TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering

ITI70LT
Oliver Kalmend 132680 IASM

ROBOT PLATFORM FOR NATURAL LANGUAGE DIALOGUE SYSTEMS

Master's thesis

Supervisor: Jüri Vain
Professor

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutitehnika instituut

ITI70LT
Oliver Kalmend 132680 IASM

ROBOTPLATVORM LOOMULIKU KEELE DIALOOGSÜSTEEMIDELE

magistritöö

Juhendaja: Jüri Vain
professor

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Oliver Kalmend

09.01.2017

Acknowledgement

I would like to thank my supervisor Prof. Jüri Vain for his assistance, guidance and the work he did with the dialogue system used in this Master's thesis. I also want to acknowledge Dr. Tanel Alumäe for his help and continued work with Estonian speech recognition. And finally, I would like to thank Dr. Jan Wielemaker, the author of SWI-Prolog, for helping overcome some of the difficulties encountered with the SWI-Prolog API.

Abstract

The goal of the given Master's thesis is to design and develop a robot platform that is capable of limited natural language dialogue. The final prototype system must be able to understand, execute and respond to predetermined commands, which are given in natural speech.

This thesis deals with the architecture, design and implementation of the described prototype platform. The final solution is a composition of subsystems working together. Each subsystem maintains some critical functionality for a natural speech capable robot. The thesis describes the functionality of the subsystems, their design, implementation details and communication flows.

This thesis is written in English and is 61 pages long, including 5 chapters, 68 figures and 11 tables.

Annotatsioon

Tänapäeval on loomulikku keelt võimaldavad kasutajaliidesed muutunud väga populaarseks ja kiiresti arenevaks valdkonnaks. See on modernsetes nutiseadmetes oodatud funktsionaalsus, mis lihtsustab klientide jaoks keeruliste süsteemide kasutust. Keelepõhine kasutajaliides on inimese jaoks mugav ja loomulik viis arvutisüsteemidega suhtlemiseks. Väga levinud on tarkvaralised assistendid nagu Microsofti Cortana, Applei Siri ja Amazoni Alexa. Ka riistvaraliste teendindusrobotite hulk on kasvamas. Leidub robootilisi assistente, giide, kullereid ja isegi medõe ülesandeid täitvaid roboteid. Vabalt on saadaval tarkvara, mis implementeerivad mingi osa sellisest robotist, kuid puudub vabavaraline ühtne platvorm, mis võimaldab kõnejuhtimist kasutavat robotit luua.

Käesoleva magistritöö eesmärk on kavandada ja välja töötada robotplatvorm, mis võimaldab hõlpsasti luua piiratud dialoogiga roboti rakendusi. Prototüüprobot peab loomulikus keeles esitatud eeldetermineeritud käskutest aru saama ja neid täitma. Dialoogsüsteemi prototüüp on loodud simulatsioonroboti baasil ning kasutab eestikeelset dialoogsüsteemi, kõnesünteesi ja -tuvastust, et loomulikus keeles antud käskudele reageerida. Robot peab vastavalt käskudele täitma simuleeritud maailmas lihtsaid objektide manipuleerimis- ja navigeerimisülesandeid. Eesti keel on valitud selle tõttu, et eesti keele baasil ei ole sellises mahus kõnejuhtimisega robotprojekte varem tehtud.

Magistritöös on kirjeldatud platvormi arhitektuur, disain ja teostus. Esitatud lahendus on modulaarne platvorm, mis põhineb ROSi (Robot Operating System) arhitektuuril. Loodud platvorm on alamsüsteemide kogum, mis koostöös täidavad püstitatud ülesandeid. ROSi modulaarsus võimaldab lahenduse komponentide lihtsat täiendamist ja asendamist, andes võimaluse uute rakenduste genereerimiseks. Magistritöös on esitatud informatsioon kõikide välja töötatud alamsüsteemide kohta, mis aitab loodud platvormi mõista, kasutada, muuta ja laiendada. ROSi kasutamine tagab selle, et disain on paindlik ja kõik komponendid on kergesti taaskasutatavad. Kõnejuhtimise tagamiseks on kasutusel sügavatel närvivõrkudel põhinev kõnetuvastus, mille tekstiline väljund suunatakse dialoogsüsteemi. Dialoogsüsteem kasutab DCG (Definite Clause Grammar) parsimist, et genereerida tegevused, mida robot peab täitma. Dialoogsüsteem hindab kontektsipõhiselt kõnetuvastuse väljundis oleva lause sobivust ja aitab mitmeti tõlgendatavuse korral valida kandidaat-interpretatsioonidest parima. Roboti reaktsioone käskudele simuleeritakse roboti tegevustena virtuaalses keskkonnas ning genereeritud vastustega, millest sünteesitakse kõne.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 61 leheküljel, 5 peatükki, 68 joonist, 11 tabelit.

List of abbreviations and terms

GUI	Graphical User Interface
ROS	Robot Operating System
GST	GStreamer
API	Application Programming Interface

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

In today's world, natural language interfaces have become commercially very successful. It is an expected feature in many modern appliances and smart technologies. Due to an overall increase in the complexity of systems natural language interfaces are becoming an essential tool to guarantee usability. The growing need for non-trivial and semantically rich interaction between humans and the contemporary cyber world has led to the rise of natural language user interfaces and intelligent personal assistants like Microsoft's Cortana, Apple's Siri and Amazon's Alexa. In addition to these software level assistants, we have seen an increase in robotic companions such as robot assistants for elderly people and kids, robot guides in public buildings and even autonomous delivery robots.

Figure 1. RIBA the bear-shaped nursing robot [19]

There is a smattering of disparate freeware pieces of software available that implement some part of such a robot. However, there is a distinct lack of an easily extendable, modifiable and reusable freeware platform that is both a complete example and can serve as a starting point for creating robots that are interfaced via natural language. The primary aim of this thesis is to provide a prototype of such a platform.

1.2 Objectives

The main goal of this thesis is to design and implement a prototype robot platform and dialogue system that allows for natural language communication. We aim to present a platform that sufficiently demonstrates the use of speech recognition and synthesis to allow communication with a robot through natural human speech. The robot should be able to listen to commands within a set of predefined phrases, understand them and execute some reactive behaviour. As proof of concept functionality, we have chosen simple indoors navigation tasks and simulated object manipulation tasks for the prototype's initial capabilities. Our robot prototype will be simulated in a virtual world model. An example of a physical robot that could perform such tasks is depicted in Figure 2.

Figure 2. Pioneer research platform with manipulators [17]

We have chosen Estonian as the initial language for the robot. This is because of the current lack of robot workers, assistants and guides in the Estonian market. We want to provide a platform that is a good starting point for any such Estonian speaking smart-technology. A simulated robot that is capable of completing simple tasks in the

simulated environment of Tallinn University of Technology IT-building was chosen for the demonstration goal. However, the system should be flexible enough to enable easy switching between languages. We also want to provide sufficient explanation of the final platform to make it easier to build, understand, utilize, modify and expand. These general goals can be broken down to a list of subtasks:

- Using common architecture to create cohesion between all the system components.
- Finding and integrating a flexible audio-capture, -playback and -manipulation system.
- Finding and selecting a speech recognition toolkit.
- Finding and selecting a speech synthesis toolkit.
- Finding or developing a light-weight demo dialogue system.
- Setting up a communication flow and synchronization.
- Integrating, modernizing, modifying and debugging all the components to fit into our system.
- Implementing missing functionality to get to a fully functional demo.
- Optimizing the whole architecture.
- Creating easy-to-use configuration and launch options.
- Automating the build process.
- Documenting the work.

1.3 Outline

Chapter 2 explains the behaviour of the final system with examples and illustrations. The key features and design decisions of the platform are elaborated here.

Chapter 3 gives a general overview of the architecture of the platform. This includes details of the components chosen and an overview of the component composition and communication of the running robot.

Chapter 4 provides details on all the nodes used and created as part of the solution. This includes a description of the functionality, diagrams, parameters and the ROS communication paradigms used with the node. Nodes that integrate GStreamer also have a description of the data pipeline.

Appendix 1 describes the code-base, where to find the source code and how to build the platform.

Appendix 2 lists future development ideas.

Appendix 3 gives an overview of other alternative launch configurations that were created for use with this platform and can be used to test various services.

2 Software behaviour and implementation

This chapter illustrates the most important workflows and design decisions of the natural speech controlled robot platform.

2.1 Robot reactions

The process of transforming spoken audio to robot actions can be illustrated with the following sequence diagram:

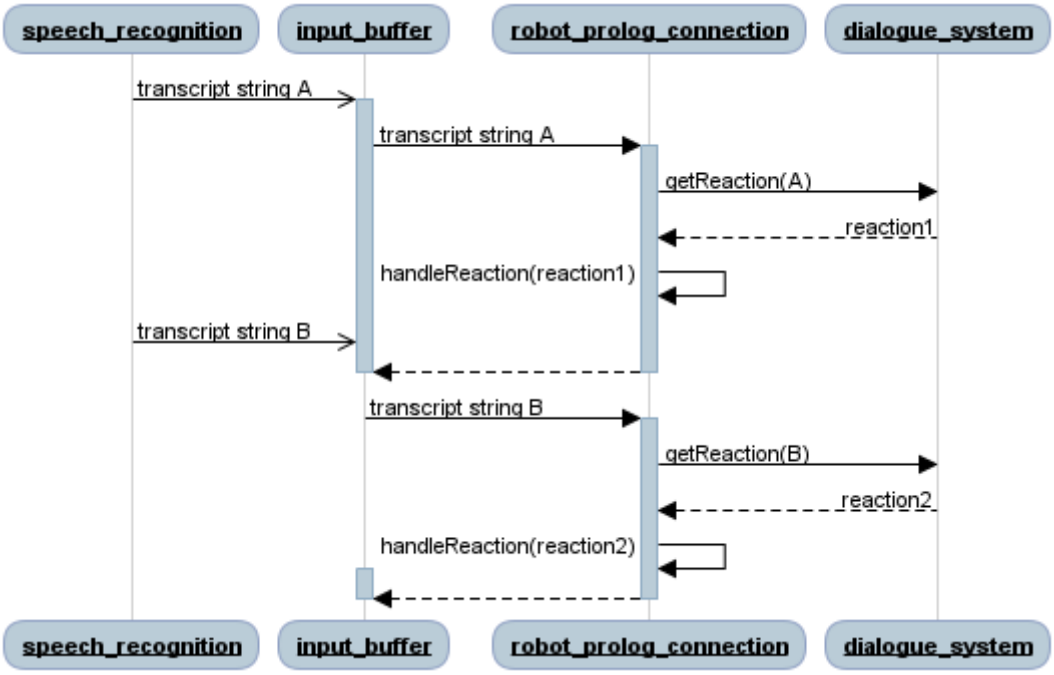

Figure 3. Generating reactions from speech transcript

The *"speech_recognition"* component is the only one to work asynchronously from the rest of the flow. It transforms voice commands to a textual form.

The *"robot_prolog_connection"* component processes one text transcript at a time, which means that additional text inputs arriving, while one is being processed, get stored in an input buffer and processed sequentially.

For each text input to *"robot_prolog_connection"*, a query against the dialogue system (see Chapter 3.3) is run, where the command is interpreted and a "*reaction*" behaviour of the robot is generated. A reaction is a set of operations that the robot has to perform, that make sense in response to what was spoken. For instance, if the robot is told to go to the cafeteria, the reaction task would involve the following steps:

1. responding that the command was understood (*respond* command)
2. navigating to the cafeteria (*goto* command)
3. responding that the desired location was reached (*respond* command)

The described reaction is **one of sixteen** currently supported by the dialogue system. Reactions are context sensitive – meaning they take into account various facts about the world and current situation. The commands recognized by robot's command language grammar are translated into one of these reactions. A full list of the reactions is a good descriptor of the platform's current capabilities.

Table 1. List of dialogue system reactions

reaction identifier	**description**	**commands used**	**synchronicity**
respond	Triggers default response, which is to say that the command was not understood	respond	S
stop	Triggers stop command, which stops current reaction handling and discards all queued up reaction tasks.	stop	S
juhata	Leads the target to desired location	respond, goto	AS
ütle_kus1	Responds to queries about the location of objects.	respond	S
ütle_kus2	Responds to queries about the location of people.	respond	S
ütle_kus3	Responds to queries that are in an alternative form.	respond	S
ole	Tells the robot where to stay.	respond, goto	AS
tule1	Robot goes to speaker.	respond, goto	AS

tule2	Robot comes to the place described	respond, goto	AS
otsi_leia	Robot looks for the mentioned object according to its internal world context.	respond, goto	AS
anna_ulata	Robot hands over an item.	respond, place	AS
võta_haara	Robot takes an item.	respond, pick	AS
tõsta_pane	Robot places an item at specified place.	respond, place	AS
too	Robot goes and gets an item	respond, goto, pick, place	AS
vii	Robot takes an item to a specified place	respond, goto, pick, place	AS
liigu_mine	Robot goes to location.	respond, goto	AS

After the query described in Figure 3, the *"robot_prolog_connection"* component checks if the reaction task can be completed synchronously (denoted by S in Table 1, column 4). Synchronous reactions are immediately run and completed - they have no long run-time associated with them. Asynchronous reactions (denoted by AS), on the other hand, get inserted into a priority queue and handled at a later time in order of their priorities.

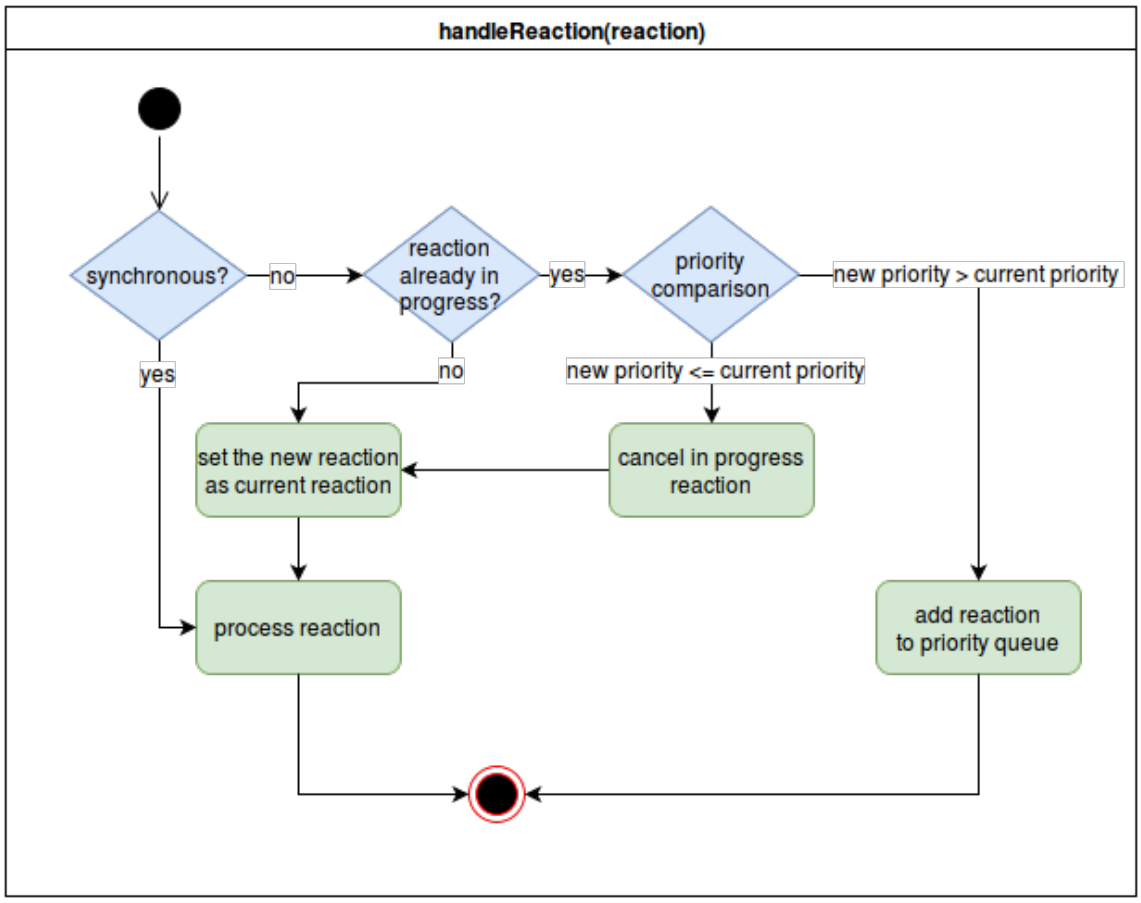

Figure 4. Processing reactions returned by the prolog dialogue system

The activity box *"process reaction"* is elaborated in the next chapter. Note that if there was another higher priority task in progress, the current reaction is queued up for later handling. It is only processed once all higher priority tasks have finished.

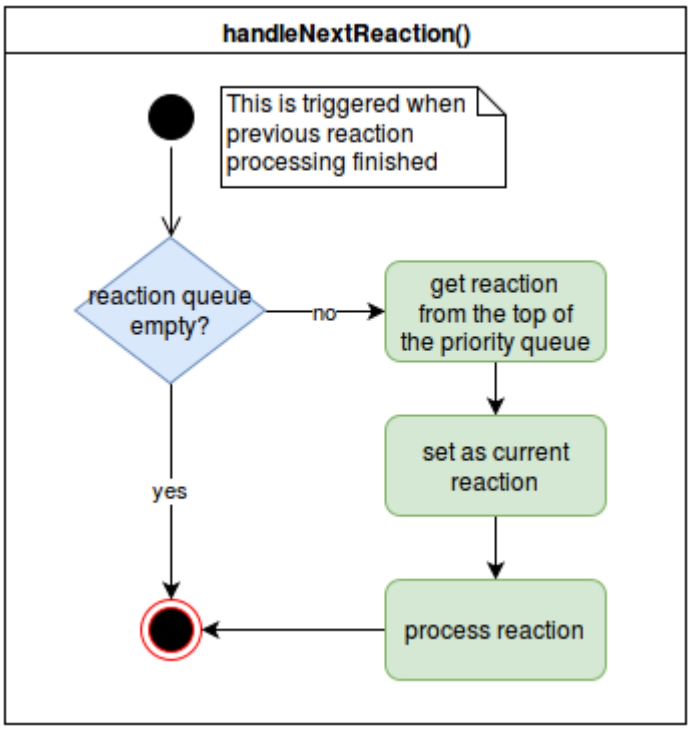

Figure 5. Processing the next reaction task in queue

Currently the robot prioritizes reactions as shown in Table 2:

Table 2. Reaction prioritization

priority	**reactions**
1	stop, respond, ütle_kus1, ütle_kus2, ütle_kus3
2	anna_ulata, võta_haara, tõsta_pane
3	juhata
4	liigu_mine, ole
5	all others

The priority for synchronous reactions does not matter, because they are always handled immediately after the dialogue system returns them, i.e. they can be viewed as having the highest priority.

2.2 Robot commands

We can see in Table 1 that all the reaction tasks are just permutations of 5 basic commands. Basic commands are operations directly executable by the robot. In order for the *"robot_prolog_connection"* component to get these commands, additional queries against the dialogue system are necessary. These queries result in sequences of basic commands and completing all of the commands means completing the reaction. Commands themselves are single operations that the robot is capable of performing.

Table 3. Robot executable commands

command	**description**	**synchronicity**
stop()	Immediately stops all commands and reactions. Empties the reaction queue.	S
respond(text)	Immediately sends the text to speech synthesis and returns. Speech synthesis has its own input queue and says the message as soon as possible.	S
goto(place,X,Y,Z,Q)	Goes to place with coordinates XYZQ. The place variable is used to display the name of the location in theGUI and for logging purposes.	AS
pick(object,X,Y,Z,Q)	Takes object from coordinates XYZQ. The object variable is used to display the name of the object in the GUI and for logging purposes.	AS
place(object,X,Y,Z,Q)	Places object to coordinates XYZQ. The object variable is used to display the name of the object in the GUI and for logging purposes.	AS

Synchronous reactions can only contain synchronous robot commands, asynchronous reactions can contain both types of commands. Picking up and placing objects is stub behaviour that can be completed via the GUI (Chapter 2.3).

The *"robot_prolog_connection"* node is aware if the reaction is synchronous or asynchronous.

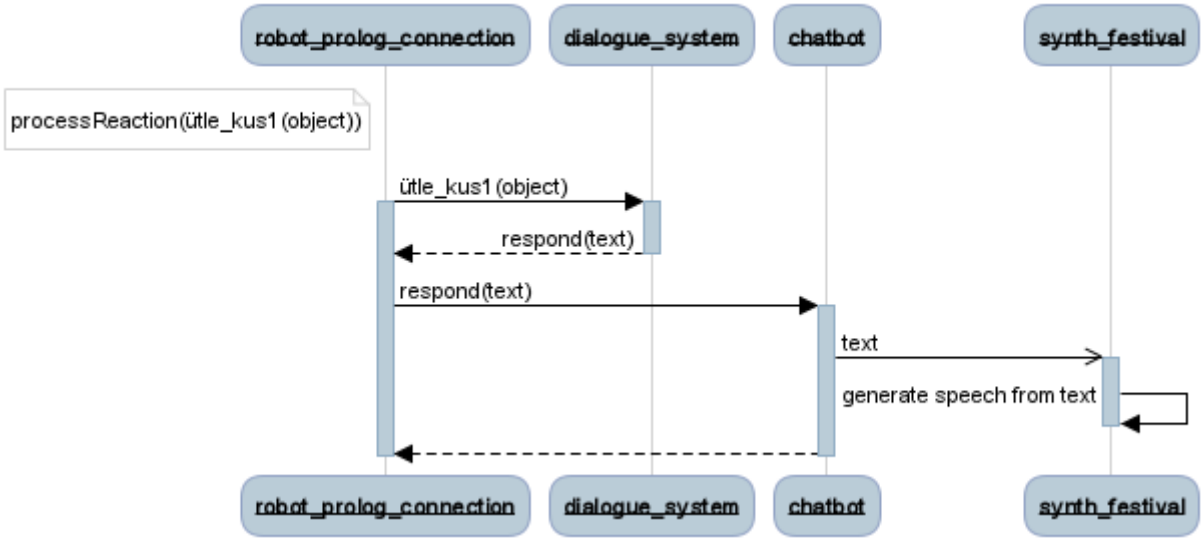

Figure 6. A synchronous reaction command-resolution example

The *"chatbot"* node on the diagram interfaces the textual response from the dialogue system with the speech synthesizer. In the chatbot node *"respond()"* is linked to speech synthesis, *"pick()"* and *"place()"* are linked to the GUI and *"goto()"* is linked to the navigation stack. The navigation stack, in turn, is providing inputs to the robot (in this work to a robot simulation in ROS). Because of ROS's plug-and-play nature, it would be simple to swap out the simulated robot with a real one.

As shown in Figure 6, in case of synchronous reaction tasks, the resulting commands are also synchronous and are handled immediately. If there was more than one command in the *"ütle_kus1"* reaction's execution scenario, then these would be handled sequentially.

For asynchronous reactions a thread is spun and some additional signalling is necessary because they involve long-running tasks. An example based on navigating to a new location:

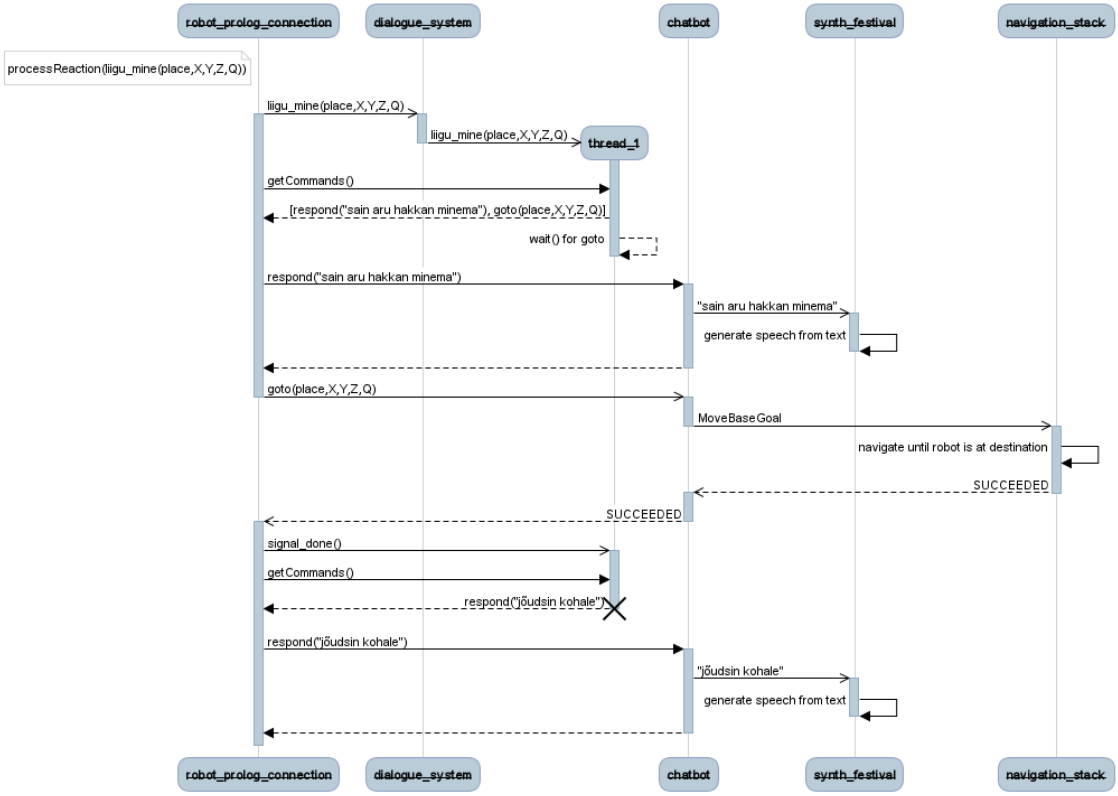

Figure 7. An asynchronous reaction command-resolution example

As Figure 7 shows, there is need for a *"signal_done"* signal from asynchronous commands. This signal is used to acknowledge the dialogue system that the command finished executing. As a result, the dialogue system stops waiting and produces the next command for execution. The *"signal_done"* signal is sent when the asynchronous command completes successfully. If the asynchronous command is either aborted or fails, the reaction thread is forcibly killed and the next reaction in queue is processed. One possible case where this failure can occur is if the robot is unable to navigate to the desired location.

2.3 The GUI

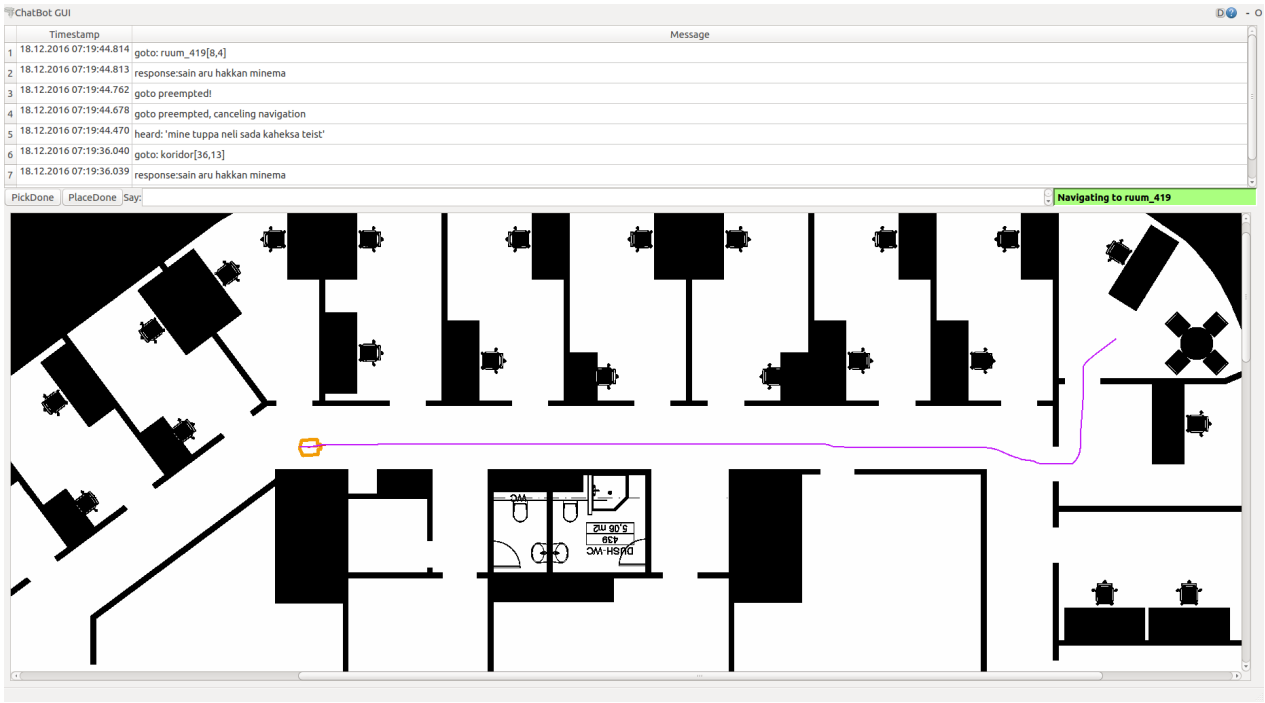

Figure 8. Graphical user interface

The graphical user interface was added to provide a clean overview of what the robot is seeing and doing. It features:

- A log window that can easily be extended with new log lines (Chapter 4.8).
- Buttons to complete stub *"pick()"* and *"place()"* commands.
- A text box for entering input to the dialogue system. This is a useful debugging tool if speech recognition is having problems or the user does not want to run speech recognition at all.
- A status bar displaying the robot's current activity.
- A map window that displays the robot's position and navigation plan.

For a more detailed view, it is possible to use the ***"visualize"*** argument on our launch file. This shows a more complex GUI implemented in rviz [27].

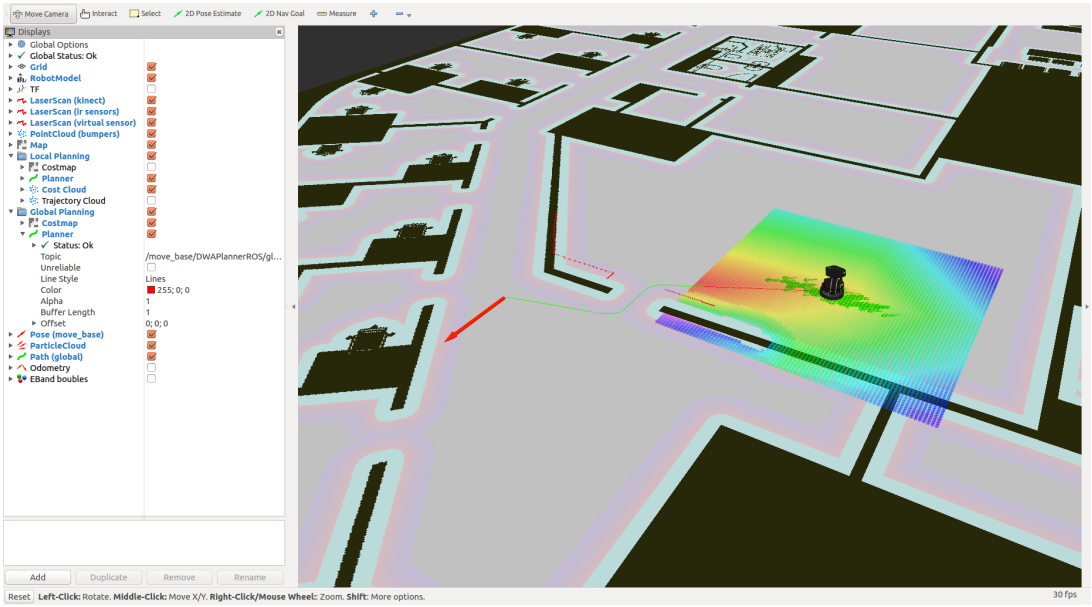

Figure 9. Detailed rviz based graphical user interface

The rviz GUI features:

- The ability to directly change the location of the robot on the graphical map.
- The ability to issue navigation goals without having to go through the natural language dialogue system.
- Additional debug information, e.g. the costmap, navigation goals, pose estimation particle cloud and others.

2.4 The simulation

In our solution the robot performs in a simulated world using the stage robot simulator [41]. The simulated world is idealized in the sense that it is directly generated based on the map provided. In real situations, the robot would also have to deal with a dynamic environment, which will add noise to the localization system. Changing the map and robot's dimensions is as easy as modifying a couple of configuration files (Chapter 4.9).

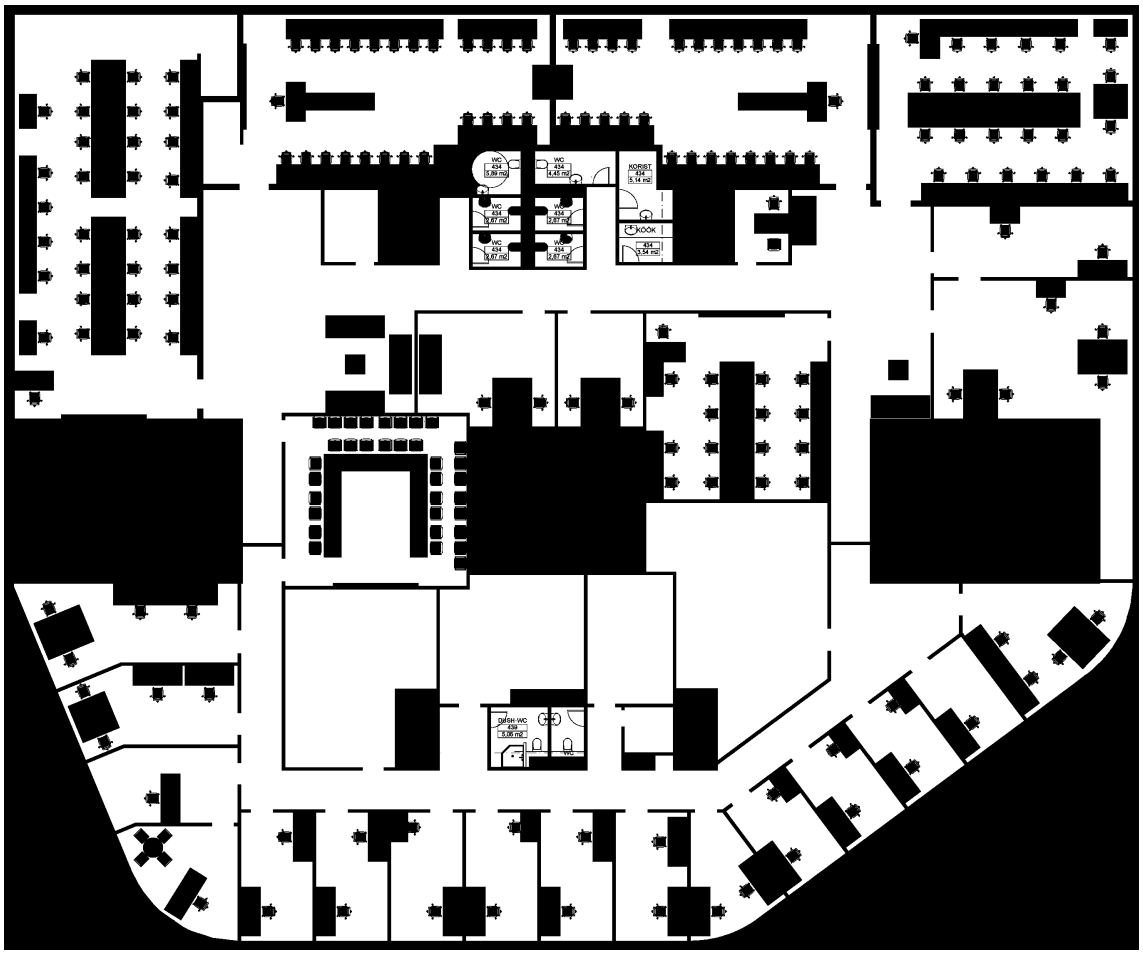

Figure 10. Tallinn University of Technology fourth floor map

The stage simulator is capable of creating a map based on any black and white image. Figure 10 was used as part of the default simulation environment of our natural language robot platform.

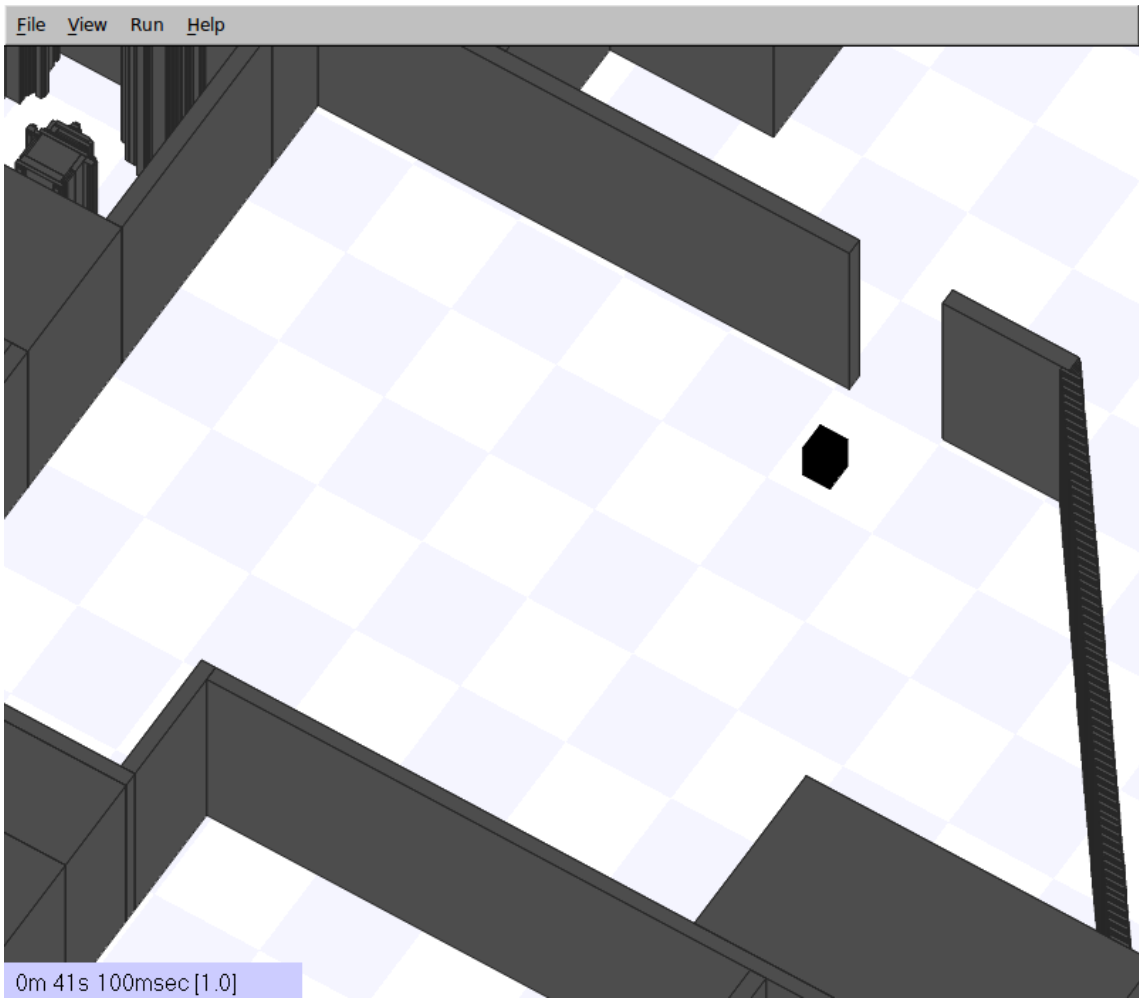

Figure 11. Stage simulator

Figure 11 shows the stage simulator's world. It offers another view of the robot, which gives us additional debugging and analysis opportunities. This view can also be enabled using the "*visualize*" parameter described in Chapter 2.5.

2.5 The main launch file

This section gives a quick overview of the parameters applicable to the main launch configuration. These are parameters for the unifying launch configuration that runs all of the nodes together forming one cohesive robot platform. Note that this is not the limit of the configurability - per-component parameters also exist and are described in Chapter 4.

The entire platform can be launched and arguments given in the following way:

```
~$ roslaunch chatbot chatbot.launch no_recognition:=true visualize:=true
no_voice:=false output:=screen
```

Figure 12. Starting the platform from the console

Table 4. Launch parameters for robot platform

parameter	**expected values**	**default value**	**effect**
config_file	path to prolog configuration file	“$(find prolog_server) /config/mt_server.yaml”	Allows creating multiple prolog configurations and selecting one
output	“log” or “screen”	“log”	“log” will output messages coming from nodes to log files. “Screen” will output messages to the standard output stream.
no_recognition	“true” or “false”	“false”	“true” means the voice recognition will not be started.
no_voice	“true” or “false”	“false”	“true” means the voice synthesis will not be started.
visualize	“true” or “false”	“false”	“true” means that additional simulator visualization tools will be displayed.
map_file	Path to .yaml formatted map file [24]	"$(find chatbot_simulator) /maps/4korrus.yaml"	Load file specified as the map for the navigation stack.
world_file	Path to .world formatted world file [3]	$(find chatbot_simulator) /maps/stage/4korrus.world	Load file specified as the world file for the stage simulator.
initial_pose_x	floating point coordinate within the world as a string	“31.0”	Set the robots initial x coordinate to this position. The default values match the map used.
initial_pose_y	floating point coordinate within the world as a string	“16.0”	Set the robots initial y coordinate to this position. The default values match the map used.

3 Software Architecture

3.1 ROS

As the base implementation platform for writing the natural language robot platform, the well known Robot Operating System (ROS) was chosen [23]. ROS presents a middleware that holds everything together and to which all of the components we use must conform. It is an open-source framework that provides a set of commonly used services and functionality in robotics development. ROS offers functionality ranging from low level hardware abstraction and device control, to high level features like communications architecture [21] and package management. It also offers various tools to develop, debug, inspect, build and write code. ROS was selected because of its role as de facto middleware standard in robotics, it is widely used, well documented and well supported – this is as close as we get to an industry standard when it comes to open-source robotics development platforms.

The basic idea of ROS is to provide a set of ready-made and community made components (or *"nodes"* in ROS lingo). Developers can easily use these together, in various configurations because of standardized interfaces. A big part of this thesis was creating these nodes using new systems and the systems described throughout Chapter 3. Audio capture, speech recognition, audio synthesis, a prolog engine and client and various other utilities were integrated into ROS compatible nodes in order to create a robot that understands and reacts to natural language.

ROS also provides a convenient XML-RPC based communication framework for signalling and messaging between nodes. The communication framework offers three common communication patterns: topics [29], services [28] and actions [20].

3.1.1 Topics

This communication pattern is meant for one-way periodic communication. A topic is essentially a named bus to which publishers push strongly typed data. The publisher knows nothing about subscribers, it only knows the topic name and the message type.

The same goes for the subscriber - thus we have a public interface through which loose coupling is achieved.

3.1.2 Services

In contrast to the topic pattern's many-to-many one-way transport, this pattern offers one-to-one two-way communication. Topics use one message, while the services pattern uses two for each service - a request and a response. The services pattern mimics remote procedure calls by returning data to the caller of the service through the response message. The messages are again strongly typed and must be predefined.

3.1.3 Actions

Actions are another communication pattern built on messages. Actions are meant for executing long-running tasks like navigating to a new location or picking something up. It differs from the services pattern in that it is non-blocking and preemptable. A service call blocks until a value is returned, but an action request immediately returns. While the action is being executed on the action server, the client may choose to cancel the action.

In short, each node in our architecture may have have services, topics or actions associated with them to provide means for communication. This information is presented with more detail per node in Chapter 4. All the topics, services and actions can also be directly called from the command line, which makes testing and running the nodes separately feasible. For example, it is possible to run only the speech synthesis node and give it an input message from the command line interface for a quick synthesis test.

3.2 GStreamer

Gstreamer (GST) [11] is a multimedia framework allowing data stream manipulation through pipelines. A pipeline in GStreamer is a chain of elements that manipulate data coming into the pipeline to achieve a specific task. For example, one could setup a GST pipeline that samples audio, applies filters and encodes it to a mp3 file. Very complex

pipelines are easy to create and manipulate thanks to the plug-and-play nature of GStreamer. GStreamer is widely used in audio-video applications.

Many nodes in our architecture use GStreamer to capture, convert, resample, filter and play-back audio. However, for audio transportation across ROS nodes, the audio stream is handed off to ROS. The output of a GST pipeline running within a ROS node is converted to ROS messages and transported using the *topics* pattern.

GStreamer is also used for communication with the speech recognition toolkit through a tool called gst-kaldi-nnet2-online (Chapter 3.5).

3.3 The Prolog dialogue system

Prolog is a general-purpose declarative programming language. It is often used in the field of artificial intelligence and natural language analysis and processing [18] - making it a great fit for the thesis' goals.

A SWI-Prolog [35] program was used to implement the actual dialogue system. This program was written by the supervisor of this thesis Prof. Jüri Vain. The dialogue system program takes speech command transcription text as an input and outputs a string describing an appropriate reaction. Further querying of the dialogue system with the reaction string results in a list of commands that the robot has to execute. The dialogue system also implements additional speech recognition error-correction, meaning that by using speech context information and context related vocabulary it can approximate what the speaker was trying to say up to a preset threshold of likelihood.

The dialogue system parses input transcriptions based on Definite Clause Grammar (DCG) [34]. As a result of the DCG parsing, a prolog fact *"goal"* is created that contains all the information required to execute a task (reaction name, target location, action subject, object etc). The information in the *"goal"* fact is used to drive execution of the commands contained in each *"reaction"*. Reactions determine a list of commands (an execution scenario) and have the ability to wait for the commands to finish executing using the prolog *"wait"* predicate. The reactions use information about the environment returned from *"world"* prolog facts. These facts are updated when executing commands to keep the robot's knowledge base in sync with the physical

world. For instance, if the robot transports a cup to another room, it must update the location of the cup in the *"world"* facts.

3.4 Kaldi speech recognition toolkit

This component does all of the speech recognition work. It is a powerful toolkit that is capable of both Gaussian mixture model (GMM) and deep neural network (DNN) based recognition [14]. The robot platform uses the DNN based model because it has been shown to outperform GMM based models [1]. The Estonian speech corpus necessary for recognition was trained and provided by the National Programme for Estonian Language Technology [37].

The traditional speech recognition pipeline consisting of a feature extractor, an acoustic model, a pronunciation model, a language model and a decoder has been largely replaced by DNN based solutions, where the network is trained to directly compute target hidden Markov model (HMM) states based on the input features. Kaldi uses this approach due to its increased accuracy, even though DNNs are difficult to train and computationally resource intensive.

3.5 gst-kaldi-nnet2-online

The *gst-kaldi-nnet2-online* package wraps Kaldi's speech recognition interface into a GStreamer plugin [2]. As part of the thesis work, the GStreamer plugin was in turn wrapped into a ROS node. The *gst-kaldi-nnet2-online* package allows for quick and easy communication with the Kaldi components that perform the speech recognition and transcription.

The GStreamer plugin built by the package is called *"kaldinnet2onlinedecoder"*. In order to use this plugin an environment variable *"GST_PLUGIN_PATH"* must be set to the directory of the plugin binary (*"thesis_platform/deps"* in our default configuration). After that, a list of plugin parameters, their default values and descriptions can be examined by launching *"gst-inspect-1.0 kaldinnet2onlinedecoder"*. This also reveals the following information about the GStreamer plugin's input and output:

```
Pad Templates:
  SINK template: 'sink'
    Availability: Always
    Capabilities:
      audio/x-raw
                  format: S16LE
                channels: 1
                    rate: [ 1, 2147483647 ]

  SRC template: 'src'
    Availability: Always
    Capabilities:
      text/x-raw
                  format: { utf8 }
```

Figure 13. kaldinnet2onlinedecoder interface

Figure 13 shows that the GStreamer element accepts single channel signed 16-bit little-endian audio and outputs utf8 encoded text. This means that no matter what encoding the audio is in (Speex, MP3), we must first convert it to this format to get speech transcription.

3.6 Festival speech synthesis toolkit

Festival is a speech synthesis toolkit developed by the University of Edinburgh's Centre for Speech Technology [36]. The C++ API was used to wrap it into a ROS node (Chapter 4.7).

3.7 The platform in abstract

All the ROS nodes working together form a network, which simulates a robot capable of understanding and responding in natural language. The following is an abstract depiction of the system:

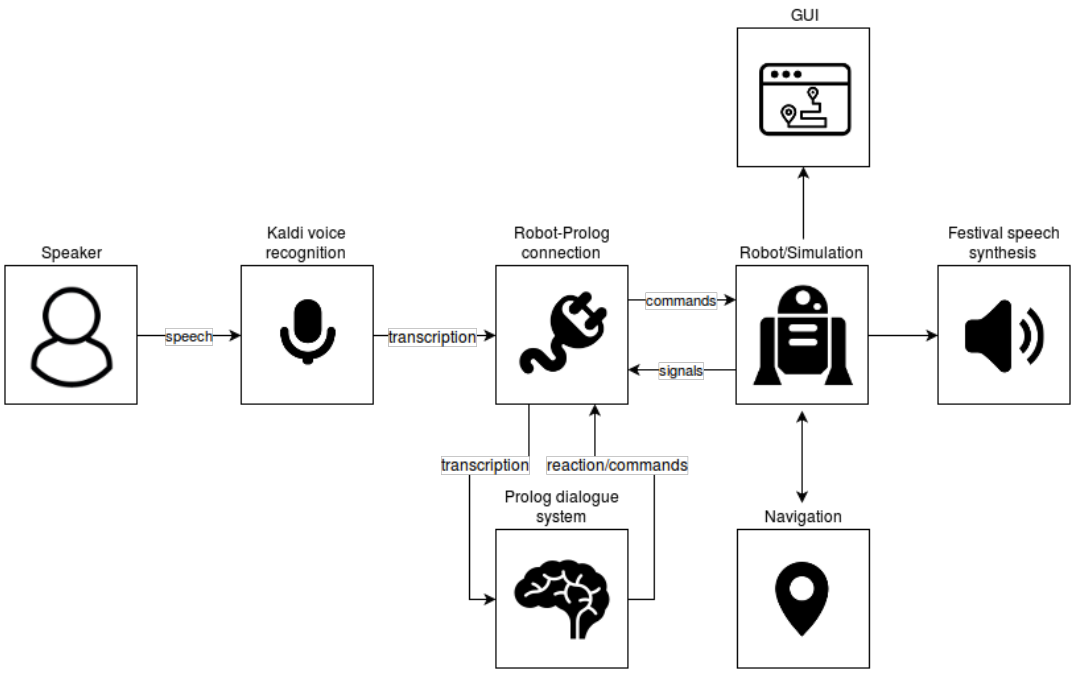

Figure 14. An abstract view of the robot's speech control architecture

3.8 Full robot configuration

This section presents the main launch configuration created as part of the thesis. The configuration of the nodes results in a cohesive system that simulates a robot capable of natural language communication. An overview of how the nodes communicate and work together is given. Further detail on individual nodes, their development process and functionality is presented in Chapter 4.

3.8.1 Nodes and topics

Figure 15 and Figure 16 present a full picture of all the nodes running in the main configuration and their communication. These two images represent the same network with different components abstracted due to space limitations.

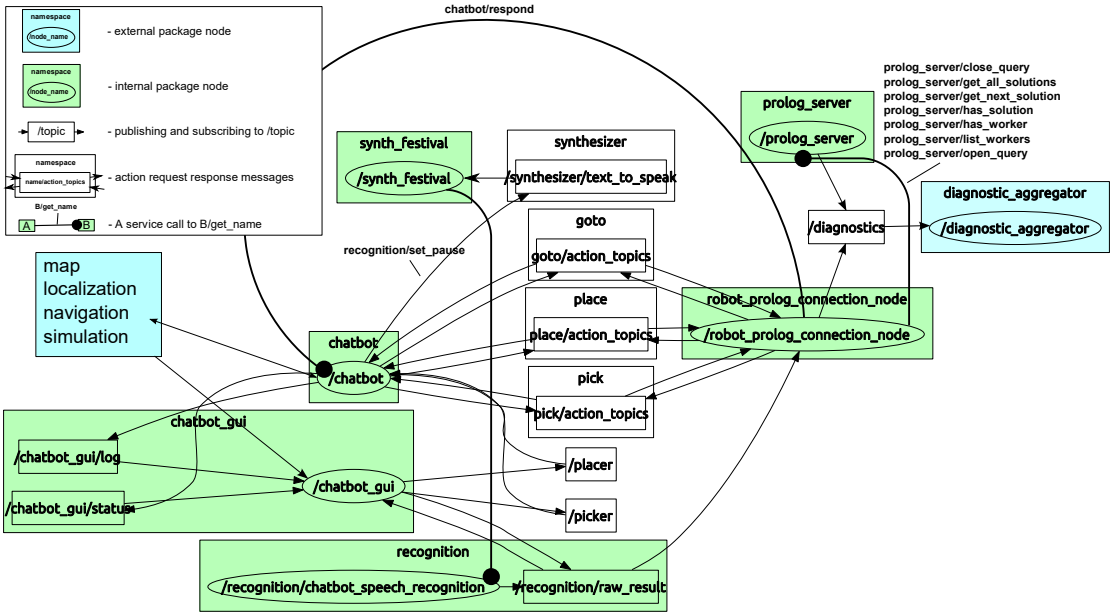

Figure 15. A detailed view of the final architecture (1/2)

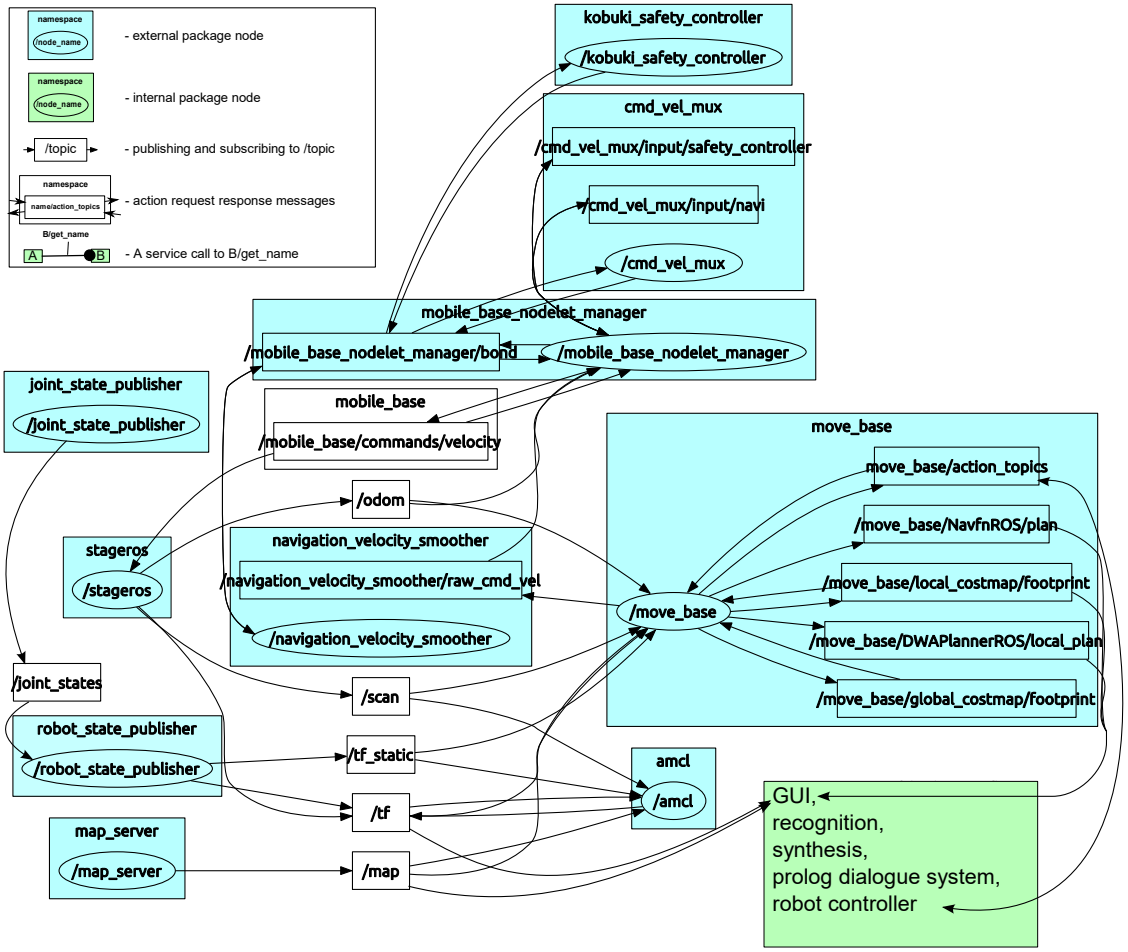

Figure 16. A detailed view of the final architecture (2/2)

"Internal package node" in Figure 15 and Figure 16 means that the node in question was either created from scratch or heavily modified (*prolog_server*). The source code of *"internal package"* nodes is located within the final project's files. *"External package"*, on the other hand, means that these nodes come with ROS or some other package and have to be installed separately. These nodes only have configuration files and other input files (e.g. map of the world) as part of the final project's filebase.

Note that Figure 16 omits the various services offered by stock nodes, which was done for readability. The graph was also simplified by pruning away leaf topics and dead sinks, which some of the stock ROS components generate, but which are not required as part of the natural language platform.

The following gives a quick overview of the nodes and their purpose.

Table 5. Node overview

node name	**description**
amcl	2D probablistic localization system.
chatbot	Interface that abstracts the robot implementation.
chatbot_gui	GUI node for viewing robot activity.
chatbot_speech_recognition	This a *speech_recognition_simple* type node. It captures audio and forwards it to the speech recognition system. It publishes the resulting transcript under /recognition/raw_result. Is pausable.
cmd_vel_mux	Robot velocity command multiplexer.
diagnostic_aggregator	Holds and categorizes diagnostic messages .
joint_state_publisher	Publishes robot joint state.
kobuki_safety_controller	Safety feature tied to bumpers and wheel drop events.
map_server	Keeps the map information and provides the data for the robot.
mobile_base_nodelet_manager	Manages robot movement
move_base	This accesses the navigation stack.Lets the robot move to desired position.
navigation_velocity_smoother	Bounds velocity messages according to robots

node name	description
	velocity and acceleration limits
prolog_server	Keeps an instance of the Prolog runtime. This is where the dialogue system is loaded into and queried.
robot_prolog_connection_node	Handles raw text coming from speech recognition. Queries prolog_server for appropriate reactions, prioritizes the reactions and executes commands related to the reactions on chatbot node.
robot_state_publisher	Publishes robot state, which is used in localization.
stageros	2d mobile robot simulator.
synth_festival	Synthesizes speech based on input text.

These services and the functionality they offer working in unison create the platform described in the thesis objectives.

3.8.2 A variation on the final configuration

There are variations possible to the configuration presented in Figure 15. For instance, we could replace *"the /recognition/chatbot_recognition"* node with a pair of nodes "*capture_vad_speex"* and *"speech_recognition_speex"* that achieve the same result.

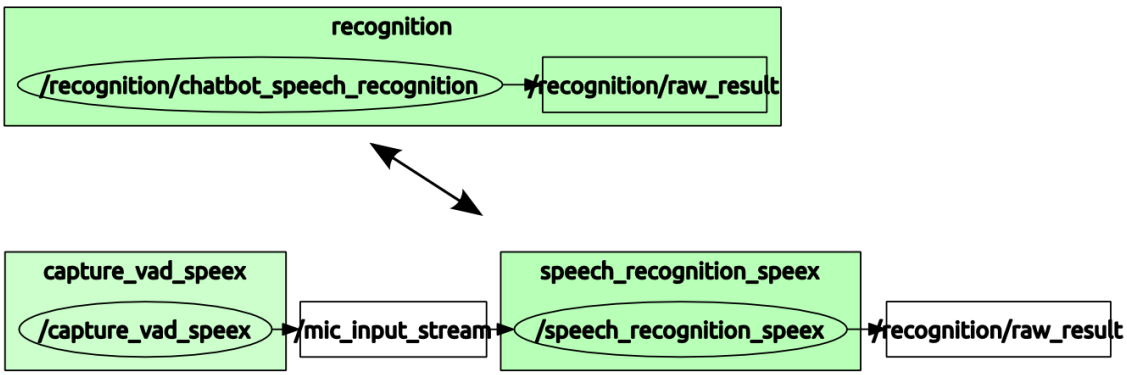

Figure 17. Contained vs separated functionality

The replacement in Figure 17 separates audio capture and speech recognition, but results in the same behaviour. There is overhead to separating the two nodes in the form

of transport cost over the ROS messaging protocol. In Figure 15 both capture and recognition are processed in one node and this overhead does not occur.

The benefit for this replacement is flexibility. Going even further, it would be possible to use ROS tooling to run nodes in different machines by communicating messages over the network [30], [32]. This could be used to separate the resource demanding speech recognition from the rest of the system.

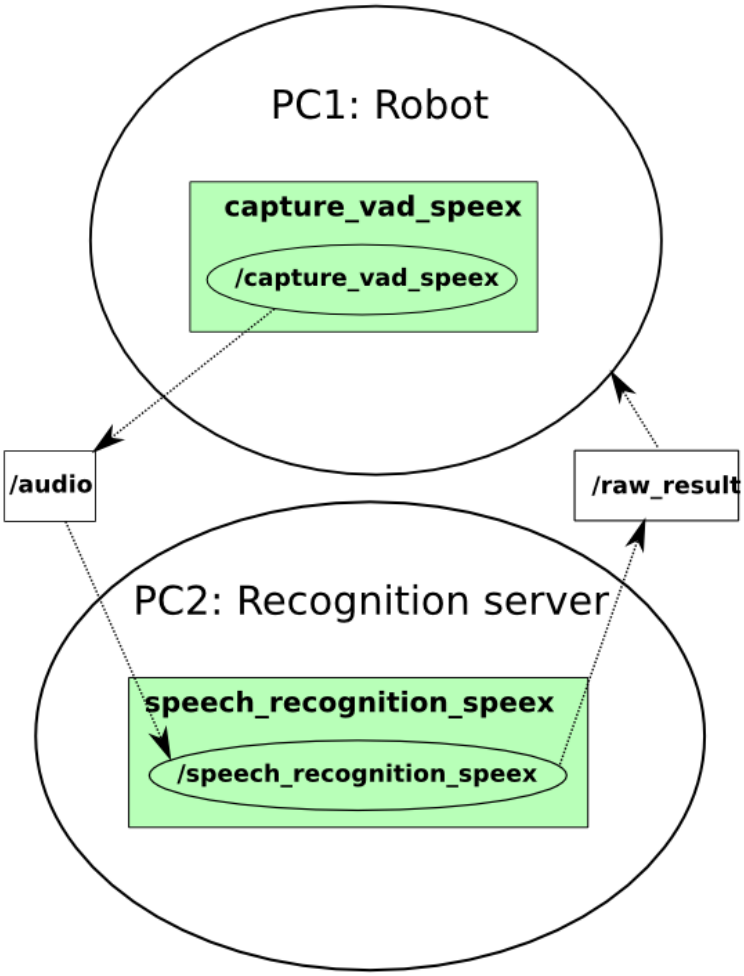

Figure 18. Nodes running in multiple machines

The separation across networks was not part of our final launch configuration as the system was optimized enough to run everything locally. It is offered here as an alternative for lower-end machines.

4 Component implementation details

This chapter gives a detailed overview of the components developed and used as part of the natural language controlled robot platform. All of the components follow common ROS architecture [22], which allows for easy replacement, reconfiguration, reusability and extension. They are listed here by their ROS node names.

We also offer a description of the nodes, ROS inputs and outputs, a descriptive graph and a list of parameters. Before any ROS node can be started, the ROS core must be started and our platform project's environment properly sourced.

```
~/platform$ source devel/setup.sh
~/platform$ roscore
```

Figure 19. Sourcing the platform and running roscore

For each component listed in this chapter, the parameters available are shown with their default values and a description. Note that the tilde (*"~"*) character in front of each of the parameters means they are in a private namespace for the node and are protected from collisions in the parameter server. When running a node we switch out the tilde for an underscore (*"_"*). An example of how to set the bitrate parameter when launching the audio capture node from command line is depicted in Figure 20.

```
~$ rosrun capture_audio capture_audio _bitrate:=192
```

Figure 20. Sourcing the platform and running roscore

The GST pipeline diagrams shown with some of the nodes, were auto-generated by running the pipeline stand-alone from the command line interface and by generating the diagrams [12]. The command describing how to start the pipeline in GST without ROS is also shown. This can be used to get a better understanding of how the audio streams are manipulated. The ROS component *rqt_graph* [26] was used to create the ROS diagrams.

4.1 capture_audio

4.1.1 Description

Capture_audio is the simplest component. It is a slightly simplified version of Nate Koenig's *audio_common* package [15]. It takes audio from the microphone, converts it into ROS compatible messages and publishes to a ROS topic. The motivation for integrating this component to our platform is twofold. Combined with *play_audio*, it can be used to quickly test if the microphone, ROS audio transport and playback is working properly. The second use for this could be to capture audio and transport it over the internet using *rosbridge_suite* [30] to a more powerful machine for speech transcription and command generation - this would mean that the robot itself can be very lightweight.

Capture_audio is used with the launch configuration **test_microphone.launch** (Appendix 3).

4.1.2 Diagrams

rqt_graph generated diagram of the node, showing the published audio topic:

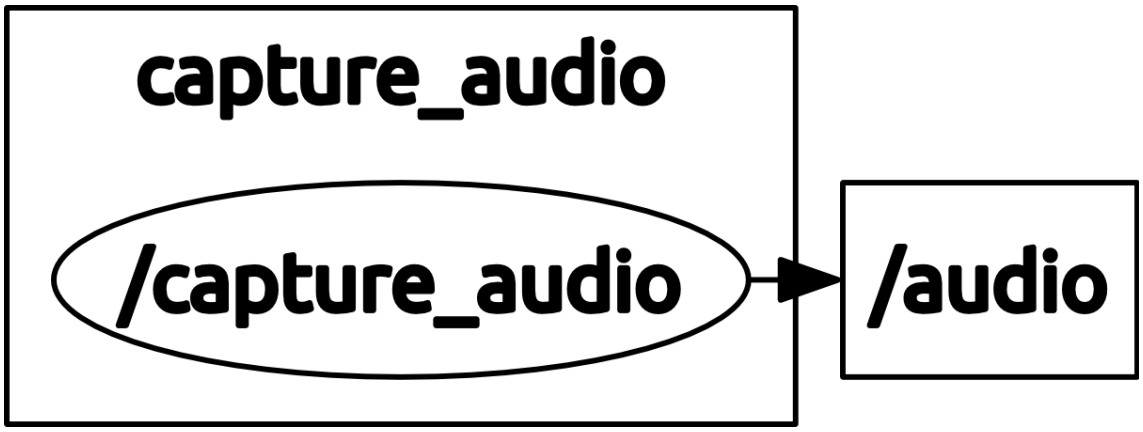

Figure 21. The *capture_audio* node and published topic

To create an equivalent GST pipeline that is running in the ROS node, the following launch command can be used:

```
~$ gst-launch-1.0 alsasrc ! audioconvert ! lamemp3enc quality=2.0 bitrate=192
! Appsink emit-signals=true max-buffers=100
```

Figure 22. GStreamer pipeline for *capture_audio*

The pipeline created shows how the audio is processed within the ROS node.

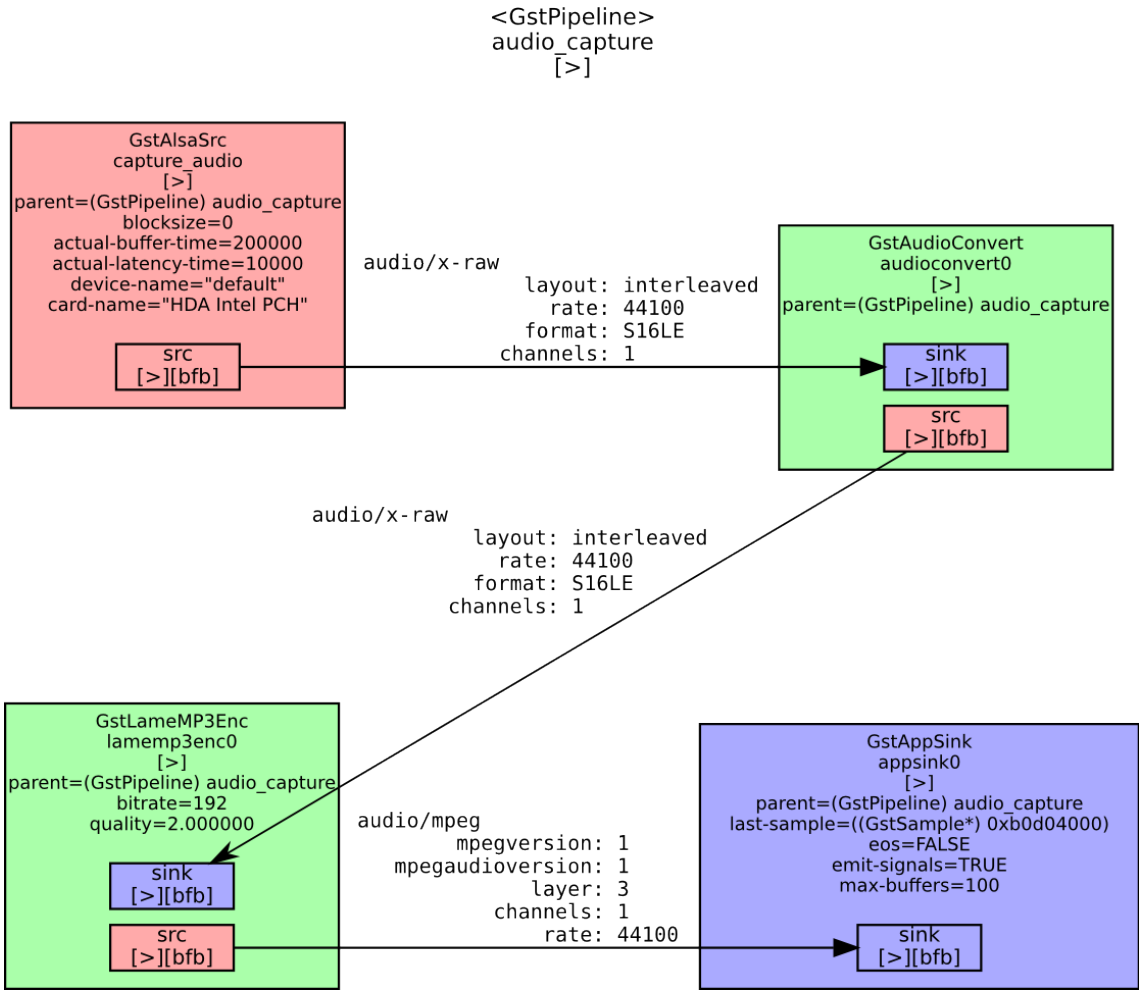

Figure 23. Audio processing in the *capture_audio* node

Within the ROS node, *appsink* is connected with an event handler that publishes to the topic *"/audio"*. The one-to-many nature of ROS topics allows any subscribers to receive the audio data.

Since *capture_audio* can also sink captured audio into a file (see *~dst* parameter), we have a second equivalent pipeline:

```
~$ gst-launch-1.0 alsasrc ! audioconvert ! lamemp3enc quality=2.0 bitrate=192
! filesink location=/tmp/capture.mp3
```

Figure 24. GStreamer pipeline for *capture_audio* with a file sink

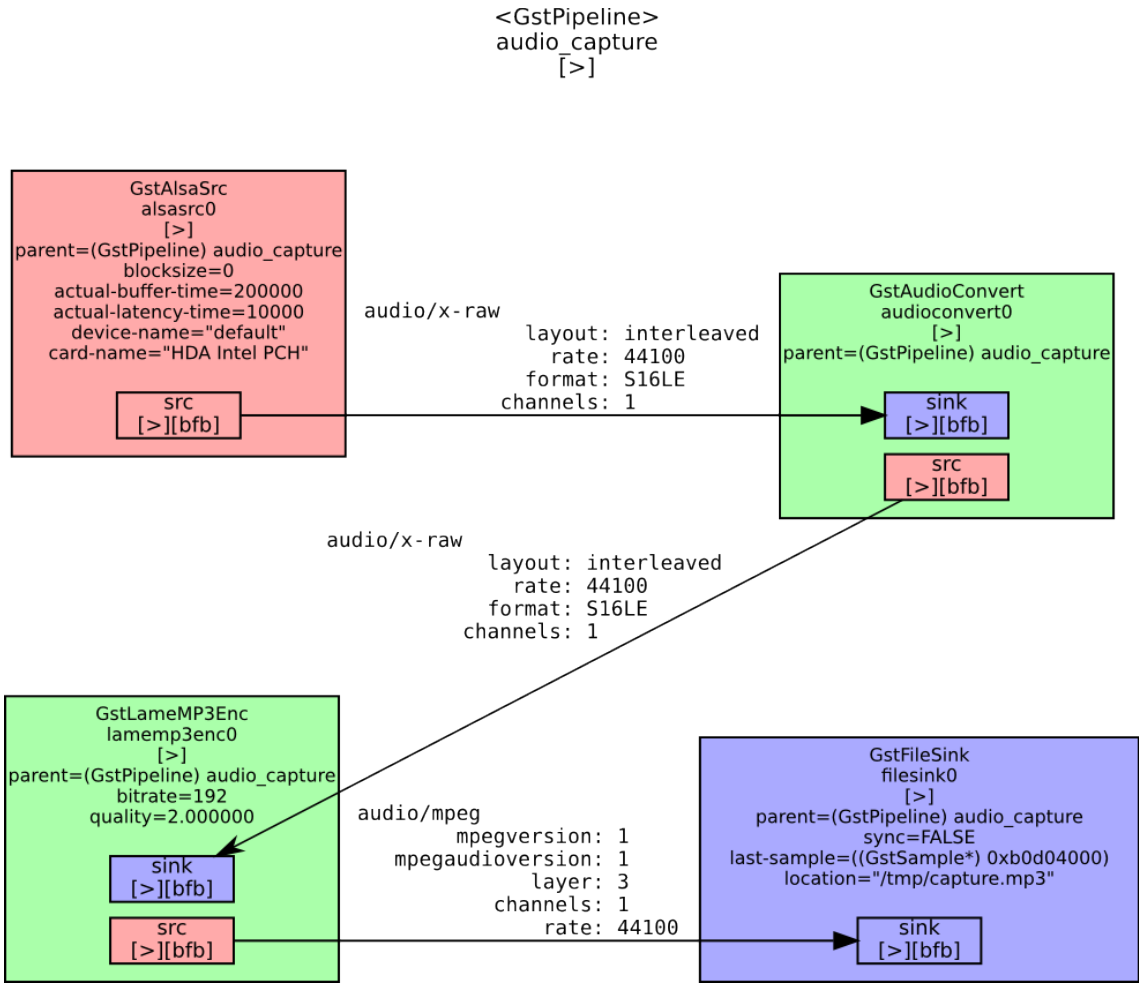

Figure 25. Audio processing in the capture node with a file sink

This pipeline fully works and creates a playable audio file *"/tmp/capture.mp3"*.

4.1.3 Running

The *"rosnode info"* command illustrates what *audio_capture* publishes. Figure 26 shows that the output of the GStreamer pipeline was converted to a ROS message of the type "*audio_common_msgs/AudioData"* and published with the topic name "*/audio"*.

```
~$ rosrun capture_audio capture_audio
~$ rosnode info /capture_audio
Node [/capture_audio]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /audio [audio_common_msgs/AudioData]
```

Figure 26. *capture_audio* node info and run command

Any ROS node can subscribe to the *"/audio"* topic and receive the audio buffers as they are being published. Note that *capture_audio* only publishes to the topic if file sink was not selected as the output type.

"/rosout" is a logging topic that is common to most nodes.

4.1.4 Parameters

Table 6. *capture_audio* run parameters

parameter	**type**	**default**	**effect**
~format	string	“mp3”	Allows selecting the type of encoder used in the nodes GStreamer pipeline. The node supports “mp3” and “wave”. MP3 encoding is done using the premade gstreamer element lamemp3enc [10]. WAVE is just raw audio [6].
~bitrate	unsigned integer	192	Applies for MP3 format, allows selecting the encoding bitrate.
~channels	unsigned integer	1	Applies for WAVE format, Allows selecting between mono, stereo and multichannel output.
~depth	unsigned integer	16	Applies for WAVE format, allows selecting the amount of bits per sample. Capture from the card happens via alsasrc [7] GStreamer plugin and is converted to desired depth via audioconvert [8] GStreamer plugin.
~sample_rate	unsigned integer	16000	Applies for WAVE format, allows specifying capture sample frequency.
~dst	string	“appsink”	This parameter allows selecting the output of the GStreamer element. If “appsink” is selected, GStreamer will link with the ROS node and start publishing the audio buffers to the “*/audio*” topic. Otherwise, if a file path is entered here the captured audio will be written to that file in the format desired and specified with the parameters described above. This is useful if you want to capture and store audio for later use. For instance, if we want to test performance of different speech recognition parameters, we want the input to be constant and invariable to the tests.

4.2 play_audio

4.2.1 Description

This is the counterpart to *capture_audio*. The component takes the audio received via ROS topic and either saves it to a file or plays it from the speakers.

This element is used in the launch configuration **test_microphone.launch** (Appendix 3).

4.2.2 Diagrams

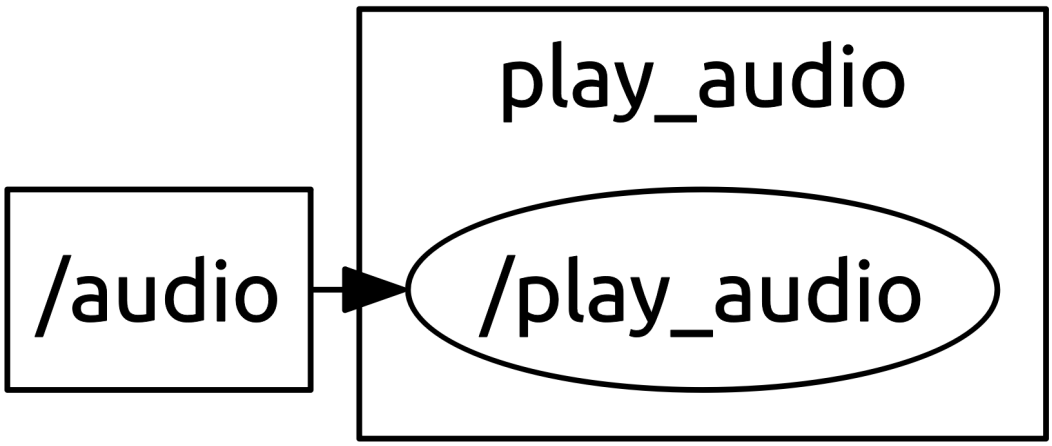

Figure 27. The *play_audio* node and subscribed topic

To create an equivalent GST pipeline that is running in the ROS node, the following launch command can be used:

```
~$ gst-launch-1.0 appsrc ! audioconvert ! autoaudiosink
```

Figure 28. GStreamer pipeline for *play_audio*

The pipeline created shows how the audio is processed within the ROS node.

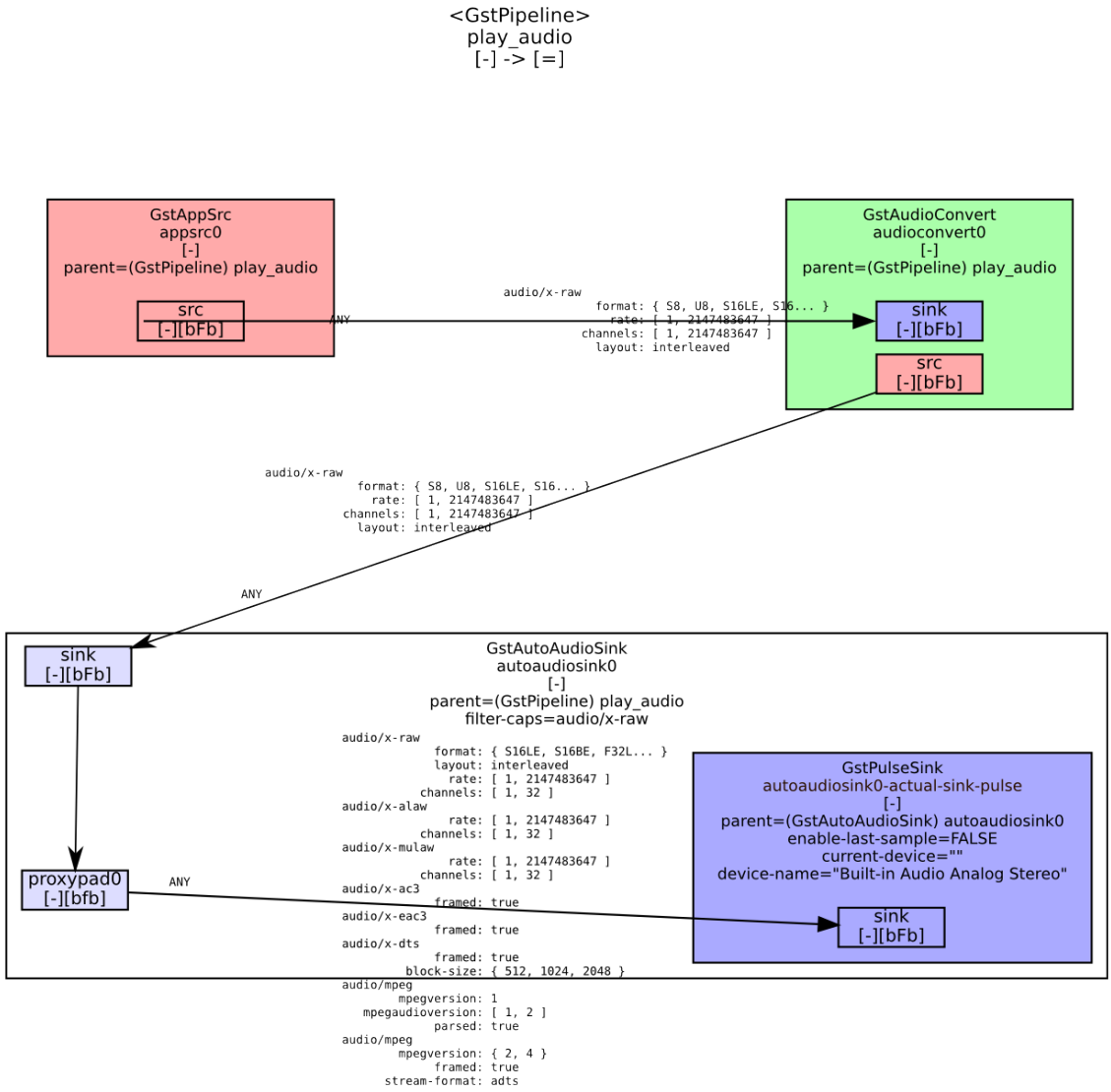

Figure 29. Audio processing in the *play_audio* node

The first *appsrc* element in the pipeline is hooked up to the *"/audio*" ROS topic. Each message published to that topic gets picked up by an event handler in the node and forwarded to the GStreamer. The *audioconvert* component then converts the acquired audio stream into a format acceptable by the *autoaudiosink* component. The *autoaudiosink* component automatically detects available audio sink components installed on the machine by doing a registry scan. It then connects the pipeline to that component and pipes the audio data to it [9]. This will most commonly end up being the audio card and will play the audio from the speakers.

play_audio can also save information received over the ROS topic to a file (see *~dst* parameter). This is useful for validating what was sent versus what was received, doing experimentation and debugging. GStreamer equivalent pipeline to what is running in the ROS node:

```
~$ gst-launch-1.0 appsrc ! filesink location=/tmp/played.buffers
```

Figure 30. GStreamer pipeline for *play_audio* with a file sink

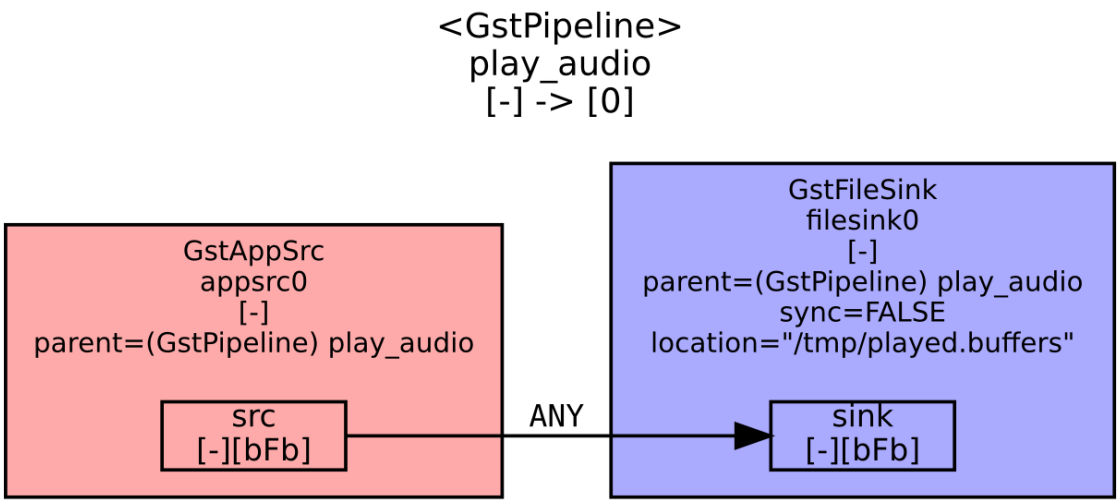

Figure 31. Audio processing in the *play_audio* node with file sink

As Figure 31 shows, data received over the topic is saved directly to a file determined by the *location* parameter.

4.2.3 Running

```
~$ rosrun play_audio play_audio
~$ rosnode info /play_audio
Node [/play_audio]
Publications:
 * /rosout [rosgraph_msgs/Log]
Subscriptions:
 * /audio [audio_common_msgs/AudioData]
```

Figure 30. *play_audio* node info and run command

Node inspection reveals that the node subscribes to the "*/audio*" topic of message type *"audio_common_msgs/AudioData"*. This is the message that *capture_audio* publishes.

4.2.4 Parameters

Table 7. *play_audio* run parameters

parameter	**type**	**default**	**effect**
~dst	string	"alsasink"	This parameter allows selecting the output of the GStreamer element. If "alsasink" is selected, GStreamer will take the buffers received over the "*audio*" topic and convert the stream to a suitable format for a device found by *autoaudiosink* [9]. If a file path is entered here, the received audio will be written directly to a file without any conversions.

4.3 capture_vad_speex

4.3.1 Description

This node is an enhanced version of *capture_audio*. There are two main differences:

1. Instead of mp3 or raw wave, audio is encoded in the ***speex*** format [33].

The speex audio compression is specifically designed and optimized for speech.

2. The node provides voice activity detection (VAD).

VAD is used to detect silences and only send whole sentences to the voice recognition node. This is useful in cutting up the audio into segments for transfer over the network. Kaldi is already capable of endpointing by itself - this is an early optimization that lightens the load on the speech recognition engine.

This element is used in the launch configurations **test_vad_speex.launch** and **test_sr_speex.launch** (Appendix 3).

4.3.2 Diagrams

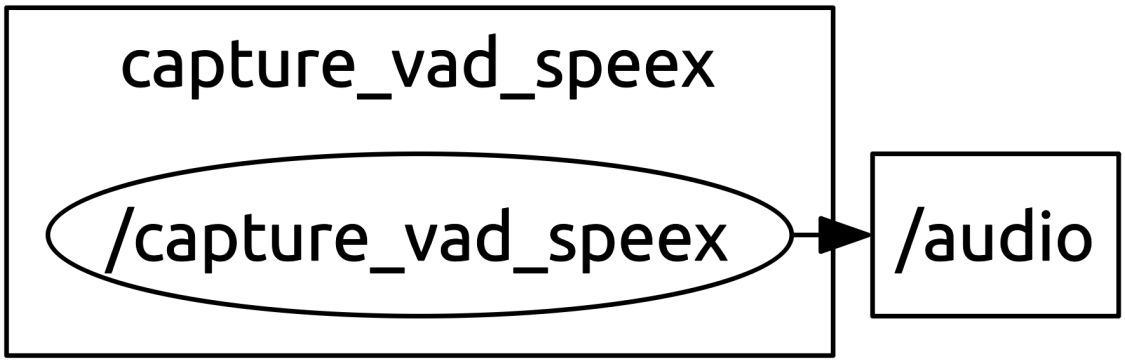

Figure 33. The *capture_vad_speex* node and published topic

To create an equivalent GST pipeline that is running in the ROS node, the following launch command can be used:

```
~$ gst-launch-1.0 alsasrc ! audioconvert ! audioresample ! audio/x-raw,
rate=32000 ! speexenc vad=true dtx=true bitrate=0 ! appsink emit-signals=true
max-buffers=100
```

Figure 34. GStreamer pipeline for *capture_vad_speex*

Based on the pipeline, a diagram showing GStreamer configuration in the ROS node is generated.

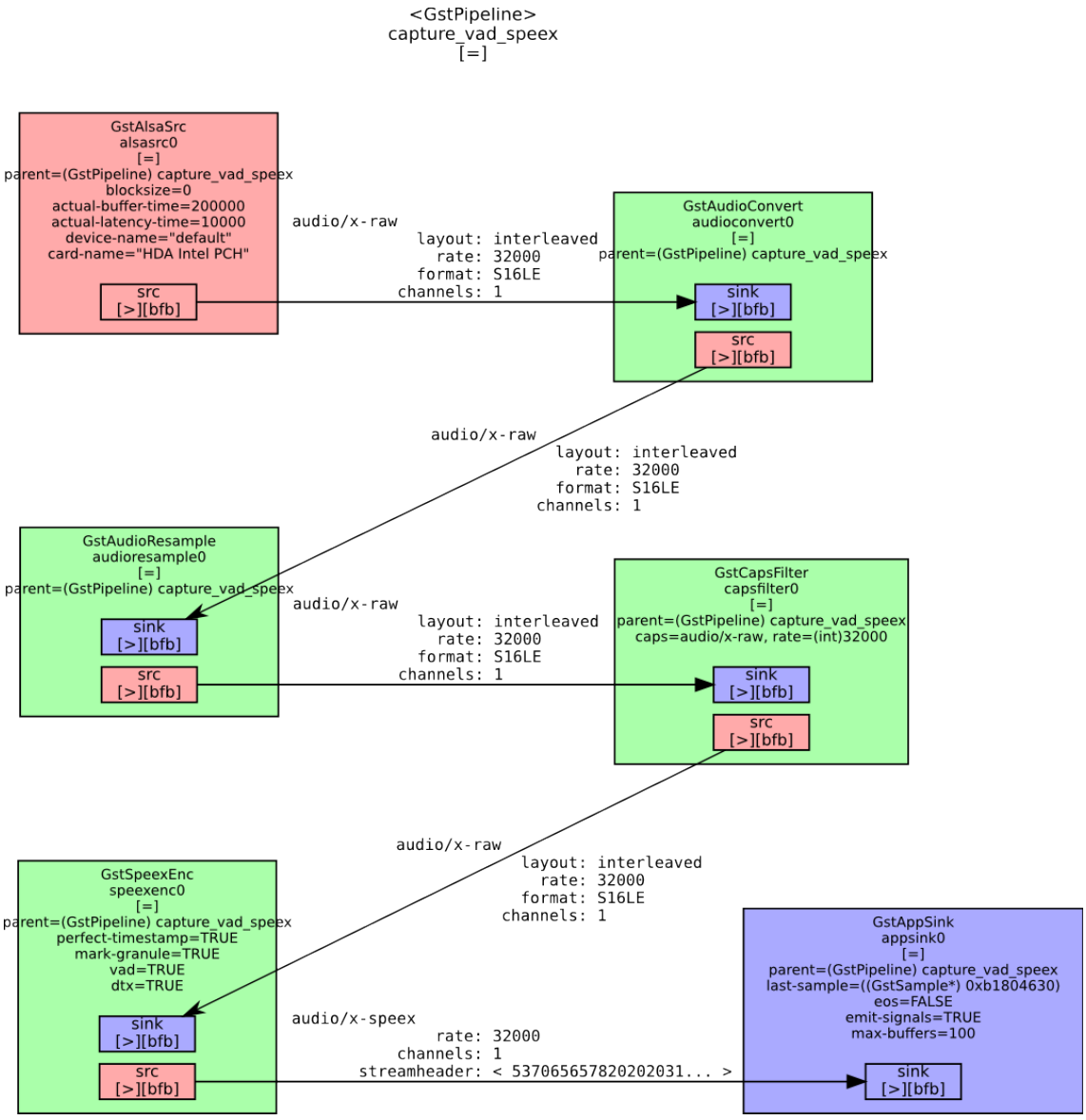

Figure 35. Audio processing in the *capture_vad_speex* node

Figure 35 shows that an audio resampling element is used within the pipeline. This was not strictly necessary, but useful because the speex codec is optimized to for sampling rates of 8 kHz, 16 kHz and 32 kHz. The audio card used had a default sampling rate of 44100 Hz, which was downsampled to 32 kHz to get a better compression rate out of the speex codec.

The *capture_vad_speex* node also has the capability of saving capture audio to a file using using the *~dst* parameter.

```
~$ gst-launch-1.0 alsasrc ! audioconvert ! audioresample ! audio/x-raw,
rate=32000 ! speexenc vad=true dtx=true bitrate=0 ! filesink
location=/tmp/capture.speex
```

Figure 36. GStreamer pipeline for *capture_vad_speex* with a file sink

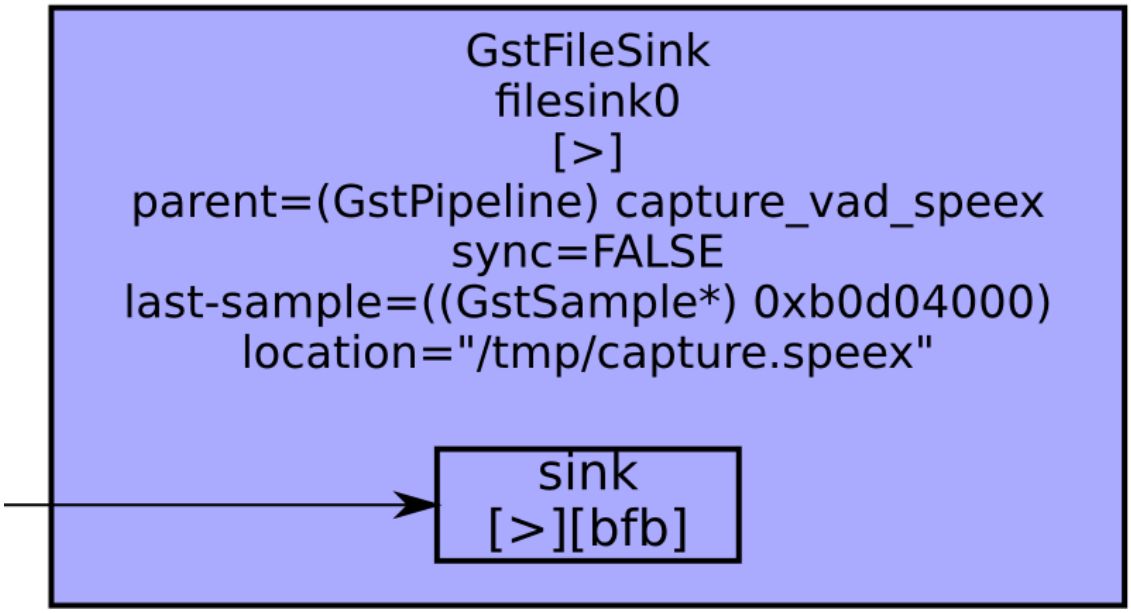

Figure 37. File sink replacement for the *capture_vad_speex* pipeline

The result is a speex encoded audio file, which may be listened to with speex capable players.

4.3.3 Running

```
~$ rosrun capture_vad_speex capture_vad_speex
~$ rosnode info /capture_vad_speex
Node [/capture_vad_speex]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /audio [audio_common_msgs/AudioData]

Subscriptions: None
Services:
 * /capture_vad_speex/get_sink_capabilities
 * /capture_vad_speex/get_loggers
 * /capture_vad_speex/set_logger_level
```

Figure 38. *capture_vad_speex* node info and run command

In addition to publishing "*/audio"* messages, *capture_vad_speex* also offers a custom service "*capture_vad_speex/get_sink_capabilities*". This additional service is necessary because GStreamer's *appsink* and *appsrc* elements are not able to appropriately negotiate the audio format for speex encoded data. The service returns a string describing the audio format, so the consumer of the audio can setup a matching pipeline. Each node that wants to use this data, must first specify the audio format. The service *get_sink_capabilities* helps with this. A slightly less flexible alternative would hav e been to just hardcode an audio format. The function *play_audio_speex.cpp::PlayAudioSpeex::getSinkCapabilities()* found in the source code (Appendix 1) details how to setup the receiver.

4.3.4 Parameters

Table 8. capture_vad_speex run parameters

parameter	**type**	**default**	**effect**
~bitrate	unsigned integer	0	Speex encoding bitrate. 0 means automatic.
~dst	string	"appsink"	"*appsink*" means sinking the audio to ROS and publishing to the "*/audio*" topic. If *~dst* is a file path, the captured and speex compressed audio will be saved to that file.

4.4 play_audio_speex

4.4.1 Description

This is the counterpart to *capture_vad_speex*. The component takes the audio received via ROS and either saves it to a file or plays it from the speakers.

This element is used in the launch configuration **test_vad_speex.launch** (Appendix 3).

4.4.2 Diagrams

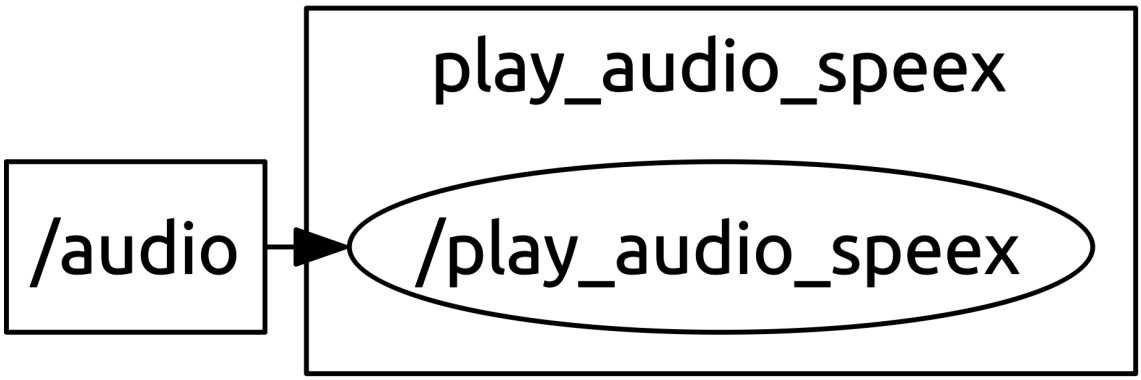

Figure 39. The *play_audio_speex* node and subscribed topic

To create an equivalent GST pipeline that is running in the ROS node, the following launch command can be used:

```
~$ gst-launch-1.0 appsrc ! speexdec enh=false ! audioconvert ! autoaudiosink
```

Figure 40. GStreamer pipeline for *play_audio_speex*

The pipeline created shows how the audio is processed within the ROS node.

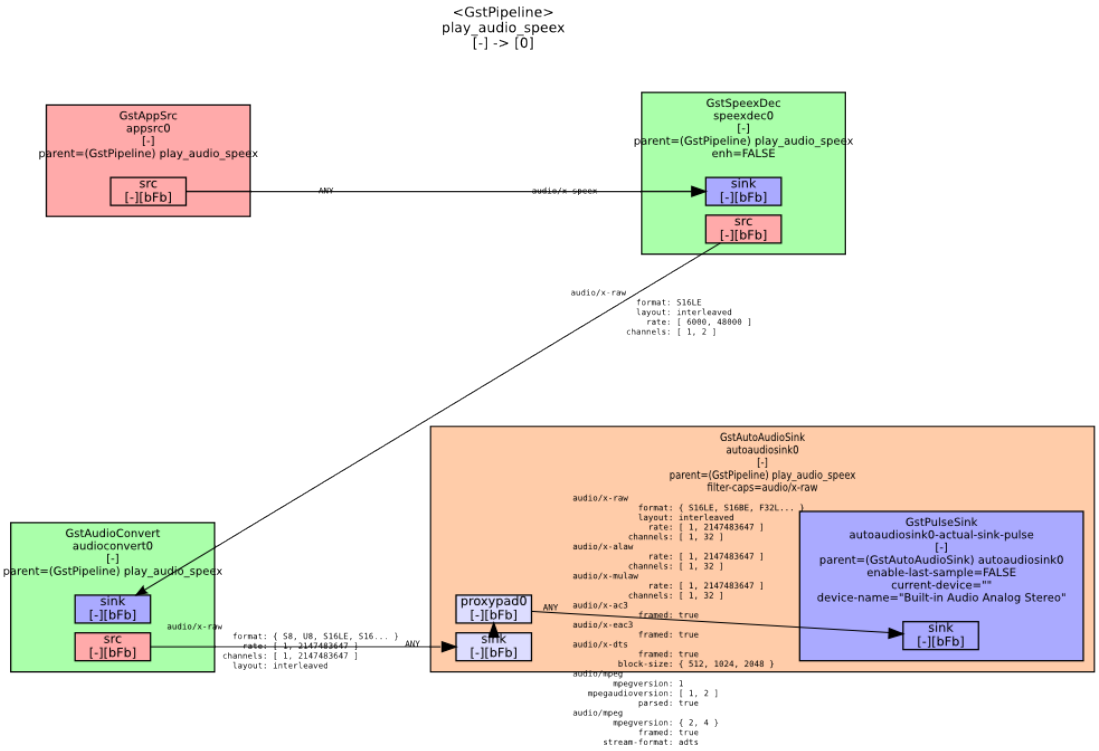

Figure 41. Audio processing in the *play_audio_speex* node

Figure 41 shows that an audio decoder has been added to the pipeline. This decoder makes it possible to play speex encoded streams. The rest of the pipeline is analogous to *play_audio.* Just like *play_audio,* this node has *~dst* parameter, which can be used to save the received stream directly to a file.

4.4.3 Running

```
~$ rosrun play_audio_speex play_audio_speex
~$ rosnode info /play_audio_speex
Node [/play_audio_speex]
Publications:
 * /rosout [rosgraph_msgs/Log]
Subscriptions:
 * /audio [unknown type]
```

Figure 42. *play_audio_speex* node info and run command

This is the same as *play_audio* in Chapter 4.2. An important distinction is that *play_audio_speex* requires *capture_vad_speex* to be running before it can do anything. This is because the nodes need to negotiate the speex audio format. If *play_audio_speex*

is launched alone, it will wait until *capture_vad_speex* makes the "*/capture_vad_speex/get_sink_capabilities"* service available.

4.4.4 Parameters

Table 9. *play_audio_speex* run parameters

parameter	**type**	**default**	**effect**
~dst	string	"alsasink"	Allows choosing between alsasink and file sink (see Chapter 4.2.4).
~enh	boolean	false	Enable or disable speex perceptual enhancement [40].

4.5 speech_recognition_simple

4.5.1 Description

The *speech_recognition_simple* node captures audio, does speech recognition using *gst-kaldi-nnet2-online* and outputs the transcript of what is spoken under the ROS topic *"/raw_result"*. This component cuts away the overhead of transmitting audio over the XML-RPC communication protocol and does both audio capture and recognition in the same node. It also removes the overhead of additional encoding and decoding, which would be necessary for audio transfer. The details on why this overhead occurs are highlighted in *speech_recognition_speex* (Chapter 4.6). The downside of this optimization is that it is less flexible – audio capture and speech recognition are in one executable. This node is used in the final platform, but low-end computers may need to use a dedicated speech recognition server and the *speech_recognition_speex* node, since this node is resource intensive.

Finally note that *speech_recognition_simple* is used in the launch configurations **test_sr_simple.launch** and **chatbot.launch** (Appendix 3).

4.5.2 Diagrams

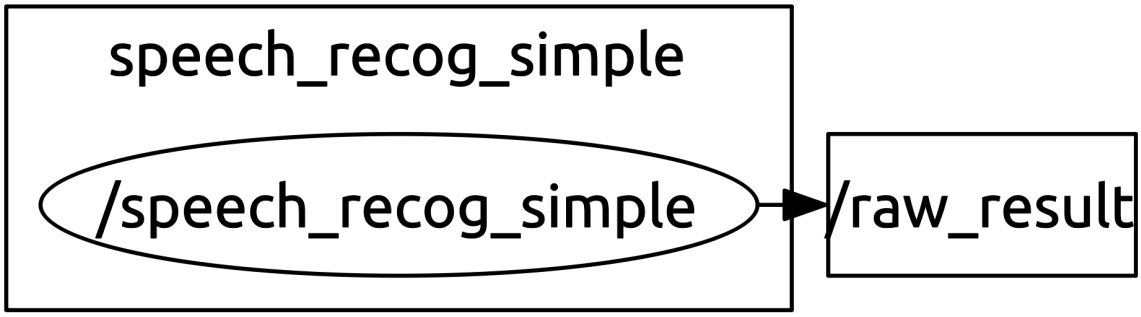

Figure 43. The *speech_recognition_simple* node and published transcript

If the *~dst* parameter is set to "*appsink*", the node publishes transcript text to the *"/raw_result"* topic. The following GST pipeline is equivalent to what is running in the ROS node if we set the *~dst* parameter to "*/home/oliver/test.txt*":

```
~$ gst-launch-1.0 alsasrc \
! audioconvert ! audioresample ! kaldinnet2onlinedecoder \
use-threaded-decoder=false \
model=$GST_PLUGIN_PATH/final.mdl \
fst=$GST_PLUGIN_PATH/HCLG.fst \
word-syms=$GST_PLUGIN_PATH/words.txt \
feature-type=mfcc \
mfcc-config=$GST_PLUGIN_PATH/conf/mfcc.conf \
ivector-extraction-
config=$GST_PLUGIN_PATH/conf/ivector_extractor.fixed.conf \
max-active=10000 beam=10.0 lattice-beam=8.0 chunk-length-in-secs=2.0 \
traceback-period-in-secs=1.0 do-endpointing=true \
endpoint-silence-phones=1:2:3:4:5:6:7:8:9:10 \
! filesink location=/home/oliver/test.txt buffer-mode=2
```

Figure 44. GStreamer pipeline for *speech_recognition_simple*

Note that gst-launch-1.0 needs to know where the *kaldinnet2onlinedecoder* plugin is located (see Chapter 3.5). This is ensured by setting an environment variable.

```
~$ export GST_PLUGIN_PATH=/home/oliver/thesis/catkin_ws/deps
```

Figure 45. Setting Gstreamer plugin path

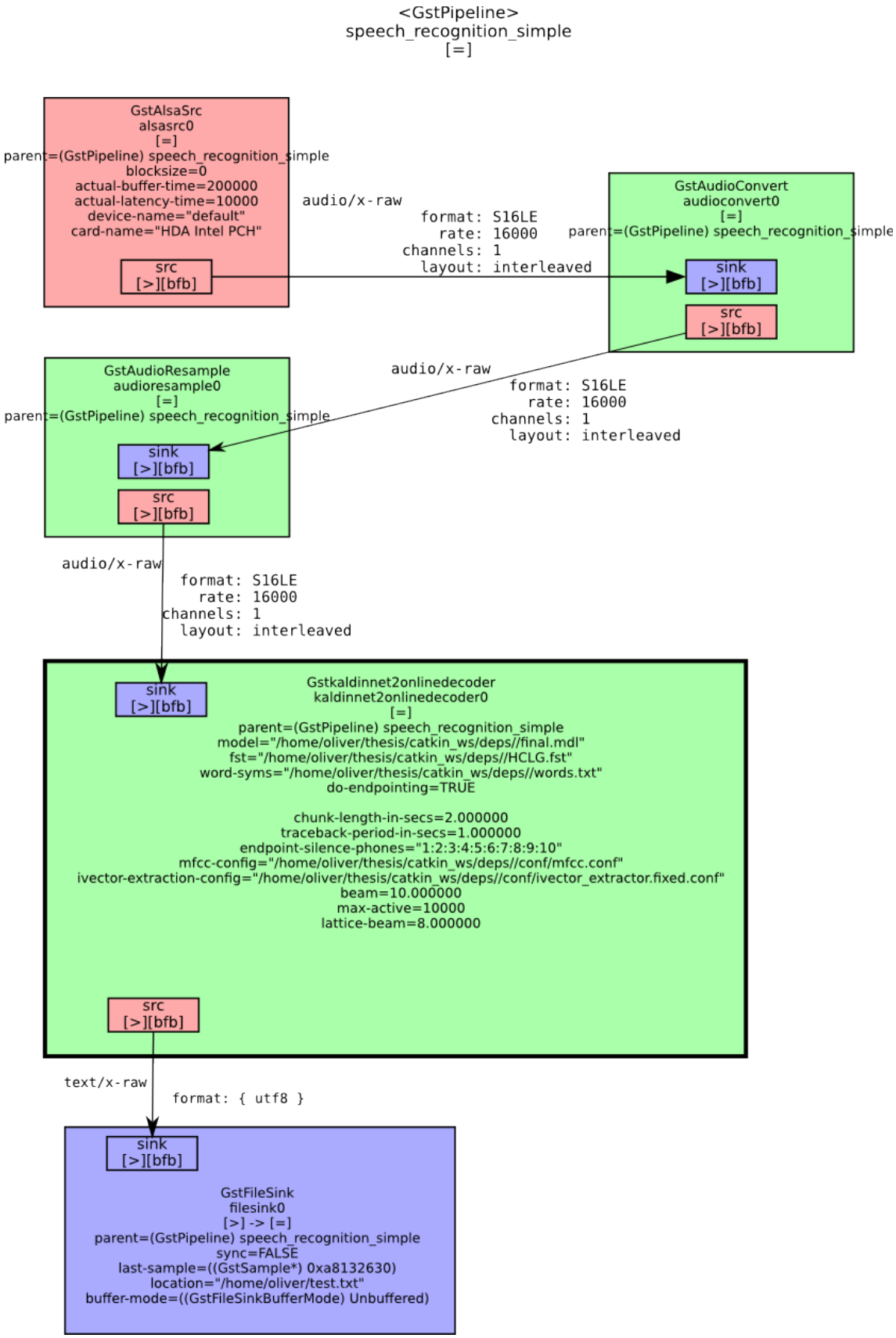

Figure 46. Processing audio into transcript text in the *speech_recognition_simple* node

The resampling and conversion in Figure 46 is necessary to convert the audio data to a format matching the *kaldinnet2onlinedecoder*'s sink specifications. It only accepts audio streams of a certain format (see Chapter 3.5). From the diagram we can see that the output format is an utf8 encoded string.

After speaking a couple of sentences the example file "*/home/oliver/test.txt*" contains:

```
tere üks kaks kolm neli viis
testime kõne süntees see aastal kaks tuhat kuus teist
robot normaalne raamat
robot too mulle raamat
kuidas läheb
```

Figure 47. Speech transcription results example

4.5.3 Running

```
~$ rosrun speech_recognition_simple speech_recognition_simple --
_model_dir:=/home/oliver/thesis/catkin_ws/deps
~$ rosnode info /speech_recog_simple
Publications:
 * /raw_result [std_msgs/String]
 * /rosout [rosgraph_msgs/Log]
Subscriptions: None
Services:
 * /set_pause
```

Figure 48. *speech_recognition_simple* node info and run command

Figure 48 node information shows that speech recognition results are published to the topic "*/raw_result*" with message type of "*std_msgs/String*". Any node can subscribe to this and get a transcript of what is spoken. A service named "*/set_pause*" is also available. This service can be called from other nodes or the command line to pause and unpause speech recognition. This feature was necessary so that speech recognition could

be paused while the speech synthesizer was speaking. Without doing that, the robot would hear itself speak and get stuck in a loop.

4.5.4 Parameters

The descriptionless parameters in Table 10 correspond one-to-one with the parameters of the *kaldinnet2onlinedecoder* plugin (Chapter 3.5) - the ROS node just wraps them and assigns a default value.

Table 10. speech_recognition_simple run parameters

parameter	**type**	**default**	**description**
~dst	string	"appsink"	If "*appsink*" is selected, GStreamer will link publish speech transcript to the "*/raw_result*" topic. Otherwise, if a file path is entered here, the text will be written to a file. It is also possible to specify "*/dev/stdout*" here to print the recognition results to the standard output stream.
~model_dir	string	"error"	This is a parameter that **must be passed** to the node when running, if left empty a fatal error will occur. It tells the node where the speech recognition corpus along with configuration files are located. It uses the string passed as a root path for looking up the configuration files. It is also possible specify each config separately in case the standard directory layout is not acceptable via *~model, ~fst, ~word_syms, ~mfcc_config* and *~ivector_extraction_config* parameters.
~threaded_decoder	bool	false	-
~do_endpointing	bool	true	-
~model	string	"model_dir/ final.mdl"	-
~fst	string	"model_dir /HCLG.fst"	-
~word_syms	string	"model_dir	-

parameter	type	default	description
		/words.txt”	
~feature_type	string	“mfcc”	-
~mfcc_config	string	“model_dir /conf/mfcc.conf”	-
~ivector_extraction_config	string	“model_dir /conf/ivector_extractor.fixed.conf”	-
~ep_silence_phones	string	"1:2:3:4:5:6:7:8:9:10"	-
~max_active	int	7000	-
~beam	float	10.0	-
~lattice_beam	float	5.0	-
~chunk_length_s	float	2.0	-
~acoustic_scale	float	0.0833	-
~traceback_period_in_secs	float	1.0	-

4.6 speech_recognition_speex

The only difference between *speech_recognition_speex* and *speech_recognition_simple* is that *speech_recognition_speex* takes audio captured by a different node, while *speech_recognition_simple* captures the audio in the same executable.

4.6.1 Diagrams

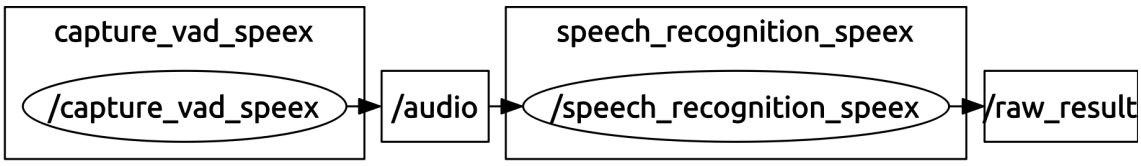

Figure 49. The *speech_recognition_speex* node with published transcript and input audio topics

Figure 49 shows that, unlike *speech_recognition_simple*, the audio is captured separately. This offers additional flexibility – like running the nodes on different computers. Because of the speex decoding done in speech_recognition_speex the same limitations are present as were for the node *play_audio_speex*, i.e. the capture node must be launched and the audio format negotiated (Chapter 4.4).

Because the *kaldinnet2onlinedecoder* can not link directly with a speex encoded stream the audio must be decoded and converted before passing to audio recognition. This is done in exactly the same way as the stream in Figure 41 – with a *speexdec* decoder element. Getting rid of this extraneous decoding is one of the future optimization opportunities listed in Appendix 2.

4.6.2 Running

```
~$ rosrun speech_recognition_speex speech_recognition_speex --
_model_dir:=/home/oliver/thesis/catkin_ws/deps
~$ rosnode info /speech_recog_simple
Node [/speech_recognition_speex]
Publications:
 * /raw_result [std_msgs/String]
 * /rosout [rosgraph_msgs/Log]
Subscriptions:
 * /audio [audio_common_msgs/AudioData]
Services:
 * /speech_recognition_speex/set_logger_level
 * /speech_recognition_speex/get_loggers
 * /set_pause
```

Figure 50. *speech_recognition_speex* node info and run command

The only difference from *speech_recognition_simple* is the subscription to audio.

4.6.3 Parameters

Parameters are the same *speech_recognition_simple*, with the addition of the *~enh* parameter for perceptual enhancement, which is a feature of the speex codec.

4.7 synth_festival

This ROS node wraps the festival speech synthesis toolkit (Chapter 3.6). It takes text input from the *"/synth_festival/text_to_speak"* topic and converts it to speech in the voice specified with the *~voice* parameter. This node also calls *"/set_pause"* on *speech_recognition_simple* and *speech_recognition_speex* so the robot ignores itself speaking.

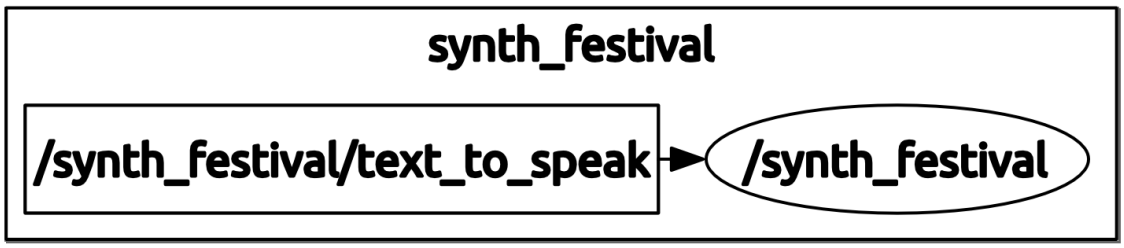

Figure 51. The *synth_festival* node and subscribed topic

The only parameter for this node is a string called *~voice* which lets the user select between installed voices. In our platform it defaults to an Estonian voice *"voice_eki_et_riina_clunits"* from The Institute of the Estonian Language [5].

4.8 chatbot_gui

The *chatbot_gui* node gives the platform a simple graphical user interface. Its appearance and features are demonstrated in Chapter 2.3. This chapter takes a closer look at this ROS node. The GUI was written in python and is based on ROS's rqt framework [13].

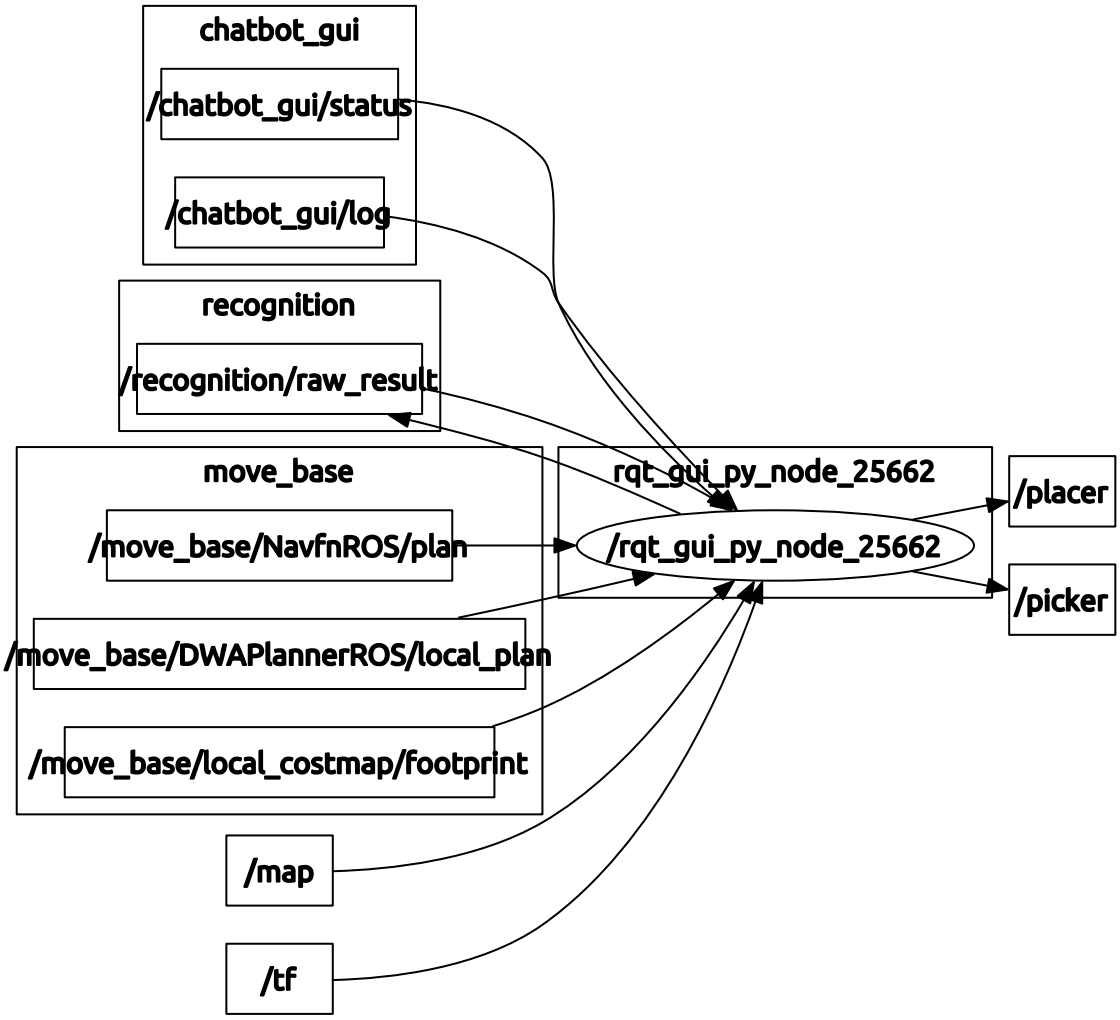

Figure 52. GUI node subscriptions and publications

Figure 52 displays all of the messages handled by the GUI. These can be linked to various GUI elements.

"/chatbot_gui/status" - This subscription aligns with the status bar. Any message published to this topic will show up in the status box.

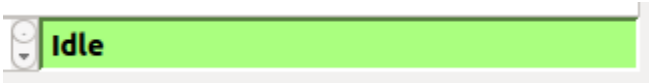

Figure 53. GUI - status box

"/chatbot_gui/log" and *"/recognition/raw_result"* as a subscription – Any string published to these topics is shown in the message log section.

	Timestamp	
1	19.12.2016 19:20:30.752	response:jĀµudsin kohale
2	19.12.2016 19:20:30.711	goto completed.
3	19.12.2016 19:20:13.912	goto: koridor[36,13]
4	19.12.2016 19:20:13.911	response:sain aru hakkan minema
5	19.12.2016 19:20:13.609	heard: 'mine koridori'

Figure 54. GUI - message log

Note from Figure 52 that *"/recognition/raw_result"* is also an output of the GUI. This was done to provide a convenient way to give the dialogue system input, instead of relying on speech recognition.

Figure 55. GUI - fake recognition results

The "*/placer"* and *"/picker"* topics are there because the GUI offers stub functionality for picking up items and placing them down.

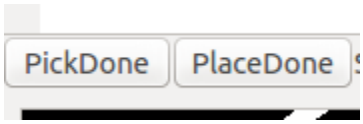

Figure 56. GUI - fake picking up and placing down objects

The rest of the topics provide navigation information, a world map and the robot's coordinates respectively.

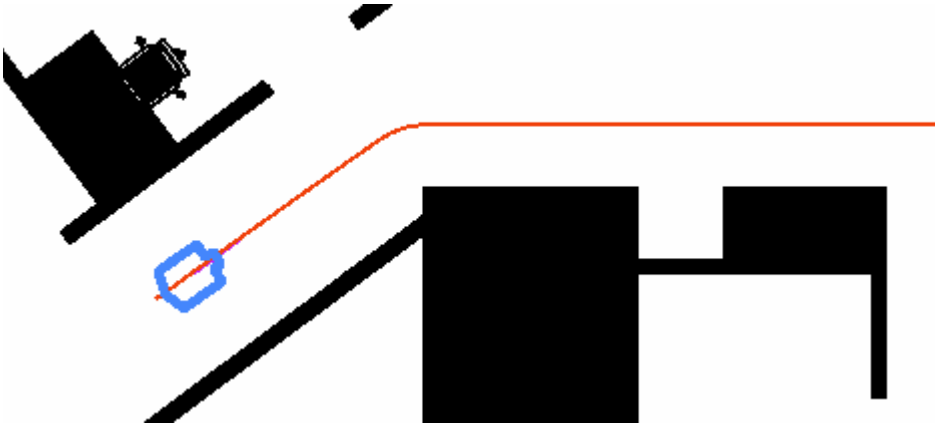

Figure 57. GUI - map and nav

4.9 chatbot_simulator

The created *chatbot_simulator* node package has no source code on its own. The package is essentially a collection of configuration files that were created as part of this thesis. The package launches a simulated environment that suits our needs. The configuration files launch the stage simulator with the correct map file and robot parameters. They also start the navigation stack, set the initial pose of the robot and start the *safety_controller* and *velocity_smoother* nodes mentioned in Chapter 2.4. It is easy to edit the robot's shape and size and world's costmap from these configuration files. The configuration files were based on ROS turtlbot's stage simulator setup [4].

4.10 prolog_server

The *prolog_server* node hosts a Prolog runtime, which is used to run the robot's dialogue system. The dialogue system is a Prolog program that is loaded into this runtime by the *robot_prolog_connection* node and then queried.

Prolog_server is a modified version of the *ros-prolog* package [13]. As part of the thesis work, a subclass called “SimpleServer” was created, that only implements features required for the robot platform's operations. Multibyte encoding support was also added to the package, to ensure proper Estonian character handling in the dialogue system.

4.11 chat_core

The *chat_core* node was created by the author, to act as an interface and abstract away the robot implementation. All operations/calls to the robot are passed to this node. The *chat_core* node then takes advantage of ROS's plug-and-play nature and forwards the calls to an actual implementation or a stub implementation.

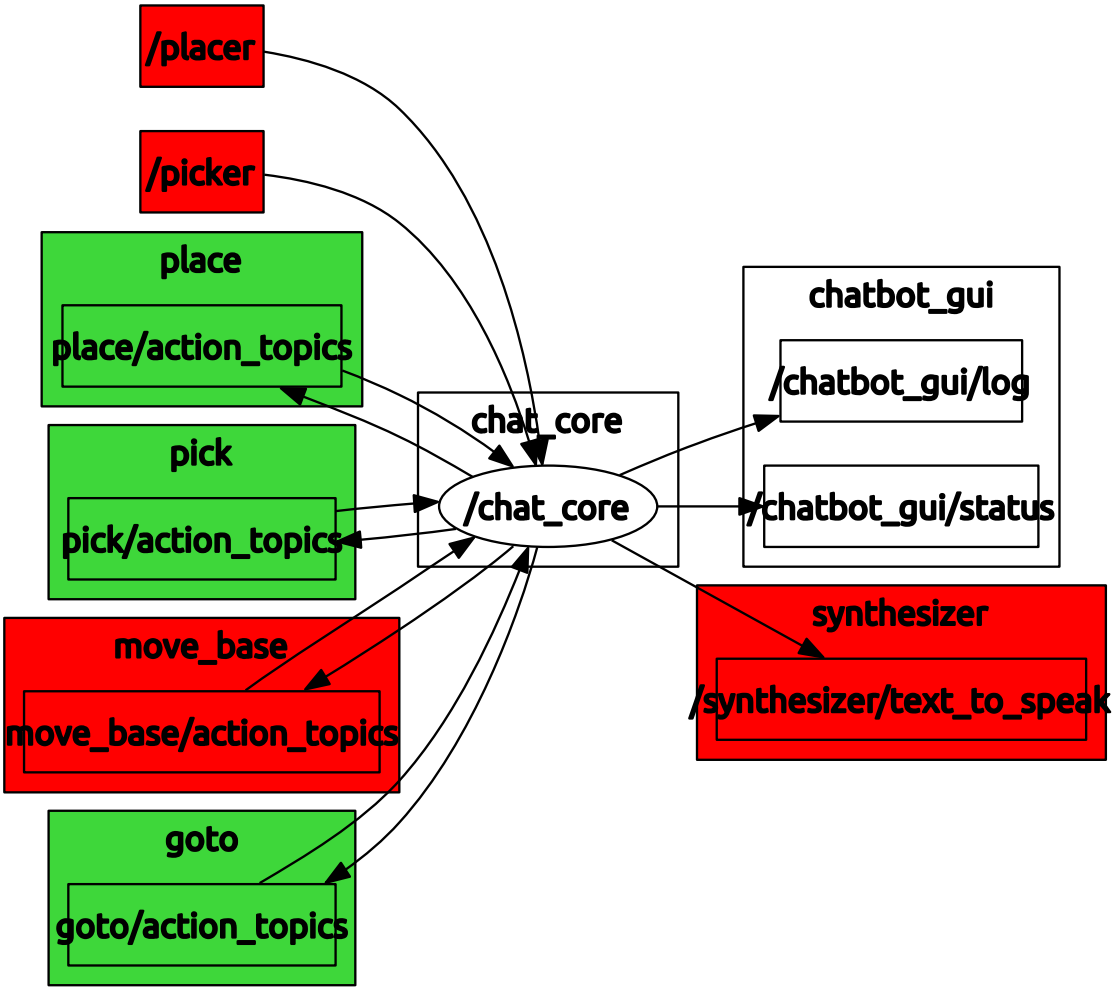

Figure 58. The chat_core node and its connections

Figure 58 shows action servers (Chapter 3.1.3) offered by the node with a green background. These are the calls to the interface. Marked with a red background are the actual implementations or stub implementations in the case of *"/pick"* and *"/place"*. As described in Chapter 4.8 the published topics under the *chatbot_gui* namespace are for logging purposes.

Note that there are four implementations and only 3 interface calls. The respond command that gets forwarded to a *"/synthesizer/text_to_speak"* implementation topic is not on this messaging graph, because it was implemented as a synchronous service call (Chapter 3.1.2). This is because requesting text to be synthesized is a fast operation of just copying the string to "*/synthesizer/text_to_speak"* topic.

```
~$ rosrun chatbot chat_core
~$ rosnode info /chat_core
Node [/chat_core]
...
Services:
 * /respond
```

Figure 59. Services are shown in rosnode info

4.12 robot_prolog_connection

The *robot_prolog_connection* node is a hub that links speech recognition, the prolog runtime and the robot interface together.

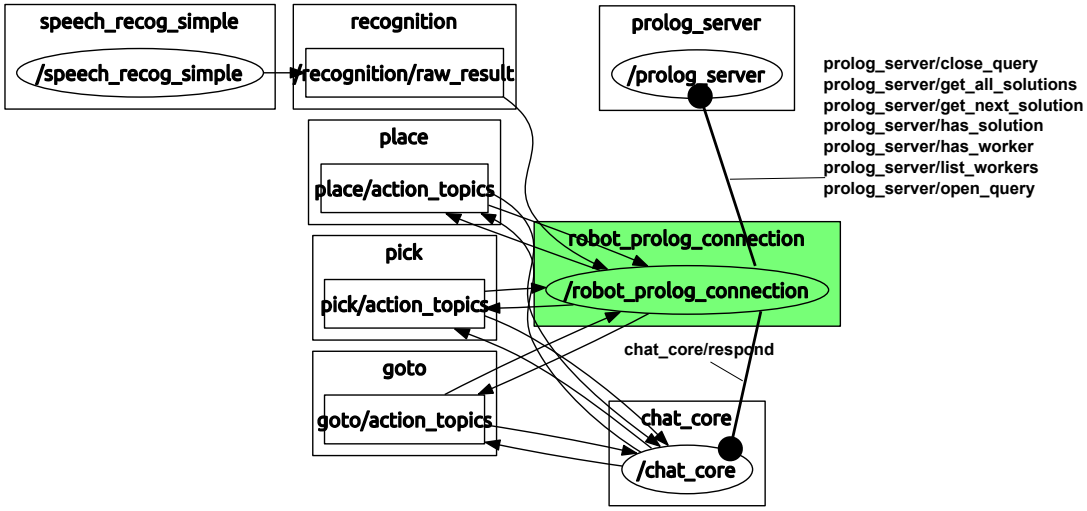

Figure 60. The *robot_prolog_connection* node

Figure 60 shows that *robot_prolog_connection* is subscribed to the *"recognition/ raw_result"* topic. This is the transcript produced by speech recognition nodes. The node forwards the transcript to the dialogue system living inside *prolog_server* node.

The *prolog_server* node is actually just an empty prolog instance on its own. *robot_prolog_connection* loads our dialogue system program into *prolog_server* on startup. After that *robot_prolog_connection* can start forwarding speech transcripts and getting reaction task responses (Chapter 2.1).

After getting the reactions, *robot_prolog_connection* prioritizes them, queues and runs them. Running the reactions involves querying *prolog_server* for the commands associated with the reaction task (Chapter 2.2) and executing those commands. The command execution happens through the implementation interface *chat_core* and is implemented via ROS actions and ROS services. *robot_prolog_connection* is also responsible for signalling the dialogue system when a command has been completed and reaction handling should proceed. Thread management and cleanup on the dialogue system is controlled by *robot_prolog_connection.* And finally failure and abortion states are handled here.

4.13 Utility nodes

This section lists additional smaller nodes used in the robot platform.

4.13.1 capture_vad_sphinx

Capture_vad_sphinx is a node that captures audio and performs voice activity detection. It was developed by the author as part of the thesis work, but was not used in the final configuration because other components implement the same feature – speex encoder has VAD and voice recognition does endpointing. It may still be useful as a stand-alone voice activity detection implementation.

4.13.2 prolog_common, prolog_msgs, prolog_serialization, prolog_swi, prolog_test, roscpp_nodewrap, roscpp_nodewrap_msgs, roscpp_nodewrap_tutorials.

Developed by Ralf Kaestner [31], these nodes wrap and interface with the prolog C++ API. They were added to the platform source because acquiring them with the package management is not possible. These nodes were updated for compatibility with our platform as part of the thesis work.

Please refer to Appendix 3 for additional configurations of the nodes described throughout Chapter 4.

5 Summary

The aim of the thesis has been to create a prototype robot platform that is capable of understanding and reacting properly to commands spoken to it in a bounded subset of natural language. Achieving the thesis' goals involved designing the system, choosing the implementation platform, creating and configuring the robot system and its subsystems, testing and optimizing the robot platform.

The resulting design is a simulated robot, with the capability of listening, responding and executing commands based on the command’s semantic interpretation. The quality of speech recognition in our robot platform is enhanced by DCG parsing and context sensitive approximation. The demo robot speaks Estonian and navigates through a simulated world. The platform was designed to be easily reconfigurable for other languages, dialogue systems, simulated robots and simulated environments.

Multiple subsystems that form a cohesive system were created as part of the thesis. Some of these subsystems were developed entirely from the beginning, while others wrap the public API of existing systems. The integrated system is complex and highly configurable. Due to compatibility and optimization issues, much of the integrated software required code-level understanding and modifications to fit into our framework. A thorough explanation of the subsystems that form the platform was provided and each subsystem's communication model and contributing functionality was described. The information provided in this thesis also serves as a source of documentation on how to utilize, run, modify, reconfigure and expand the system in the future. The results will serve for developing an interactive demonstration robot, where all the software developed in the thesis project will be ported to a physical robot platform, which is an extended version of PeopleBot [16].

References

[1] Al-Sarawi, S. Hashemi-Sakhtsari, A. McDonnell, M. Performance Evaluation of KALDI Open Source Speech Recogniser. [WWW] https://www.eleceng.adelaide.edu.au/students/wiki/projects/index.php/Projects:2015s1-06_Performance_Evaluation_of_KALDI_Open_Source_Speech_Recogniser (20.12.2016)

[2] Alumäe, T. GStreamer plugin around Kaldi's online neural network decoder. [WWW] https://github.com/alumae/gst-kaldi-nnet2-online (18.01.2016)

[3] Creating Custom World in Stage. [WWW] http://wiki.ros.org/stage/Tutorials/Creating%20Custom%20World%20in%20Stage (14.10.2016)

[4] Customizing the Stage Simulator. [WWW] http://wiki.ros.org/turtlebot_stage/Tutorials/indigo/Customizing%20the%20Stage%20Simulator (14.10.2016)

[5] Eesti Keele Instituut - Kõnesüntees. [WWW] https://www.eki.ee/heli/index.php?option=com_content&view=article&id=6&Itemid=465 (22.10.2016)

[6] GStreamer Media Types and Properties. [WWW] https://gstreamer.freedesktop.org/documentation/plugin-development/advanced/media-types.html#list-of-defined-types (14.12.2015)

[7] GStreamer plugin alsasrc. [WWW] https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-base-plugins/html/gst-plugins-base-plugins-alsasrc.html (17.12.2015)

[8] GStreamer plugin audioconvert. [WWW] https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-base-plugins/html/gst-plugins-base-plugins-audioconvert.html (09.12.2015)

[9] GStreamer plugin autoaudiosink. [WWW] https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/gst-plugins-good-plugins-autoaudiosink.html (17.12.2016)

[10] GStreamer plugin lamemp3enc. [WWW] https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-ugly-plugins/html/gst-plugins-ugly-plugins-lamemp3enc.html (14.12.2015)

[11] GStreamer. [WWW] https://en.wikipedia.org/wiki/GStreamer (22.07.2016)

[12] How to generate a Gstreamer pipeline diagram (graph). [WWW] http://developer.ridgerun.com/wiki/index.php/How_to_generate_a_Gstreamer_pipeline_diagram_(graph) (20.11.2016)

[13] Kaestner, R. A C++ implementation for using Prolog in ROS. [WWW] https://github.com/ethz-asl/ros-prolog (22.07.2016)

[14] Kaldi doc. [WWW] http://kaldi-asr.org/doc (18.01.2016)

[15] Koenig, N. ROS audio_common. [WWW] http://wiki.ros.org/audio_common (09.12.2016)

[16] PeopleBot. [WWW] http://www.mobilerobots.com/ResearchRobots/PeopleBot.aspx (29.12.2016)

[17] Pioneer Manipulator Research Platform. [WWW] http://robosklep.com/en/mobile-manipulators/148-pioneer-manipulator-research-platform.html (28.12.2016)

[18] Prolog. [WWW] https://en.wikipedia.org/wiki/Prolog (03.02.2016)

[19] Robot Caregivers. [WWW] http://web-japan.org/trends/09_sci-tech/sci100225.html (28.12.2016)

[20] ROS actionlib. [WWW] http://wiki.ros.org/actionlib (08.12.2015)

[21] ROS Communication. [WWW] http://wiki.ros.org/ROS/Patterns/Communication (08.12.2015)

[22] ROS Concepts. [WWW] http://wiki.ros.org/ROS/Concepts (23.11.2015)

[23] ROS Introduction. [WWW] http://wiki.ros.org/ROS/Introduction (23.11.2015)

[24] ROS map_server. [WWW] http://wiki.ros.org/map_server (14.10.2016)

[25] ROS nodelet. [WWW] http://wiki.ros.org/nodelet (03.12.2016)

[26] ROS rqt_graph. [WWW] http://wiki.ros.org/rqt_graph (20.11.2016)

[27] ROS rviz. [WWW] http://wiki.ros.org/rviz (14.10.2016)

[28] ROS services. [WWW] http://wiki.ros.org/Services (08.12.2015)

[29] ROS topics. [WWW] http://wiki.ros.org/Topics (08.12.2015)

[30] Rosbridge suite. [WWW] http://wiki.ros.org/rosbridge_suite (26.11.2016)

[31] RQT. [WWW] http://wiki.ros.org/rqt (14.10.2016)

[32] Running ROS across multiple machines. [WWW] http://wiki.ros.org/ROS/Tutorials/MultipleMachines (26.11.2016)

[33] Speex: A Free Codec For Free Speech. [WWW] https://speex.org/ (05.01.2016)

[34] Sperberg-McQueen, C.M. A brief introduction to definite clause grammars and definite clause translation grammars. [WWW] http://cmsmcq.com/2004/lgintro.html (2009-04-21)

[35] SWI-Prolog. [WWW] http://www.swi-prolog.org/ (03.02.2016)

[36] The Festival Speech Synthesis System. [WWW] http://www.cstr.ed.ac.uk/projects/festival/ (22.10.2016)

[37] The National Programme for Estonian Language Technology. [WWW] https://www.keeletehnoloogia.ee/en (11.11.2015)

[38] Ubar, R. Indus V. Kalmend, O. At-Speed Functional Built-In Self-Test Methodology for Processors. IASTED International Conference on Engineering and Applied Science - EAS 2012. Colombo, December 27-29, 2012. [Online] http://www.actapress.com/Abstract.aspx?paperId=454842 (17.12.2016)

[39] Ubar, R. Indus, V. Kalmend, O. Evartson, T. Functional Built-In Self-Test for Processor Cores in SoC. The 30th IEEE NORCHIP Conference, Copenhagen, Denmark, November 12-14, 2012. [Online] http://ieeexplore.ieee.org/document/6403148 (17.12.2016)

[40] Valin, J. (2007). The Speex Codec Manual. [WWW] https://speex.org/docs/manual/speex-manual/node10.html (17.12.2015)

[41] Vaughan, R. The Stage Robot Simulator. [WWW] http://rtv.github.io/Stage/ (14.10.2016)

Appendix 1 – Source code and building the platform

The source code of the natural language robot platform is located at *"https://oliver.kalmend@git.ttu.ee/thesis/msc/oliver.kalmend.git"*.

The platform can be built using the two following scripts:

"thesis/make_all.sh" – builds release flavor of the robot platform.

"thesis/make_all_debug.sh" – builds debug flavor of the robot platform and creates an eclipse project.

The following is a summary of the git repository:

Total Files:
570
Total Lines of Code:
387711 (530305 added, 142594 removed)
Total Commits:
63 (average 1.9 commits per active day, 0.2 per all days)

Figure 61. Git general information

The following image shows a timeline of the git commits:

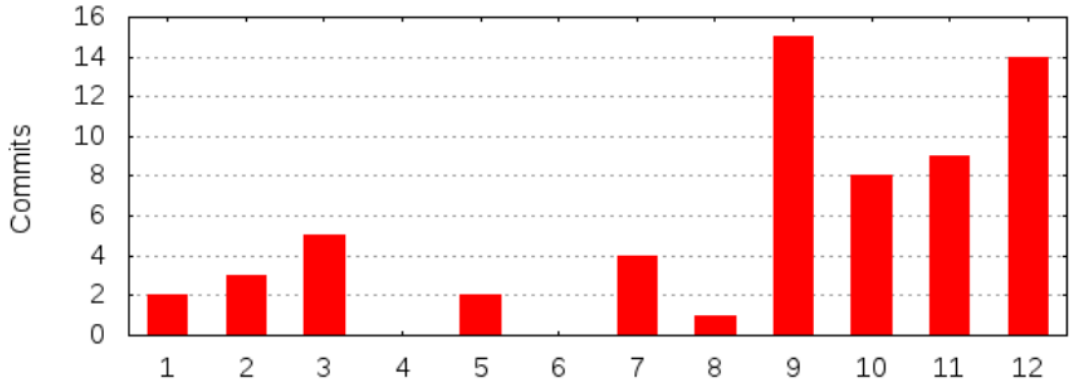

Figure 62. Commits by month

The following is a file composition breakdown of the platform's project:

Table 11. Platform files breakdown

Extension	Files (%)	Lines (%)	Lines/file
cpp	127 (22.28%)	13294 (3.43%)	104
h	119 (20.88%)	13404 (3.46%)	112
hpp	91 (15.96%)	14439 (3.72%)	158
xml	29 (5.09%)	654 (0.17%)	22
txt	27 (4.74%)	326029 (84.09%)	12075
tpp	23 (4.04%)	2038 (0.53%)	88
launch	17 (2.98%)	343 (0.09%)	20
yaml	16 (2.81%)	419 (0.11%)	26
conf	13 (2.28%)	71 (0.02%)	5
	12 (2.11%)	55 (0.01%)	4
srv	12 (2.11%)	69 (0.02%)	5
prefs	11 (1.93%)	29 (0.01%)	2
sh	6 (1.05%)	86 (0.02%)	14
cmake	4 (0.70%)	311 (0.08%)	77
msg	4 (0.70%)	28 (0.01%)	7
py	4 (0.70%)	580 (0.15%)	145
pl	3 (0.53%)	5056 (1.30%)	1685
properties	3 (0.53%)	89 (0.02%)	29
dubm	2 (0.35%)	1058 (0.27%)	529
fdt	2 (0.35%)	60 (0.02%)	30
fdx	2 (0.35%)	1544 (0.40%)	772
fnm	2 (0.35%)	1 (0.00%)	0
frq	2 (0.35%)	5803 (1.50%)	2901
gen	2 (0.35%)	0 (0.00%)	0
ie	2 (0.35%)	118302 (30.51%)	59151
libhover	2 (0.35%)	19496 (5.03%)	9748
mat	2 (0.35%)	208 (0.05%)	104
nrm	2 (0.35%)	0 (0.00%)	0
stats	2 (0.35%)	6 (0.00%)	3
tii	2 (0.35%)	7 (0.00%)	3
tis	2 (0.35%)	1908 (0.49%)	954
version	2 (0.35%)	0 (0.00%)	0
action	1 (0.18%)	9 (0.00%)	9
bak	1 (0.18%)	2 (0.00%)	2
c	1 (0.18%)	1 (0.00%)	1
inc	1 (0.18%)	27 (0.01%)	27
index	1 (0.18%)	0 (0.00%)	0
ini	1 (0.18%)	3 (0.00%)	3
json	1 (0.18%)	56 (0.01%)	56
log	1 (0.18%)	1426 (0.37%)	1426
mark	1 (0.18%)	0 (0.00%)	0
md	1 (0.18%)	49 (0.01%)	49
png	1 (0.18%)	399 (0.10%)	399

prx	1 (0.18%)	0 (0.00%)	0
resources	1 (0.18%)	18 (0.00%)	18
rviz	1 (0.18%)	590 (0.15%)	590
setup	1 (0.18%)	6 (0.00%)	6
so	1 (0.18%)	16365 (4.22%)	16365
tree	1 (0.18%)	0 (0.00%)	0
ui	1 (0.18%)	627 (0.16%)	627
world	1 (0.18%)	44 (0.01%)	44
xmi	1 (0.18%)	2055 (0.53%)	2055
zip	1 (0.18%)	0 (0.00%)	0

Appendix 2 – Future development opportunities

- Communication optimizations

ROS has a very flexible communications architecture. Communication and services between nodes are done through XML remote procedure calls over TCP/UDP. XML is flexible, but this kind of communication model can run into performance issues in real-time and near real-time systems. If communication becomes a bottleneck, a good improvement would be to implement shared memory communication or use the ROS nodelet design paradigm to get rid of copy costs [25].

- Support for other languages

The whole platform is spoken language-agnostic. This means that nothing has to be recompiled to switch to different languages. If we want to implement English for the robot, we need to switch out both the *Kaldi* speech recognition corpus and provide English *Festival* voice files - note that English comes preinstalled with some versions of festival, so we would only need to change the launch configuration file to select the correct language. The most complex file to switch out for a different language is the prolog dialogue system itself - it is currently written for Estonian grammar and dictionary only. Another possible area for improvement is the voice synthesis quality itself. There have been great leaps in voice synthesis over the past years and synthesis that is nearly indistinguishable from real speech is possible - e.g. Adobe VoCo.

- Speaker detection, directional audio and noise filtering.

Various audio processing techniques can be used to provide better transcription results. It would also be a nice add-on to be able to detect multiple speakers and generate context sensitive responses for each of them.

- Addressing the robot directly.

Currently the robot listens to all speech. If someone speaks instructions that are not directed at the robot, it still reacts. An easy fix here is to use the common method of prefixing whatever instructions you have for the robot with its name.

- Speex API

Chapter 4.6 shows the downside of using speex to transport audio between ROS nodes. Namely, the GStreamer plugin that does speech currently does not support the speex format, even though the *Kaldi* platform it is wrapping does. This introduces an extra decoding and conversion step into the pipeline if we wish to use *capture_vad_speex* with *speech_recognition_speex* instead of *speech_recognition_simple.* This can be avoided by adding speech support to the plugin generated by *gst-kaldi-nnet2-online*(Chapter 3.5).

- Small-talk or non-command based dialogue

The robot platform created in this thesis has a natural language dialogue system capable of understanding and executing specific tasks. A nice add-on would be to also be able to have small-talk with the robot. Like chatting about the weather or latest news.

- Automated end-to-end testing

Since the dialogue system we are using reduces all spoken language to pre-defined sentence types and a known lexicon, then it is possible to automatically generate tests for the robot. This sort of self-generating automatic end-to-end testing is popular in microprocessor design [39], [38], but can easily be extended to higher level concepts like the dialogue system described in this thesis. It is useful because it is future proof – new reactions, world objects and commands easily get added to the test-set. This methodology is also akin to fuzz testing.

Appendix 3 – Additional configurations of the platform

Here various possible launch configurations are described. All of these use nodes described throughout Chapter 4. These were a byproduct of developing the final solution, but they are still useful tools for testing and inspecting the platform. They showcase the flexibility of ROS and can easily be tweaked by altering the corresponding launch file.

chatbot_speex.launch

This configuration is functionally equivalent to the main launch configuration. The only difference is that the replacement described in Chapter 3.8.2 Figure 17 has been made.

test_microphone.launch

This simple configuration runs and connects the *capture_audio*(Chapter 4.1) and *play_audio*(Chapter 4.2) nodes and can be used to test if capture, audio transport and playback is working.

"*roslaunch chatbot test_microphone.launch*"

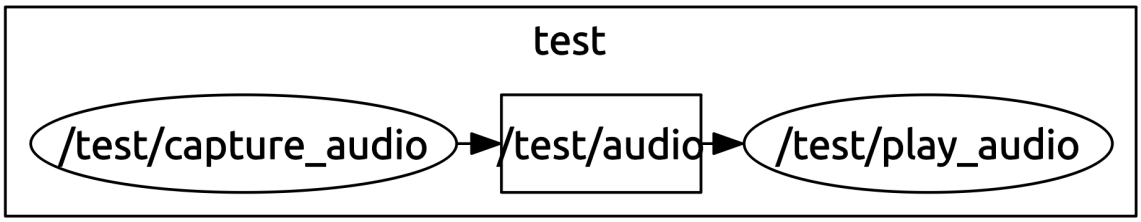

Figure 63. Launch configuration - test_microphone

test_sr_simple.launch

This configuration only launches *speech_recognition_simple* (Chapter 4.5). It is useful because it automatically sets the "*GST_PLUGIN_PATH*" environment variable and

~model_dir parameter via relative path - meaning it will work on any computer without prior sourcing or explicitly providing the *~model_dir* parameter.

"*roslaunch chatbot test_sr_simple.launch*"

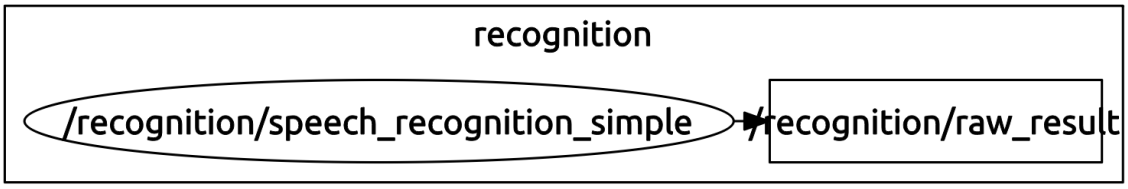

Figure 64. Launch configuration - test_sr_simple

test_sr_speex.launch

This launch configuration starts speech recognition with speex encoding. It utilizes the node *capture_vad_speex* (Chapter 4.3) and *speech_recognition_speex* (Chapter 4.6).

"*roslaunch chatbot test_sr_speex.launch*"

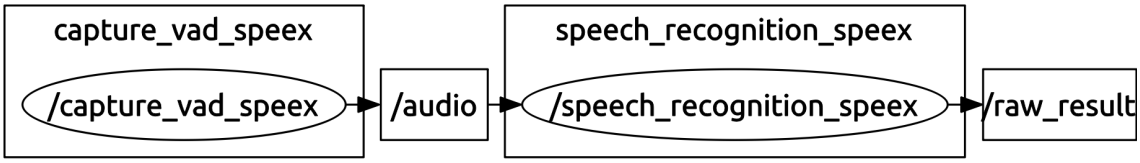

Figure 65. Launch configuration - test_sr_speex

The launch configuration also writes speech transcript to the standard output in order to quickly test speex encoded recognition. This done with the *~dst* parameter of the *speech_recognition_speex* node and is applied in the launch file as follows:

```
<node name="speech_recognition_speex" pkg="speech_recognition_speex"
type="speech_recognition_speex" output="screen">
       <param name="dst" value="stdout"/>
       <param name="model_dir" value="$(find
speech_recognition_speex)/../../deps" />
</node>
```

Figure 66. *speech_recognition_speex* node parameters in the test_sr_speex.launch file

test_vad_speex.launch

This is another simple configuration. It runs and hooks up the *capture_vad_speex*(Chapter 4.3) and *play_audio_speex*(Chapter 4.4) nodes and can be used to test if capture, speex encoding, speex decoding, audio transport, speex capabilities negotiation and playback is working.

"*roslaunch chatbot test_vad_speex.launch*"

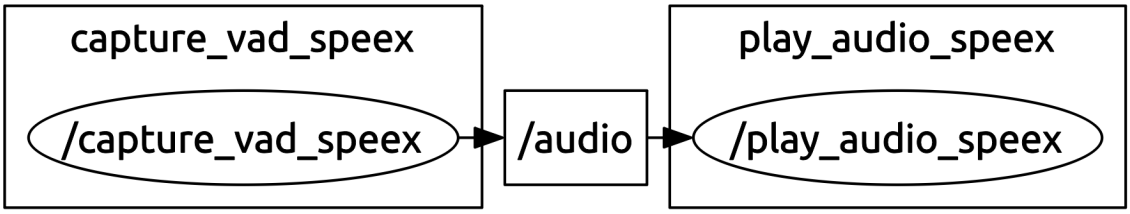

Figure 67. Launch configuration - test_vad_speex

test_vad_speex_with_perceptual_enhancement.launch

Same as *test_vad_speex.launch*, but *play_audio_speex*'s(Chapter 4.4) perceptual enhancement is enabled. This can be used to test perceptual enhancement quality of the speex codec.

test_vad_sphinx.launch

This is another test configuration that that hooks capture to playback. This one uses *capture_vad_sphinx* (Chapter 4.13.1) and can be used to test the standalone voice activity detection node.

"*roslaunch chatbot test_vad_sphinx.launch*"

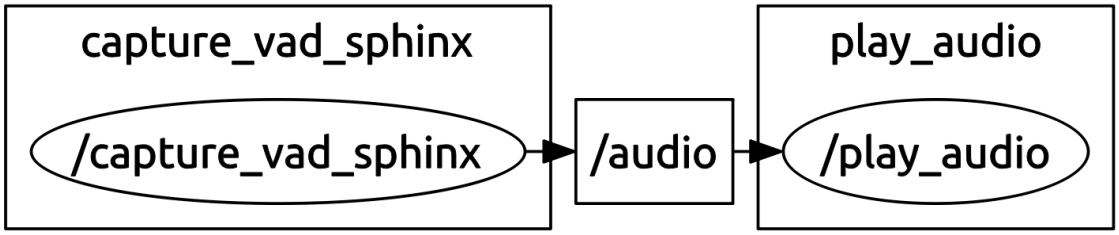

Figure 68. Launch configuration - test_vad_sphinx