block is detected as a different shape or color before and after manipulation. This will cause that the check for errors after manipulation will (correctly) detect that there are different blocks in the scene database and given by the perception. This means that the goal execution then aborts.

Another known issue is that the planning domain is only defined for one arm. It can be extended for a second arm but this was not possible in that short time. This reduces the possible workspace of the robot to a smaller area than else possible with two arms.

4.4 Manipulation

4.4.1 Key-Facts

- Responsibility:

  - Gerald Strommer

- Task:
  - Control the robot to manipulate the given scene using the robotic arms and the grippers
  - Motion planning to pick up specific blocks and put them down on a given position.
  - Report different error cases to give user feedback.

- Needed working hours:
  - approx. 100 hours

4.4.2 Topics and Services

The manipulation part of the project is controllable via two ROS-services:

- kmr19_manipulation_pick_up
- kmr19_manipulation_put_down

Both of them are hosted by the ROS-node *kmr19_manipulation_node.py* with the name *kmr19_manipulation_server*.

Pick-Up Service:

The service request consists of an identification number corresponding to a block in the scene that should be picked up. The server node sends a response containing a boolean variable for success and a specific error code.

Put-down Service:

The service request consists of a pose that should be applied to the previously picked up block. The server node sends a response containing a boolean variable for success and a specific error code.

4.4.3 Software Architecture

The whole manipulation part is implemented in one single ROS-node. Three Python classes provide all the needed functionalities:

- **Robot Control:** Enables the Baxter-Robot and set the robots head position (where the camera for perception is fixed).
- **Arm Control:** Planning and executing arm and gripper movements.
- **Scene Control:** Managing possible collision objects in the scene.

The main python function generates global objects from these types and calls their functions to fulfill the service requests for pick-up and put-down.

4.4.4 Program Description

The MoveIt Planning Framework is used for this manipulation task. Therefore, the Python *MoveGroupCommander*, imported from the *moveit_commander* library is used for planning and executing arm movements. For creating and manipulating the planning scene the *PlanningSceneInterface* is used.
When starting this ROS-node, the robot components get initialized. Fixed objects like the table and the computer screen are added to the planning scene to avoid collisions. The arms move in a position that the camera can see the whole scene.

Pick-Up Procedure

At the beginning of each pick-up request, the planning scene reinitializes. Afterward, all blocks detected by the perception are loaded from the scene database to add them to the planning scene. At the beginning of the pick-up procedure, the arm moves above the block that should be picked up. This is the so-called Pre-Grasp-Pose. The grippers are always perpendicular to the table. The following Grasp-Pose is reached by performing a Cartesian Movement (straight line movement) along the z-axis of the global frame. When the gripper has closed, the same movement is applied to reach the Pre-Grasp-Pose again. The block gets attached to the robot's arm and when everything worked correctly, the service sends their response.

Put-down Procedure

The put-down procedure has the same three steps as the pick-up procedure: Pre-Release-Pose, Release-Pose, and Post-Release-Pose. The block's goal position is given by the service request. If everything worked correctly, the response is sent and the arm moves back to the initial position.

4.4.5 Implementation Problems

The first problem we faced was introduced by the grippers. There is a feedback function if an object is gripped or not, but the return value is always negative. We tried to use the position of the grippers to measure a successful grasp, but this measurement was too inaccurate. Therefore, an unsuccessful grasp is not detectable. Errors made by manipulation are checked after each put-down procedure by using perception data.

The straight down and up movement introduced some major problems. As mentioned before, Cartesian Path Planning is used in the final implementation. Because of inaccurate translations of the arm, we needed to reduce the speed of movements. This feature is not supported in ROS-Kinetic when using Cartesian Path Planning[1]. Therefore, we tried to use orientation constraints instead to force the end-effector link to straight movements. Consequently, the planner had to deal with more difficulties. This lead to instabilities and strange arm movements before the actual plan has been executed. Therefore, we moved back to Cartesian Path Planning and reduced the speed for

[1] `https://github.com/ros-planning/moveit/issues/1556`

all arm movements by changing parameters in the *joint_limits.yaml* file (Baxter MoveIt Implementation Folder).

When MoveIt Path Planning has problems to find a valid path, the planning strategy can be changed. We stuck with the default planning strategy (RRT), but we changed the planner from *RRTkConfigDefault* to *RRTConnectkConfigDefault* to get better results (done in *ompl_planning.yaml*). Furthermore, the inverse kinematic solver can be changed to achieve better results. Therefore, we downloaded the **trac_ik**[2] kinematic solver to improve plan execution. We changed the used solver in the *kinematics.yaml* file (Baxter MoveIt Implementation Folder).

A valid path found by MoveIt gets optimized when there is time left. Due to this post-processing the path may get invalid again because of collisions with objects. To avoid this problem, we changed the *longest_valid_segment_fraction* parameter from 0.05 to 0.01. This parameter limits the maximal distance between two nodes in a RRT-based planning strategy. This measure reduces the probability that the problem arises.

4.4.6 Possible Improvements

The whole manipulation can be improved by:

- More precise robotic arms (depends on Robot)
- Gripper feedback (depends on Robot)
- The possibility to control the speed during Cartesian Path execution (depends on MoveIt)
- Grasping the blocks from different directions (increases range of robotic arms)
- Use both arms for more complex manipulation tasks and range increase.

References

[1] Michael Cashmore et al. "ROSPlan: Planning in the Robot Operating System". In: *ICAPS*. 2015 (cit. on p. 10).

[2] *Extrinsci Calibration explanation*. Jan. 2020. URL: `https://www.youtube.com/watch?v=lL3HjvxNNoc` (cit. on p. 2).

[3] *Google speech API installation*. Jan. 2020. URL: `https://cloud.google.com/python/setup` (cit. on p. 2).

[4] *Intrinsic calibration of a depth sensor*. Jan. 2020. URL: `https://wiki.ros.org/openni_launch/Tutorials/IntrinsicCalibration` (cit. on p. 2).

[5] C. Kerl, J. Sturm, and D. Cremers. "Dense Visual SLAM for RGB-D Cameras". In: *Proc. of the Int. Conf. on Intelligent Robot Systems (IROS)*. 2013 (cit. on p. 5).

[2] `https://bitbucket.org/traclabs/trac_ik/src/master/`

[6] *MongoDB*. Jan. 2020. URL: https://www.mongodb.com (cit. on p. 10).

[7] *mongodb_store*. Jan. 2020. URL: http://wiki.ros.org/mongodb_store (cit. on p. 10).

[8] *Point Cloud Library documentation*. Jan. 2020. URL: http://pointclouds.org/ (cit. on p. 5).

[9] Alex Reimann. *Depth calibration of an ASUS Xtion Live or Kinect*. Oct. 2017. URL: https://github.com/AlexReimann/depth_calibration (cit. on p. 5).

[10] ROS. Jan. 2020. URL: https://wiki.ros.org/openni_launch/Tutorials/IntrinsicCalibration (cit. on p. 7).

[11] *ROSPlan*. Jan. 2020. URL: https://github.com/KCL-Planning/ROSPlan (cit. on p. 3).

[12] *ROSPlan - Website*. Jan. 2020. URL: https://kcl-planning.github.io/ROSPlan/ (cit. on pp. 10, 13).

[13] *Stanford parser download*. Jan. 2020. URL: https://stanfordnlp.github.io/CoreNLP/download.html (cit. on p. 2).

[14] Alex Teichman, Stephen Miller, and Sebastian Thrun. "Unsupervised Intrinsic Calibration of Depth Sensors via SLAM". In: *Proceedings of Robotics: Science and Systems*. Berlin, Germany, June 2013 (cit. on p. 5).

[15] *TMKit*. Jan. 2020. URL: http://tmkit.kavrakilab.org (cit. on p. 13).