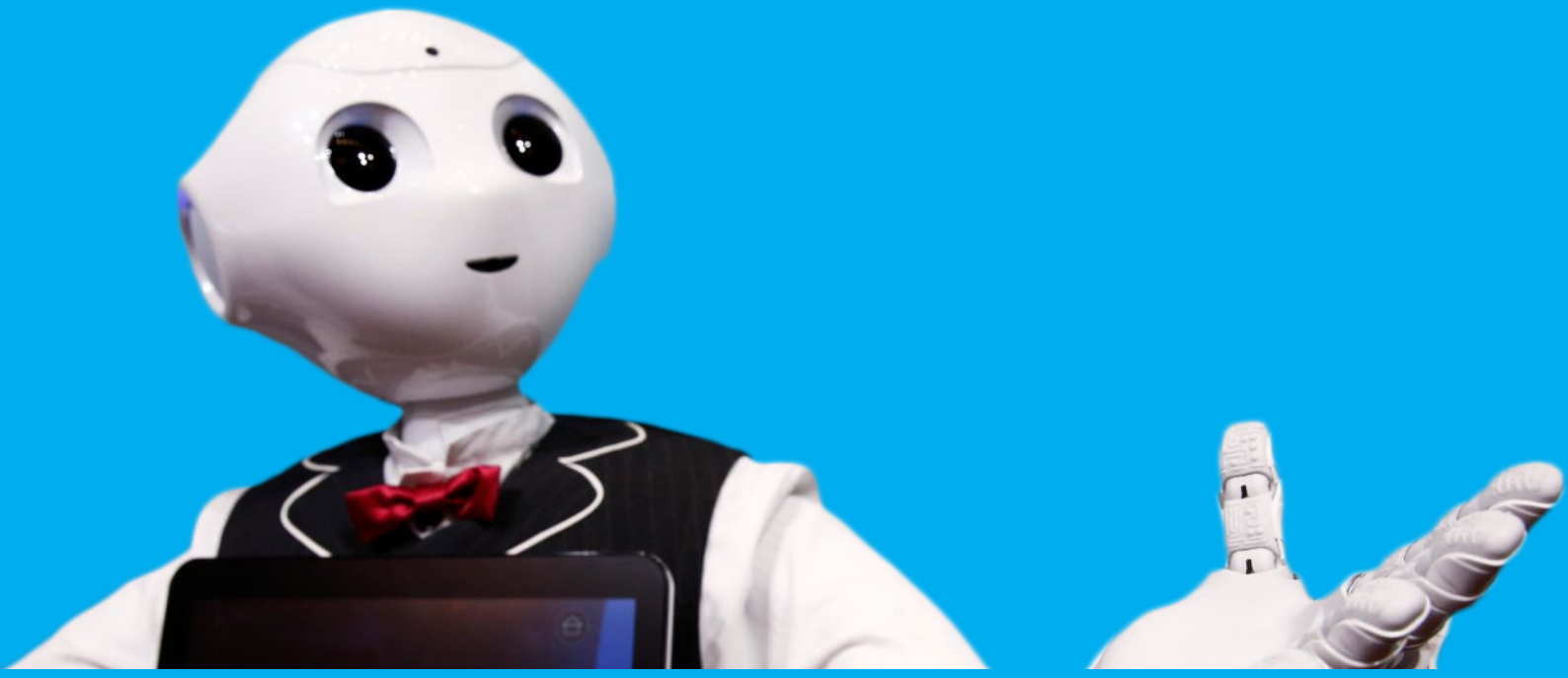
Multimodal sensory pipeline for Interactive Robots

R.S. Graafmans
M.J.E. van Osch



TU Delft

Multimodal sensory pipeline for Interactive Robots

by

R.S. Graafmans
M.J.E. van Osch

to obtain the degree of Bachelor of Science
at the Delft University of Technology,
to be defended publicly on Wednesday July 5, 2017 at 2:00 PM.

Project duration:	May 1, 2017 – July 5, 2017		
Supervisors:	Dr. J. de Greeff,	TU Delft,	Coach
	Dr. J. Broekens,	Interactive Robotics,	Client
	Dr. H. Wang,	TU Delft,	Coordinator

This thesis is confidential and cannot be made public in any form until July 5, 2019 until further notice

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.
The image used on the front page was edited ourselves but retrieved from [1].

TU Delft

Abstract

Interactive Robotics aims to familiarize children with robotics, which they try to do with their RoboTutor. Towards this goal they developed a platform for controlling the robots but this platform only worked for the Nao robot and not for the Pepper robot. So for our bachelor end project we have been asked to build a connection tunnel between the server and the Nao and Pepper robots so that it is possible to also use the Pepper robot.

As our research progressed however, it became more clear to us that in order to make a generic solution for both the robots the existing platform would not suffice. So instead of only creating the tunnel we also worked on a new server side solution. the solution we came up with, virtualization, was sure to solve the problems that caused the Pepper robot to be incompatible with the current platform. By running the server side processes for each robot inside their own environment using Docker containers we where able to create a generic solution for both robots. And with some custom code on the robots themselves, we could ensure that the robots would always connect to the server if they where connected to the Internet.

Preface

The bachelor end project is the last part of the undergraduate Computer Science program. For the project students should research and implement a solution for a real-world problem.

This bachelor end project is not the one we originally intended to do. We had some issues at the start of this project causing us to have to find an alternative project. We chose this first project because of our interests in robotics and as such working for a company called Interactive Robotics sounded very appealing to us. Luckily Interactive Robotics could find a different project for us where we would still be able to work with robots and more importantly be able to write a thesis.

Acknowledgements

We would like to offer our deepest gratitude to Joost Broekens and Joachim de Greeff. If it had not been for the the effort of Joost en Joachim to find an alternative project for us, we would not have been able to write a thesis. On top of that we would also like to thank them for their guidance and their investment in our project.

Contents

List of Tables

List of Figures

1 Introduction

In this chapter we introduce our thesis by covering the context of our project, the problem statement and our approach. We further write about the main components of our solution and the structure of this thesis.

1.1. Context

Robots are ever more present in todays society, everyone has heard about robots in factories and warehouses. Doing the repetitive and sometimes dangerous tasks humans can not or will not do or because the robots are faster than humans. Robots are becoming a more prominent piece of society and therefore it is important that people learn to live, interact and maybe even learn to program robots. This is also the idea behind the RoboTutor, a product of Interactive Robotics. They believe that children from an early age on should be exposed to robotics so they can learn the possibilities but also the impossibilities of robotics. RoboTutor learns children to work and cooperate with robots by letting them program the robots themselves.[2]

1.2. Problem Statement

We were asked by the company Interactive robotics to develop a generic sensory pipeline that would work for all Softbank Robotics their robots. While their robots are distributed over several schools across the country, the software should be able to work under all network configurations. Furthermore the robots have some specific connection demands to make sure the robots could not be controlled by a remote server.

1.3. Approach

We started this project by doing some literature research for approximately one week and a half. By then we had a clear first vision of how to create a solution for this problem, what would become the most challenging parts and what technologies would become handy to use. After the research phase we started to implement our solution step-by-step by extending our product every week to make sure the code would work after we added each layer. The first five weeks of implementation we did all on our own, while the last two weeks required some corporation with one of the employees to fully integrate our code into their systems (including specific database queries and executables for in the docker containers).

1.4. Main components

The solution to the problem does not exist as a single file or piece of code, but instead consist of separate solutions working together. Since the problem exists on multiple platforms, so should the solution. First and foremost are the robots themselves, they are required to connect to the server at startup. This should go without hiccups or other problems. They should also accept incoming commands and execute them with minimal or rather no delay. And the second component is the server, it should accept

incoming robot connections and relay them to the right client. The server should of course also accept multiple robots and clients.

1.5. Structure

The structure of this thesis is as follows: In chapter 2 the background information is found. This includes information about the specific robots and their operating system, information about technologies used, and a state of the art overview of the current robotic platforms. Chapter 3 the problem is clearly explained as well as guidelines for our solution and the ethical aspects of our problem/solution. Chapter 4 will give a clear view towards our research and development process. The design and implementation is explained in chapter 5. The evaluation of our code based on quality insurance as well as the experimentation on functionalities is described in chapter 6. The discussion can be found in chapter 7. A general summary as well as the conclusion are written down in chapter 8. Last but not least, further recommendations and future work is summarized in chapter 9.

2

Background

This chapter is about the background of different parts. Both the used technologies as well as other robotic platforms will be covered.

2.1. Technologies and concepts

2.1.1. The robots of Softbank Robotics

Interactive robotics makes use of the robots produced by Softbank Robotics, formerly known as Aldebaran[3]. Currently Softbank Robotics has two robots ready for commercial use namely the Nao and Pepper robot, a third is still in development: the Romeo. since our product had to be general and must work for all robots we examined both the Pepper and the Nao, but not the Romeo since Interactive Robotics does not use them and thus we could not test our code for the Romeo. We did keep the possibility that the Romeo might be used in the future in mind for our product.

Nao

The Nao is the first robot designed by Softbank Robotics, and was created in 2006. The Nao has since been used for a variety of tasks such as the ROBOTS show, a show created by the choreographer Blanca Li in 2013. This was the first show in the world where robots had the starring roles. There is even a theme park where a Nao is used to provide the reception and concierge services in its hotel[4], but for Interactive Robotics it is used in the classroom.

With a length of a little under 58 centimeters the Nao is by far the smallest of the robots, it is also the lightest with a mere 5.4 kilograms[5]. The Nao has a 1.6 GHz single core processor with 512KB cache and 1GB of RAM[6].

Pepper

Although the Pepper is the second robot available for sale, it is not the second robot to be designed as the Romeo was designed 5 years prior[3]. However Pepper is the first robot capable of reading human emotions and therefore it can adapt its behavior according to your mood. Pepper can even memorize personality traits and preferences thus adapting to you the more time you spent with Pepper. With its emotion reading capabilities, Pepper was designed as a true day-to-day companion. For now the Pepper is mostly used in shops to welcome and inform customers in a more amusing way, but recently the Pepper became the first humanoid robot to be adopted in Japanese homes[7].

With a length of 121 centimeters[8], the Pepper is more then twice the size of the Nao and more suitable for day to day human interaction. The Pepper is also a lot heavier than the Nao as it is 28 kilograms[9]. Hardware wise there is also a big difference, as Pepper has a quad core processor running on 1.91 GHz, 2MB Cache and 4GB of RAM[10]. In order to create a general solution, we had to keep the Nao specifications in mind.

2.1.2. NAOqi OS

The Nao and Pepper robot are both running a version of NAOqi OS which is a Linux distribution based on Gentoo. It comes with almost all the standard programs and libraries. A big difference between the Nao and the Pepper is the blocked root access on the Pepper robot while this is accessible on the Nao robot. In order to create a general solution, we should keep in mind that no additional libraries could be installed[11].

NAOqi

The NAOqi process is a broker, and runs under the NAOqi OS distribution. Upon startup it loads a file called autoload.ini where all libraries that should be loaded are defined. These libraries contain on or more modules whose methods can be accessed using the broker. Methods are attached to their respective modules, and these modules are attached to a broker forming a sort of tree. The default port the NAOqi broker listens to is 9559. We also found that without root access to the robot you can not change this port[12].

NAOqi framework

Where NAOqi is the main software used to control the robot, the NAOqi Framework is used to program the robots. This framework is cross platform, it is possible to develop with this framework on Windows, Linux and even Mac. It is also cross language, with the recommended programming being Python or C++. Both have their advantages over each other, Python is easier to learn than C++ but C++ is faster than Python[13]. Although it is possible to write software in Java and JavaScript or even with ROS, with Java and ROS it is not possible to run your code directly on the robot. JavaScript does offer the possibility to run your code on the robot directly but there is very little documentation about this.[14]

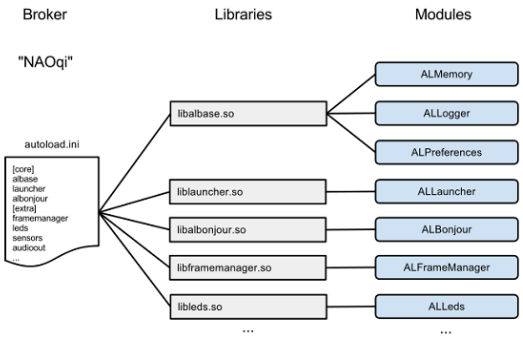


Figure 2.1: The broker uses autoload.ini to load libraries, who contain modules[13]

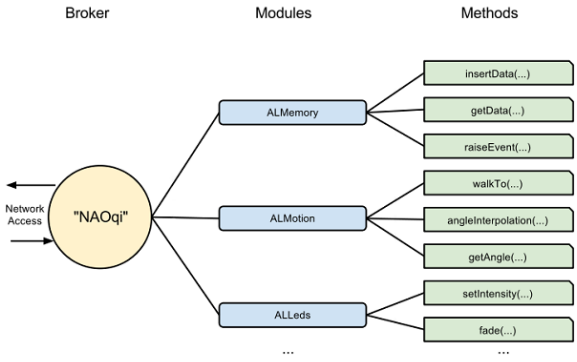


Figure 2.2: Methods are bound by modules, which can be accessed by the broker[13]

2.1.3. Basic functioning of connections over the Internet

According to the OSI-model a connection between two systems could be split into a maximum of seven layers (which depends on which protocol is used). While the OSI-model was made before there was an actual implantation, we often refer to the TCP/IP model too, which has four layers. Therefor we will focus on the TCP/IP model. (See figure 2.3)

The network access layer is responsible for the actual transmission of bits and to deliver fault-free data to the Internet layer. It does this by splitting the data in frames and controlling the speed the transmission speed by sending acknowledgment frames when packages frames are successfully received.

The Internet layer is responsible for the independent transmission of packets in a random network to the destination. (Potentially in a random order.) Therefor packet routing is an important feature in this layer. The Internet layer defines an official packet format and protocol named IP.

The transport layer is responsible for the over and forth communication between the source and destination. In order to do this the transport layer defines two protocols. The Transmission Control

Protocol (TCP) and User Datagram Protocol (UDP). The TCP protocol ensures that every data segment is received successfully by controlling the speed and resending at faulty segments while UDP just wants to transfer the data as fast as possible without the assurance that every segment is received successfully. Since we wanted to write the software for both the robot and the server to be pure Python, we had a limited choice for which protocol to use, even the Python docs mention that of the socket types only DGRAM(UDP) and STREAM(TCP) appear useful[15]. Pure Python supports both UDP and TCP but the unreliable nature of UDP made it an unfit candidate, thus we settled on TCP.

Most of he protocols from the application layer are designed for some specific tasks like HTTP, SMTP, FTP and POP3 while other protocols like SSL and DNS are used more general. In the OSI model they would split into different layers but here the are all group in a single layer [16].

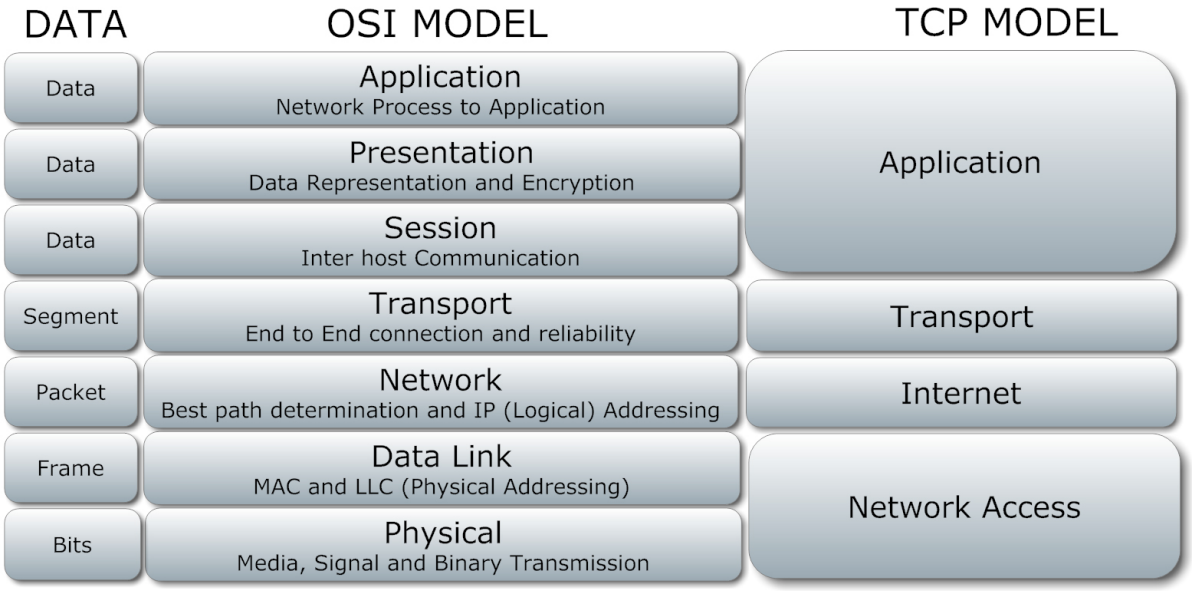


Figure 2.3: The OSI an TCP/IP model next to each other [17]

2.1.4. Network address transversal

NAT or Network Address Translation was the interim solution to the IPv4 exhaustion problem as it allows multiple devices on the same network to share a single public IP address[18]. Every device on the network or internal host has a private IP address and every packet send in and outward is rewritten by the NAT based router. For outward messages the router replaces the private IP address with its own Public IP address and maps the internal IP and port pair to an external one. When a reply message is sent, the router does a inverse translation. In order to do so, a table is maintained with the mapping of all private addresses and internal ports to public addresses and external ports. In order to add a entry to this table, a message must be sent out. If a message arrives at the router for which no mapping exist, the router does not know which device on the private network is the intended recipient and drops the message[19]. This is why a server can not reach a client behind a NAT box without the client reaching out first.

Translation methods

A NAT traversal technique is not straightforward since there is no standardized way in which the private and public IP addresses are mapped but there are four commonly used techniques.

1. Full Cone NAT. This method is the most straightforward. The private IP address and port pair, also known as an internal socket, is mapped to an Public IP address and port pair, an external socket. In this method the internal socket its port number is mapped to the external socket its same port number. All incoming traffic for which a mapping exist is forwarded to the internal; device without filtering of any kind. This is vulnerable to security risks since once a internal device

has a mapping, anyone can send messages to this device provided they know the public IP and external port of the mapping.

2. Address Restricted Cone NAT. The method here works almost the exact same way as the Full Cone NAT but filters incoming traffic. Where a Full Cone NAT accepts all incoming traffic if the sender knows the external port and IP, this method only accepts incoming traffic if the sender both knows the external port and IP and has already been addressed by the intended recipient.

3. Port Restricted Cone NAT. Not only does this method filter traffic in much the same way as Address Restricted Cone NAT, it additionally checks the port of incoming traffic. The incoming message is only accepted if the sender first received a message from the internal device it tries to contact and the source IP and port are exactly the same as the destination IP and port the internal device used.

4. Symmetric NAT. In comparison to the other methods, this method works differently. It Unlike the aforementioned methods, where the internal socket is mapped to an external socket with the same port number as the internal one if possible, this method maps the internal socket to external socket with a different port number.[18][20]

2.1.5. Traversal techniques

Because different types of NATs exist, there are also different types traversal techniques. The first one we examined that looked promising is TCP hole punching.

TCP hole punching is the technique most useful when both the client and the server who are trying to connect to each other are behind a NAT. Figure 2.4 illustrates how this process works.

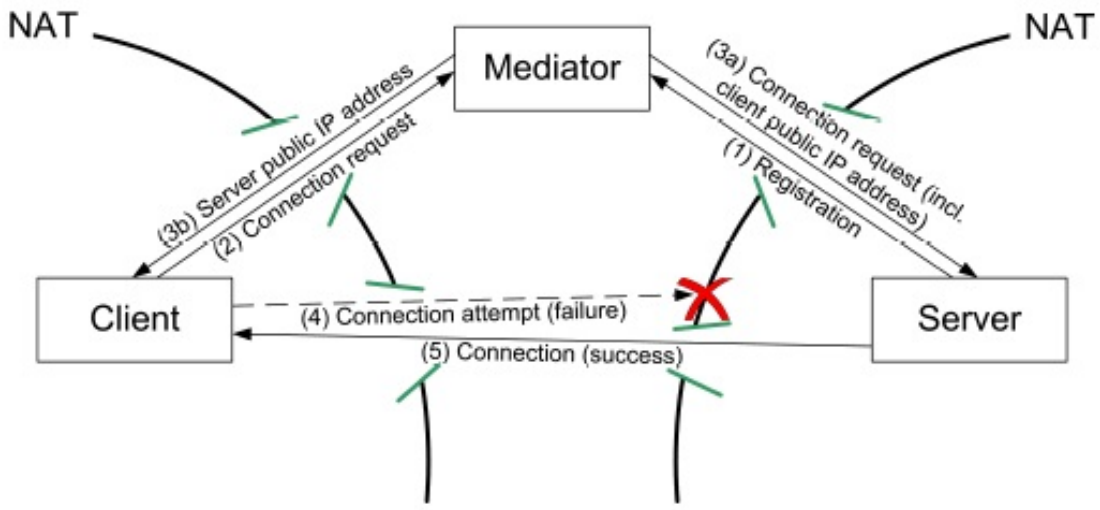


Figure 2.4: An illustration of the tcp hole punch process [21]

First the server and the client connect to a publicly accessible mediator. Because a NAT always let connection from behind the NAT to the outside through, these connections will always succeed. The mediator then sends the IP and Port of the server to the client and because client initiated contact to the mediator, the NAT in front of the client lets this message through and the same happens with the server and the mediator. The final step of this process is what earned this method its name: the client and the server both try to connect to each other but as a NAT will drop a connection if there is no mapping for it, this connection is fail. But in the act of trying to connect, both the server and the client created a mapping for each other in their respective NAT thus punching a hole through which the other can connect. This way only one connection will fail. If the client is the first to attempt to connect it will create a mapping and this attempt will fail. When the server tries to connect a little later the connection goes through because the clients NAT has the mapping for the server. The timing can be regulated by the

mediator ensuring that the client and the server do not attempt to connect at the same time, causing both attempts to fail.[18]

TCP hole punching, although promising, has one major flaw: if either the server or the client is behind a symmetric NAT the technique will not work. Because of the symmetric NAT, the port received by the mediator will not match the port in mapping created by the hole punch. So when the for example the client is behind a symmetric NAT, the mediator will receive an address with ip1 and port1 from the client and sends this to the server and receives an address with ip2 and port2 from the server and sends this to the client. Now when the client attempts to connect first, it does so on a different port than the one it used for the mediator. Thus the mapping created is between ip1:port3 and ip2:port2 and this attempt fails. When the server tries to connect to the client on ip1:port1, there is no mapping thus this attempt fails too. When the server tries to connect first it creates a mapping between ip2:port2 and ip1:port1, and this attempt also fails. Now when the client tries to connect it uses a different port for this connection than the one the server received from the mediator thus the server NAT does not have the correct mapping and the connection will fail.

Luckily the solution to the symmetric NAT problem is quite simple: it is relaying, another NAT traversal method. When the last phase of the hole punching fails we can try a different approach: instead of using the mediator for just the exchange of addresses, we can use it to relay all messages. Both the server and the client are already connected and the mediator already acts as relay by relaying the addresses it receives. By combining this solution with the hole punch solution, we should be able to handle any type of NAT. Although the relay solution works for every type of NAT, it does add complexity to the system which makes it more prone to bugs and glitches.

The last method we examined is reversal. The reason it was examined last is because reversal only works if either the server or the client is not behind a NAT or the one of the NATs supports port forwarding. Although less applicable than the previous methods, it is by far the most simple of the three. The idea behind reversal is that when for example a server not behind a NAT or a port forwarded NAT attempts to connect to a client behind a NAT, which will fail. But when that same client tries to connect to the server the connection will go through because the server is publicly accessible. So instead of the server initiating the connection with the client, the client initiates the connection. Unless the ip of the server is explicitly blocked by the clients NAT, the connection will alway succeed regardless of the network configuration in front of the client.[19]

2.1.6. Docker

Docker is a software container platform. It runs containers side-by-side on a single machine. Unlike virtual machines, docker containers do not have a full operating system. It contains only the libraries and settings that are needed to run. Therefor it does not use to much space and starts and stops in a few seconds. Docker containers are build with layers. So when there is a small difference in the last layer between two containers there is no need to build the whole container again, but only the last layer. It makes it easy to run, improve and test different setups and applications real quick.[22]

2.1.7. MongoDB

MongoDB is an open-source, cross-platform, document orientated, relational NOSQL database build for scalability and good performance. It uses JSON-like files to store the data. By using load balance, primary and secondary indexes and the use of MapReduce algorithms it can process much information fast and reliable.[23]

2.2. Related robotic platforms

We found very little about existing robot platforms, most likely due to the companies behind them not wanting to share how they did it. we also found that there was no existing solution for the problem Interactive Robotics has with their platform.

2.2.1. ROS

Robotic Operating System (ROS) is an open source robot platform which makes the development of programs much easier due to its libraries. It supports many different robots/parts. Every robot has its own project. This is why ROS not always works the same on different robots and not all robots has the same functionality. It highly depends on programmers with some spare time which could make the software for some robots a little bit outdated. It allows to setup web based sockets to easily connect to the robot to the cloud in order to share information[24][25][26]. The ROS package for the Nao an Pepper creates a direct bridge between ROS and the NAOqi client.

2.2.2. RoboEarth

RoboEarth is an open source database with information about many robots. A robot could communicate with the cloud. It can send all the information they have after which the cloud will compute the actions the robot should do in order to complete a tasks. Because the calculations in the cloud generates general tasks and the interpretation is done on the robot itself, it is easier to use on a custom made robot. RoboEarth already support different robots with different goals and is still growing[25][27].

2.2.3. Rapyuta

Rapyuta is a RoboEarth spin-off that aims to build a multi-robotics platform for the security market. It supports mainly robots with security tasks and allow them to do very heavy computations in a secured cloud.[27][28].

3

Problem Analysis

In this chapter we will dive in the problem description and get a better picture of it. This is done by extracting the main difficulties, writing user stories and create a MoSCoW chart.

3.1. Problem definition

Since the robot industry is growing and robots become more integrated in our daily life, the wish to let children discover and grow up with them is rising too. The interactive robotics company responds to this demands by facilitating a platform which make it possible to easily discover, play and program robots. Their platform works only with the Nao robot from Softbank Robotics at this time, since this is the only robot where root access has been granted. In the nearby future they want to make it possible that the pepper robot (and romeo too) can also be used within their platform. In order to make a general solution it is important not to focus on which data is passed from the server to the robot but how. When both ends of the pipeline behave correctly, the robot should work as if there is a direct connection. Furthermore the solution should be integrated within their work flow. Although this requires some small changes, most of it can be done by running the current processes next to each other virtually.

3.2. Research questions

At the beginning of the project, after some discussions with our client about what he wanted we came up with a research question: what are the requirements to establish and maintain a successful connection between the robot and the server? After some research and development however, we realized this was not the right question. Since we were expected to both research the problem and come up with a solution, we felt this question did not accurately reflect our activities and created a new question:
how do we create a generic cloud platform that works for both the Nao and the Pepper?
To answer this question, we formulated some sub questions we needed to answer first:

Why can the current solution not be adapted to work with the Pepper robot?

How do you ensure that the robot connects with the server regardless of network configuration?

How do you ensure the robustness of the connection?

3.3. User stories

As a teacher, I can turn on the robot, so that the robot starts up and automatically connect to the server.

As a teacher, I can start a script from the website, so that the robot will start to executing this within a few seconds.

As a teacher, I want to be sure that when my WiFi falters, the robot would reconnect without any action.

As an employee, I can launch the script for the first time after which I can see the robot in the database.

As an employee, I can debug any problem using the log files.

As an employee, I can ask the server for some detailed information about the status of all the robots.

3.4. Requirements

The requirements are prioritized using the MoSCoW method.[29] **M** stand for Must have. **S** stands for Should have. **C**stands for could have. **W** stands for Would like to have.

M: 1. Connection must be able to be made from every place with Internet access

M: 2. The server must be able to handle multiple connections at the same time

M: 3. The server must pass data without discrimination

M: 4. Connection must be reliable

M: 5. Manager must map server side connections to the right robot

S: 6. There should be low or no latency

S: 7. The server should run every server side connection in an own environment

S: 8. Our application should automatically startup in the boot sequence

C: 9. There could be encrypted connection

C: 10. Client could be able to change the robot name through company's website

W: 11. Fully integrate the companies robot web platform with our solution

4

Research and development process

In this chapter our way of researching before and during the project is presented. Thereafter the development process will be explained based on different parts.

4.1. Research

4.1.1. Start of the project

At the beginning of the project we did a lot research on different aspects. all of our results are already presented in chapter 2. During this research phase, we concluded that although we had found some useful information, we knew too little about the obstacles we would probably face. Therefor we decided after one week and a half to start developing a first stage and to do some specific research for everything we would face.

4.1.2. During the project

When we started developing we faced some issues where further research was needed. Especially about the virtual environment and when we started integrating out system. The results of our research we deemed background information are found in chapter 2, other problems we encountered can be found in chapter 7

4.2. Development

4.2.1. Methodology

We used agile software development during this project, but not scrum. Although scrum works great and could definitely be useful for this project, it is recommended to have more than three people in the team as there are three type of jobs to be divided in a scrum team. Since our team consists of only two team members we sought a different methodology, and we found ASD: Adaptive Software Development.

ASD consists of three phases namely Speculate, Collaborate, Learn. Speculate is called speculate for a reason as the word planning is too restrictive. When you plan something and you deviate it is often seen as failure, which in software development happens quite a lot. Whether you misinterpret client needs or requirements simply change, it is common to not follow the plan exactly to the letter. This is why the speculate was offered instead of planning. With speculate, it is not that you do not plan your work or just lay down some loose guidelines, it just admits that in the grand scheme of the project some assumptions or objectives you thought were required are most likely wrong and that is okay. ASD proposes to just speculate about the general idea about the direction in which the project should go and then adapt where needed. An overview of our speculations is found in Appendix C

Collaborate, to quote the book Adaptive Software Development[30] page 45 on the definition of collaboration by Michael Schrage: collaboration is ”an act of shared creation and/or shared discovery”. Collaboration requires active participation, with the participants having the intend of actually adding

value. Just communication between participants, although still required for good teamwork, is considered passive with only the intent to inform.

Learn, as the name suggests means gathering feedback and learning from this feedback. in the collaboration phase, the product is build or extended upon. In the learning phase, feedback on the value of these products is obtained by exposing them to stakeholders. Other ways of more accurate feedback are beta testing for example, as it is the customer who will be using the product. The learning phase is about examining mistakes, or assumptions that turned out to be correct, made in the previous cycle and adapting the results to the next.

Our cycles were usually a week long with a demo at the end of the cycle. The reason the length of these cycles fluctuated was because our coaches were also quite busy during this project or we would finish the cycle sooner or later than expected. After the demo we would lay out and discuss the plans for the next cycle, were we sometimes would have to fix some mistakes or wrong assumptions from the previous cycle first and sometimes we could start working on new features right away. [31][30]

4.2.2. Programming languages

This project is almost entirely written in Python, with only a few lines of code in shell script and some lines of plain text written in a specific format for Dockerfiles. NAOqi supports both multiple languages but the most documentation can be found for Python and C++. We chose Python for this reason but also because we where more familiar with Python than C++, and since the NAOqi OS is based of Linux we expected very little problems with using Python. Python was also recommended by Softbank Robotics for beginners developing in the NAOqi Framework as it easier to learn than C++ and quote: ”should fit all their needs”.[13] Since we could not count on the possibility of importing external libraries on the robot, the project is almost completely pure Python with two exceptions being Docker-py and pymongo. Docker-py had to be used for starting and stopping the Docker containers in which the code for controlling the robot is run. And pymongo is used for interacting with the database. Since the code using Docker-py and pymongo is run server side, there was no issue with using external libraries. The reason we used some shell script was because of Docker and Docker-py. Some issues could be resolved more elegant with a few lines of shell script and calling it with Python than using Docker-py for the task.

4.2.3. Testing

Testing of the code was hard because many parts contains code for specific socket behaviour. Since the only good way to test this was mocking it was hard to test most of the methods because many of them only contains network code. Only the auxiliary classes (which comes back in chapter 5) are fully tested. Still the total test coverage is more than 50%.

4.2.4. Tools

The tools that we used to develop our software are described below. The tools are also grouped by category in Table 4.1

Choregraphe is a visual programming environment which also allows the user to visualize everything the robot does. We used it for testing with a virtual pepper and to make the final module that has been made for the Nao or Pepper.

Docker is a software container platform. In the project Docker is used to run every NAOqi runtime in a virtual environment since the Nao and Pepper robot requires the connection to communicate all over the same port and IP.

Filezilla is a file transfer tool which makes used of the (s)ftp protocol. We used it to transfer files to the server and robot.

Git is a file version control system which we used to easily work together on the same code. Although there is another major version control system (SVN) we chose git because we used it before and SVN is not supported by Github.

Github is a hosted git repository service. We chose Github because it has free private repositories for students and because we worked with Github before.

Gitlab is an opensource git service that need to be hosted on an own server. During the project we only used Github but in the end our code has been moved to the Gitlab of the Interactive Robotics company.

Kitematic is a visual environment to manage the Docker containers. We used it in the beginning to easily see, manage and control the containers.

Travis-CI is a continues integration service which runs the tests code and checks if there are no errors. Travis checks the code on every commit. We used it to make sure no faulty code would been pushed to the master branch.

MongoDB is a relational NOSQL database which is fast and scalable. It uses JSON like files to store the data. We used it to integrate the manager in the system. It reads the scenarios and robots from the database so its data is always up to date.

NAOqi bin is the core of a virtual robot. It launches all the core modules with the exception of visual and audio modules and make it possible to test your code.

Pycharm is Python IDE with support for different plugins. We used Pycharm for the main coding and testing work as well as a tool for checking the test coverage and code duplicity.

Putty is a SSH and telnet client. We used it to remotely control the server and robot by executing commands through putty.

Sublime text is a text editor. It supports many files and is used to create, among other things, the shell scripts and the Dockerfiles.

Wireshark is a network analysis tool that makes it possible to analyze the communication with a device. We used it to analyze how the handshakes and flags were send between NAOqi and the robot. Besides the handshake we found the reason why all connections were blocked by inspecting the packages that were sent.

Category	Tool	Website
Code management	Github	`github.com`
	Gitlab	`about.gitlab.com`
Deployment	Docker	`Docker.com`
	Kitematic	`kitematic.com`
Development	Travis CI	`travis-ci.com`
Editor	Pycharm	`jetbrains.com/pycharm`
	Sublime Text	`sublimetext.com`
Network analysis	Wireshark	`wireshark.org`
NOSQL Database	MongoDB	`mongodb.com`
Remote Connection	Putty	`putty.org`
	Filezilla	`filezilla-project.org`
Robot control software	Choregraphe	`doc.SoftbankRobotics.com/2-4/software/choregraphe`
	NAOqi bin	`doc.SoftbankRobotics.com/2-1/dev/tools/robot-simulation.html`

Table 4.1: Table overview of the tools used to develop our solution

5

Design and implementation

This chapter is about the design and implementation of our pipeline software. First the basis requirements will be explained. Thereafter the three main parts of the code will be enlightened. These are: the robot side, the server side and the auxiliary files.

5.1. Pipeline

For the pipeline there are some general specifications and requirements upon which the system was designed.

The robot always initialize the connection to the server in order to work no matter the NAT it is behind.

The only connection that is always active is between the robot and the manager or relay on the server.

The server should ask the robot for a new tunnel and never builds it up by itself.

Every tunnel should be closed after it is not necessary anymore.

Docker containers should only run when the user gives the command to do so.

It should be possible to easily change settings.

Every process that contains blocking calls should be threaded.

When a connection goes down while running it should reestablish itself when the Internet connection is restored.

The NAOqi process on the robots accepts only connections from port 9559. In order to control multiple robots at the same time, some virtualization technique should be used.

How the main parts implement those requirements is explained below.
A Terminology diagram can be found in Appendix D
A flowchart of how the main parts of the code connect and how a pipeline is established can be found in Appendix E.

5.2. Robot side

The robot side consist of one main system and a sub system. The sub system only asks data from the robots memory. For example the name, id and type (which is Nao or Pepper) can be asked which uses the main system to let the server identify it as a specific robot. The main part is the robot connector. When the script launches, it will try to connect to the server. Without a specific port it will connect on the main entry port of the server. When it connects successfully it will maintain this connection until

the server tells the robot connector to reconnect on another port. This happens after robot sends it personal data on request of the server. Once connected to a specific unique port the manager socket will handle request for new socket connects so that a tunnel will always be started from the robot side. Doing it this way ensures that the connection would not be blocked by the routers firewall because of a strict NAT. The robot connector starts two new connections (one with the server and one with NAOqi) and forwards the messages to each other. Because of bidirectional mapping of the sockets, data cannot be forward over a wrong connection.

5.3. Server side

Server side can be divided into three parts: connection management part, the Docker container part and the auxiliary part. The connection management part is run on the server and launches Docker containers when asked to and connects the incoming robot connections with the right client. The Docker container part is the software run inside the containers and connects these containers to the connection management part. The auxiliary part contains helper files only needed by the server. Although there is already an auxiliary classes part, those classes are needed by both the robot and the server.

5.3.1. Connection management

The connection management part consists of a main system: the manager, and two subsystems: the web manager and the relay. The manager is responsible for handling all the robot connections. When a robot connects to the server, the manager first asks the robot to identify itself, which it does by sending back its personal data. The manager then checks if this robot exists in the database and adds them if they do not exist. Being automatically added to the database does not mean anyone with a robot can use this service. This is because although the robot is now in the database, it is not coupled to a school and therefore unable to receive commands from the server as only schools with an account can issue commands to the robots coupled to that specific school. Once a robot is verified and if needed added to the list, the manager checks if there already exists a relay for that robot because it is not very useful to have more than one relay for any given robot. If no relay exists the manager creates one and provides the ID and IP of the robot to the relay so that only one robot can connect to a relay. After this check the manager sends the port of the relay to the robot so the robot can connect with the relay. Furthermore is the manager also responsible for garbage collection, it keeps track of all the relays and containers and checks them periodically to see if they are still running. If not, the manager removes and deletes them and otherwise it leaves them be. This garbage collection frees up ports, ram, and space on the server by only using what is needed when it is needed instead of using what might be needed all the time.

The web manager is not a standalone class, it is instead constructed and utilized by the manager. The manager was required to be able to receive commands from the web server from Interactive Robotics and when this functionality was added to the manager it became quite large, thus the web manager was created. The web manager is responsible for handling all commands received from the web server and carrying them out. When the web manager receives a command it first checks if the command is known. If it is and all arguments are supplied and valid then the command is carried out. If not all arguments are supplied or they are not all valid, then the web manager sends back an error letting the web server know something went wrong. If the command is not even known the web manager responds to the web server that it does not know this command but does not take any further actions. The web manager accepts 6 commands at the moment of writing: 'launch', 'container', 'online', 'exist', 'kill', and 'status'. The launch command launches a container for a specified robot. It first checks if the Docker image requested even exist and if so, if it is not already running. If all checks are passed it launches a container and provides it with the address of the relay of the robot the container was launched for. The container command checks if a container is launched for a specified robot and if so, what is the status of that container. The online command has actually two functionalities depending on the arguments supplied. If a robot is specified, the online command checks if the robot exists and if it exists it also checks if this robot is online. The web manager then returns this information back to the web server. If however no robot is specified, this command lists all robots currently online and connected to the server. The exist command requires no arguments because if it would check only one robot, it would do the same as the online command but without checking if it is online or not. So the exist command lists all robots registered in the database. The kill command kills the container of the specified robot. It first

checks if the robot is even online and if it has a container running. If both checks are passed it forces the container to stop and then removes it. The status command requires no arguments just like exist, when the web manager receives this command it list all online robots and the relay they are connected to and all running containers and the relays they are connected to.

The relay is like the web manager no standalone class. Unlike the web manager however, which is only constructed once, a relay instance is constructed for every distinct robot that connects to the manager. The relay is mainly responsible for relaying all the data it receives from the robot to the right container and from the container to the right robot. If the relay does not receive any data or connection requests for thirty seconds, it sends out a heartbeat over all its connections. This it to detect if any connection was lost and is mainly quality control. When a robot disconnects, the relay will wait for up to ten minutes before shutting itself down. The disconnection can have any cause such as a faltering Internet connection or simply because the robot was turned off, making this timeout necessary. If a robot reconnects within these ten minutes to the relay then the relay accepts this connection and continues running as if nothing has happened. If the robot reconnects to the manager within these ten minutes then the manager forwards the robot to the relay, else it starts a new relay because the old relay has already been removed.

5.3.2. Docker container

The Docker container part is actually only one file: server_connector. This file is loaded into the Docker container when it is launched and starts up as soon as the container is ready. The server connector is responsible for connecting the container with the connection management part, but does not do so until it receives data itself. This is to prevent that ports are unnecessarily open and saves effort in port managing. Furthermore, when a Docker container is launched it needs to retrieve a scenario from the database. Since the Dockerfile splits on commas, another solution was needed. The server connector automatically starts when the container is launched and by passing the parameters for the scenario to the server connector, it can retrieve this scenario from the database and write it away inside the container without splitting on commas.

5.3.3. Auxiliary server files

There are two auxiliary files in server side: the MongoDB class and Docker_build.sh. As mentioned in subsection 4.2.2 we used some shell script as it was a more elegant solution than using Docker-py. Docker_build.sh is used by the manager to build the images that are required but not yet build. Because building these images can take quite a while, it should be done at startup and the manager should have build all the required images before starting up its mainloop.

The MongoDB class is basically a general connector with some basic queries packed in methods. It uses the PyMongo API to make a connection and execute all the queries. The queries that are used are: get specific robot, get all robots, get a specific robot field, add a robot, get all scenarios, get a specific scenario and get a specific scenario field.

5.4. Auxiliary module

The auxiliary module, in the software itself referred to as helpers, are all the files needed by both the server and the robot. The auxiliary module consist of five files: the config, the connector, the logger, the parser, and the pathfinder. These files are all helpers of the main files and with the exception of connector have no class instances of themselves.

The config file has no methods, but instead contains all settings needed to be easy to change for all files, settings such as server port and ip for example, since many files used these settings this made it easier to make a virtual robot connect to a local run manager instead of a manager run on the server. The config also contains three shared variables. These were needed by both the manager and the web manager to read and write these variables. Because the web manager is not threaded we thought it was cleaner to contain these variables in an external file.

The connector is the only file in the auxiliary module that has a class instance. This is because this

class instantiates both server sockets and client sockets. By constructing an instance of this class and calling the right method for a server or client we can immediately create a socket and let it listen on a specific port or connect to a specific address in one line of code.

At first the logger was created and its variables set in every single class which is bad for code duplicity. This was because we use Logging, which is a standard Python library. Logging accepted only a name as argument for its create method but we needed different variables based on the file for which the logger was created and a different log file per class that was logged. So the the logger was created. In the loggers create logger method the name supplied is also automatically used as the name for the log file instead of having to set it manually. It also has a optional argument for to specify if a class is threaded or not, making a different format needed to also include the thread name. All these settings had to be set manually with just the logging library, but by placing it in a separate file it could be done with a single call and the right arguments.

The parsers sole purpose is to make it possible for files that use the parser to provide arguments in the command to run that file. Although the parser has default arguments configurable in the config file, by providing the server connector for example with the port of the relay it is supposed to connect with it is possible to dynamically set up the server connector. This in turn makes port guessing harder and thus the software more secure.

The pathfinder is mainly used by the logger to find the path where it needs to create the log file. By using system calls the pathfinder is cross platform. This feature was actually rather important because the development on Windows, but the end product had to run on Linux. The pathfinder has 3 methods: one to find the directory of file specified, one to find the parent directory of the file specified, and one to find the grandparent directory of the file specified. It is also possible to not specify a file for all methods, in which case it uses the running script invoking the call as the file specified.

6

Evaluation

In this chapter we write about our quality assurance and success measurement. We cover our feedback from the Software Improved Group and some experiments we have done.

6.1. Quality assurance

To ensure the quality and functionality of our code we tested it rigorously by using it in real life scenarios. This way we were sure the code behaved as it should and were it did not we could patch it immediately. We also wrote unit test for our code but in our case, which is quite specific, tests were difficult because of the nature of our code. Testing threaded classes in Python with unit test is not easily done and testing socket based code is even more troublesome as socket calls are blocking. Our software contained both, this is why we placed our focus more on real life testing. Some further explanation can be found in subsection 7.1.3. Another way of quality insurance is SIG, the Software Improvement Group, which evaluated our code once midway the project and once on the day of the deadline of this report so the second feedback is not contained is this report.

6.1.1. Software improvement group

SIG evaluates your code and gives you a score between one and five stars and feedback about your code. This feedback is very useful because it often finds things you did not see or realize. When our code was analyzed we scored a three out of five on their maintainability model. This means our code was average maintainable. The reason we did not receive a higher score is because our score is lowered by the lower scores for Unit Size and code duplicity. (See Appendix A)
The lower Unit Size score was based on the fact that a few methods should have been split to increase the code understanding. This has been taken into account and has been implemented were useful. The score on duplicity was mainly based on the similarities between the server and robot connector. Although there are a few methods with common lines of code it is hard to group them in a parent class without including many *if else* cases to make it work for both cases.

6.2. Success measurement

Since our client also want proof of concept integration, we measured our success with functionality. We tested the functionality of our software by uploading it to a real robot and running scenarios as if the robot was in the classroom.

6.2.1. Experimentation

For every additional layer of functionalities we did some extensive, real life, test. We tested if the core functionalities still worked without the introduction of some unexpected or retarding behavior. For the tests we ran the robot connector code on both the Nao and Pepper. All the server side files were executed on a real server. The tests consist of five steps. First we tested if the robot was able to connect to the server behind a complex network setup. The second step was to test if connection drops were handled well in every situation. The third action to test was to execute a robot behavior code at the

beginning of the tunnel and see if the robot does execute this behavior correctly. We did this multiple time at different intervals between them. The fourth step was to test if the web server commands were executed well and the containers were build and destroyed correctly. The last step was to check whether the ports were released after the connection was closed or dropped.

7

Discussion

In this chapter we review some unexpected problems, furthermore we talk about some ethical questions raised during the project.

7.1. Encountered problems

During the project we faced some problems which lead to complications for the general solution. The problems and our solutions for them are described below.

7.1.1. Specific IP and Port

One of the main problems was the requirement to connect to a specific port. The robots require to make a connection on port 9559. Furthermore NAOqi sends the IP and port it expects a reply on inside every message. When the address of the incoming connection is different than the address in the message, because we had build a connector to initialize the connection with the server, NAOqi sends to and expects replies from 127.0.0.1:9559. Since this was not the case as we retransmitted the messages, the robot will drop its connection and try to reconnect to the address it expected a reply from. We figured this out because all the data send in the messages is in plain text, when we used Wireshark to check whether the packets we send would arrive or not, we stumbled upon it. We solved this problem by making use of the three main parts. The input stream connects to the server_connector on 127.0.0.1:9559, the server_connector makes a connection to the relay. The relay forwards this to the robot_connector and this part send the message to 127.0.0.1:9559 . This way the robot thinks it has a direct connection and it would not disconnect.

7.1.2. Autostart the connector

Without root access it is harder to make programs start at boot. NAOqi OS has not all the basic system libraries installed. For example the crontab is not available and cannot be installed without root access. The only way to make our robot_connector startup automatically is by adding it to the autostart.ini from the NAOqi. In order to fix this solution we created package through Choregraphe. The disadvantage of this solution is that the Python files should be imported in choregraphe and then will be pushed to the robot. When a change in the code should be applied on the robot, the package should import the code again and then e updated on the robot. Until there is a better solution this could be a workaround.

7.1.3. Code testing

Unit testing in Python is hard for some of our classes. Especially threaded classes and those using the socket library. Sockets make the code hard to test because of two main reasons. First is that most calls from/to a socket are blocking. In the second place since it is impossible to mock one only one connection. This makes it hard to test specific elements of the code without mocking everything. This basically replaced a whole method and made the test useless. The problem with threads is with the standard Python framework. the Python unit test library is not aware of threads in test classes[32]. Although there are some thread testing libraries they are not perfect. Especially in combination with sockets that should send in a synchronous way without blocking calls in order to test it, it is really hard.

Therefor we chose not focus on the maximum code coverage but testing as much as possible within the limits of reasonable effort.

7.1.4. Docker comma split

Inside the Docker containers scripts that dictates the behavior of the robot depending on the provided script runs. The provided scripts are stored in the database. In the early versions the containers took the script as an input and stored them into files inside the container. This was working until commas appeared inside some scripts. Apparently Docker containers split their input on commas and the provided scripts were broken by them. We solved this by providing the script id instead of the script self. When launching the container the server_connector script inside takes the id, pulls the code from the database and writes it away.

7.1.5. Integration into the current system

The current platform uses Java to run the GOAL code files. In oder to control many robots, this part of the system should be ran inside a container. In comparison with Python the Java library is quite big and consumes a lot more RAM. For now this would not cause any trouble but it lowers the scalability of the system.

7.1.6. Port management on Linux

In comparison to Windows, Linux leaves ports open after script termination. This could lead to incorrect behavior when for example due to further developing, error occurs that kills the program. Before restarting the server, one should kill all Python open Python processes in order to make sure all the connections are closed.

7.2. Ethics

Ethics is an important aspect in the IT, and this project is no exception. A few questions about ethics were raised while developing the software. The first question was raised early on in the project: is it not ethically wrong for Softbank Robotics to decline clients full access to their products? Although this in only the case with the Pepper robot, if this issue did not exist then there would be no need for this project in the first place. The port on which the robots communicate can only be changed with root access and you do not have this with the Pepper robot. Add this to the fact that Softbank Robotics implemented some requirements for the connection between a robot and a computer, they make it much more difficult for third party developers to create a cloud software solution for their robots. To the best of our knowledge Softbank Robotics does not have their own cloud platform, so why they implemented these restrictions is not clear to us. Because of theses restrictions a second question was raised: is it unethical for us to use data found with deep packet inspection? At first we had some connection issues, so we used Wireshark in order to test if the packets reached their destination. We did not intend to seek out these restrictions but since all the data was transmitted in plain text, we stumbled upon them. With this newfound data we were able to comply to the restrictions imposed by Softbank and find a solution for the problem.

8

Summary and conclusions

In this chapter we present a summary and the conclusions of our work. Here we answer the research questions we proposed.

8.1. Summary

In this thesis we have presented a solution for the sensory pipeline problem. We covered quite a bit of background we researched in order to create this solution and covered the structure and functionality of our code.

Our project started a bit later than most of the other groups because of some communication failure at the start of this period. The project we intended to do, which was also for Interactive Robotics, fell through and so we required another project assignment. Because of the effort of Joost and Joachim who came up with another assignment for us we were able to start, albeit a week later. This also limited the time for a research phase so we decided to do a preliminary research phase to investigate everything we needed to know to start the project and then research more where needed. During this first research phase we mainly focused on networking and already existing solutions.

Since we evaluated our product on the basis of how functional it was we started working on a very basic product with little features and worked on extending it from there. When we had a working product we showed it to Joost and Joachim who then pointed out what we missed, what they would like to see in the next iteration, and what pleased or pleasantly surprised them. This way the product evolved with their preferences and requirements during the development.

The sensory pipeline was build from scratch and with good reason. By building it from the ground up and using nothing but pure Python we had full control of which direction the product could go, and because the server of Interactive Robotics was also being overhauled we could also adapt to changing requirements more easily because our product did not depend on anything.

To verify our product actually worked we tested it on a real robot. Most of our test where on virtual robots but these virtual robots did not have all the functionalities that the real robots have, so for a better and more real test we used the robots for which the software was designed. this enabled us to get a better look at what went wrong, well, or what functionality was lacking or redundant.

8.2. Conclusions

The research subquestions as presented in section 3.2 can now be answered as follows:

Why can the current solution not be adapted to work with the Pepper robot?
The current solution of Interactive Robotics actually *can* be adapted to work with the Pepper robot, but it would just not be feasible. In one of our first prototypes we succeeded in connecting the Pepper with

the server, but since we can not change the port Pepper listens on we would need a different server per Pepper. The reason Interactive Robotics succeeded with the Nao is because on the Nao you can have root access and thus change the port it listens on.

How do you ensure that the robot connects with the server regardless of network configuration?
In subsection 2.1.5 we examined the most promising NAT traversal techniques for our problem and discovered that by combining *hole punching* and *relaying* we would almost always succeed in creating a connection. But since those two solutions expect both the server and the client to be behind a NAT, the solution is much simpler for our problem. The fact that the server Interactive Robotics uses is publicly accessible makes it possible for us to make use of the *reversal* technique. When the reversal technique is possible the connection will always succeed regardless of network configuration.

How do you ensure the robustness of the connection?
The robustness is ensured in two different ways. First of all by using TCP it ensures that every package is delivered. When there is a package loss the package would be send again. Secondly is the fact that in case the connection falters the robot would always reconnect itself to the server again. All the sockets and Docker are kept alive for a specific amount of time. When the robot reconnects before this interval is over, the connection is restored again. If the time slot is passed when the robot reconnects, than the robot is given a new connection on a different port. By catching every exception and take the right action the program is never forcedly killed.

Now that the subquestions are answered, we can answer our main research question:
How do we create a generic robot cloud platform that works for both the Nao and the Pepper?
In order to have to solution work for both the Nao and the Pepper robot we had to take into account that we could not change the ports on which the robots listen, as is with the platform currently used by Interactive Robotics. We solved this problem by using virtualization. The fact that each Pepper robot would require its own server was first dismissed as not practical. This holds true for physical servers, but with virtual servers this is doable. However it is not very useful to have a virtual server running for every possible robot that can connect, and as such we needed some management. An employee of Interactive Robotics suggested we use Docker, as it was easy to use and they were also busy integrating it into their new system. Docker enabled us to dynamically create and destroy environments. And because every Docker container has its own ip, we could use the same port for every robot. This allowed us to create a generic solution without it being too resource intensive.

In section 3.4 we listed the requirements in the form of a MoSCoW, Table 8.1 shows these requirements and if they have been met or not with a description of how we succeeded or why we have not. M stands for must have, S for Should have, C for could have, and W for Would like to have.

Priority	#	Requirement met	Description
M	1	yes	By using the reversal NAT traversal the connection always succeeds
	2	yes	Our software is theoretically able to handle as many robots as the server itself is physically able to handle
	3	yes	With the exception of data the server requests itself it passes all data non-discriminatory
	4	yes	By using the TCP protocol, which is itself a reliable protocol
	5	yes	By using robot IDs and running server side connection in their own environment
S	6	partially	Our software itself does not really introduce latency but because of all the Java processes in the Docker container there is some latency between starting the container and actually running the software.
	7	yes	By using Docker every server side connection is run in a separate container
	8	partially	Our current ”clean” robot side does not automatically run at startup but an edited version where all separate modules are merged in one cluttered file did work
C	9	no	Because of time limitations we thought it better to focus on more important requirements
	10	no	Although not difficult to implement, the company’s website did not support this yet so we focused on other requirements
W	11	no	our solution and the company’s server are not yet ready for a full integration as there are still some server side problems that have to be fixed.

Table 8.1: Overview of the requirements and indications whether they are met.

9

Recommendations and future work

In this chapter our recommendations and advice for future works toward the company are grouped. It is divided in two sections. First the recommendations that will improve the system of the company are put apart. After that our ideas for future improvements and/or new features will be explained.

9.1. Recommendations

9.1.1. Java

The original framework from the Interactive Robotics company was mostly written in Java. Since Oracle changed their 'Operating System Distributor License for Java' it is forbidden to redistribute the Java files. Therefor there are no official Java images for Docker available. Because there are some compatibility issues with openJDK, the Docker image with the original Java software had to be fully installed manually in the image. To keep the Docker images maintainable it is better to recreate the software in Python for example.
Furthermore the Ubuntu + Java image is really big in comparison with a Ubuntu + Python image. When everything is written in C++ or Python some significant data storage and RAM could be saved. The latter is a very important when it comes to the scalability of the Docker containers.

9.2. Future work

9.2.1. Docker images

At the moment only one Dockerfile has been made. All the Java modules for every scenario are present in the Docker images. To decrease the size of the images and make it faster, different Docker images could be made, one for every type of scenario. Besides this it is possible that in the nearby feature different modules would be made for the pepper and Nao robot. By dividing the current Dockerfile in a base image and extending this one, the Docker images stays maintainable and up-to-date.

9.2.2. Web interface

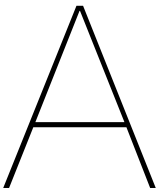
The current web interface is not linked to the web-manager yet. The web-manager has a simple communication protocol through which containers for robots could be launched or stopped. Furthermore the web interface could be linked to the specific SocketIO port of the containers so that the running goal processes could be manipulated.

9.2.3. Autostart of the robot connector

Currently the robot connector at the Nao of Pepper could only autostart by uploading a module through Choregraphe. By the lack of support for dynamic links to the Python scripts, every new version has to be manually add to the module. This makes it vulnerable for mistakes when trying to update the software. In order to fix this a generic module should be created that loads Python files from the file system instead of embed them.

9.2.4. Secure connection

In the future the current connection could be secured by forcing to make use of SSL, or TLS. During this project we chose not to focus to use this, since it makes it harder to debug and there were other criteria to focus on. Because the connection uses a self-made communication protocol it drops the connection when the right responses are not given. When implementing a secure connection the web side of the company should be secured too in order to make it safe.



Feedback Software Improvement Group

De code van het systeem scoort 3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code gemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Unit Size en Duplication.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt.

Jullie methodes zijn nog niet exteem lang, maar het valt wel op dat jullie soms een aanpak hanteren die tot onderhoudsproblemen gaat leiden op het moment dat de hoeveelheid functionaliteit blijft groeien. Zo is Relay.run() duidelijk een centrale methode binnen het systeem. Het zou daar beter zijn om de inhoud van het try-block naar een aparte methode te verplaatsen, net als de error handling logica. Op die manier introduceer je meer abstractie in je code, wat bij een groter systeem zeker van pas zal komen.

Voor Duplication wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren.

Bij jullie valt vooral de duplicatie tussen server_connector en robot_connector op. Zowel de code als de naamgeving wekken de indruk dat het hier om gelijksoortige concepten gaat. Het is dan beter om een parent class te introduceren en de gedeelde functionaliteit daarheen te verplaatsen. Een gerelateerd punt is dat beide nu van hun eigen class Connection gebruik maken. Die class zou dan beter apart kunnen staan zodat dezelfde code kan worden hergebruikt.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Over het algemeen scoort de code dus gemiddeld, hopelijk lukt het om dit niveau nog wat te laten stijgen tijdens de rest van de ontwikkelfase.

Dennis Bijlsma | Senior Consultant
+31 6 45 172 816 | d.bijlsma@sig.eu
Software Improvement Group | www.sig.eu

B

Original assignment as on BEPSys

B.1. Project description

In our interactive robot framework, we would like to integrate sensory sources and deal with sensory events in a generic way. using our AI technology. Students will work in the architecture to do this as well as develop proof of concept integration. Key is that integration is over multiple locations (i.e., websocket/ros/or other methods are to investigated). Students can build upon our existing interaction platform.

B.2. Company description

Interactive Robotics is developing a Robot Interaction Engine, using cognitive technology. With robots quickly becoming more able to assist us, human robot interaction is the next big challenge that needs to be solved for robots to successively enter into our society. Robots not only need to perform tasks for us but need to do so in a way that makes sense to us.
This requires robots with the social intelligence to understand us, robots that have natural interaction capabilities to talk with us, and robots that are able to adapt to us. Our Robot Interaction Engine enables you to quickly develop interactive scenarios for your robot application. Whether it's a robot host, robot waiter, a care robot, or a robot teaching assistant, Interactive Robotics' solutions deliver an optimal interaction with people

C

Speculation for ASD

Week	Task	Done
1	Determine the project organization	y
	Meet the client and supervisor	y
	Create the first MoSCoW	y
	Start of literature research	y
	Formulate a clear problem definition	y
2	Analyse the Naoqi sockets	y
	Build a basic tunnel	y
	Test and vonclude the tunnel requirements	n
3	Build an own connection protocol	y
	Automatically open and close sockets	y
	Create a sequence diagram	n
	Create a flow chart of the connections	n
	Make the multiple sockets threaded	y
4	Support for multiple robots	y
	Split the server infrastructure and make it threaded	y
	Support for temporarily disconnected robots	y
	Update the logger	y
	Create a terminology diagram	y
5	Code refactorization	y
	Implement unit tests	y
	Test implementation on pepper	y
	split code into multiple classes	y
	Remove code duplicity	y
	Create a flowchart diagram	y
6	Integrate with MongoDB	y
	Building Docker containers to run the serverside with test code	y
	Create a robot identification method	y
	Replace all the user set variables from the code to a config file	y
7	Integrate the original Java behavior manager into the container	y
	Building a module for the robots that can be automatically start	y
	Split the manager for better code understanding	y
	Implementing heartbeats for faster port cleanup	y
8	Final report	y
	Small code optimization	y

Table C.1: Table overview of the speculation we compose every week.

D

Terminology diagram

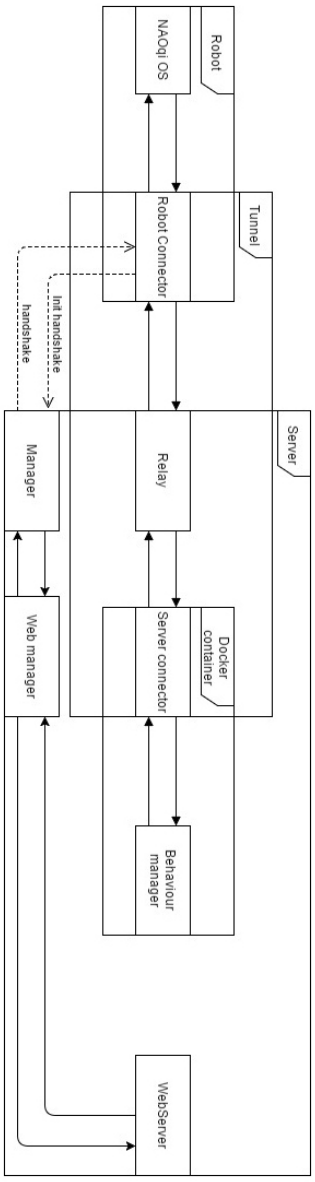


Figure D.1: Terminology diagram. Shows all the name conventions

E

Flowchart diagram

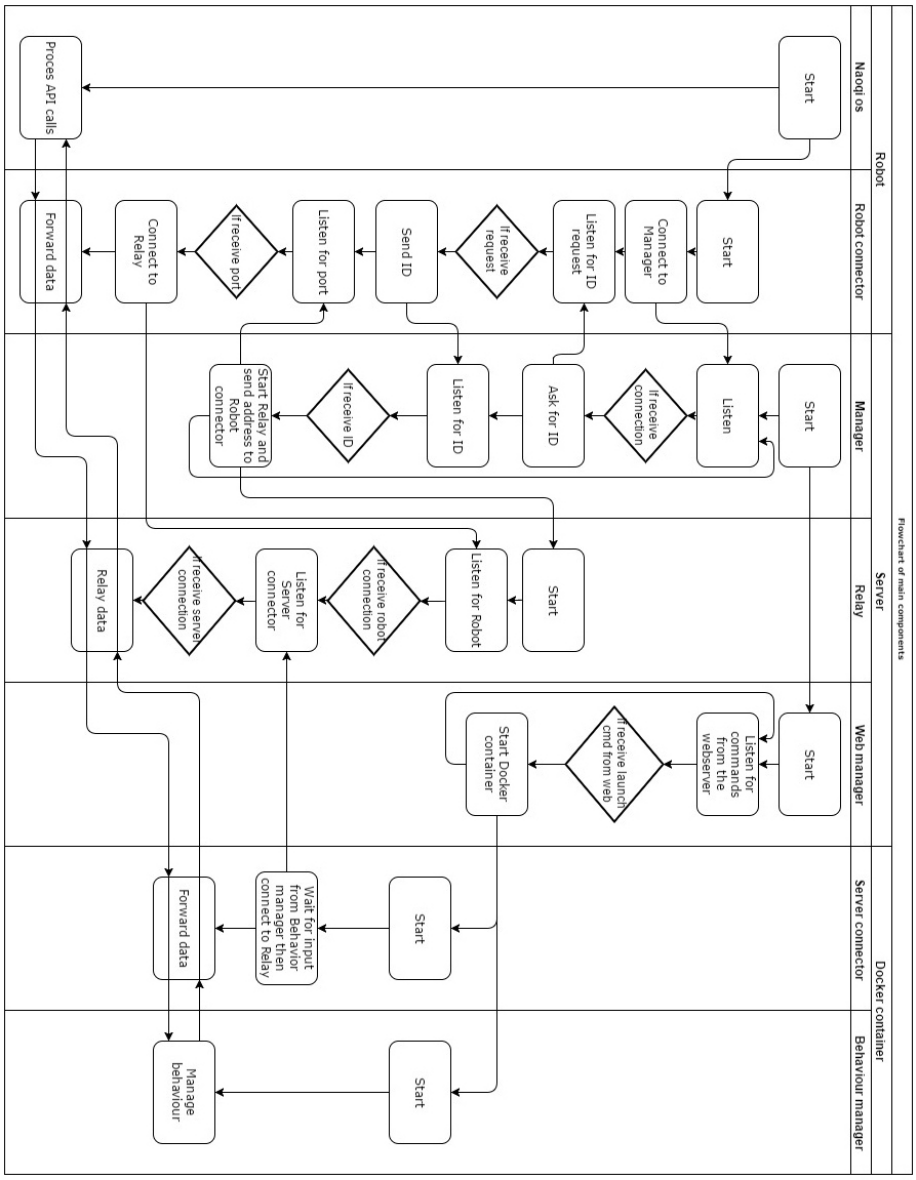


Figure E.1: Flowchart diagram. Shows all the connection decision steps

F

Infosheet

Title of the project: Multimodal sensory pipeline for Interactive Robots
Client organization: Interactive Robotics
Date of final presentation: July 5, 2017

Description:

Interactive Robotics has asked us to create a sensory pipeline that would relay data non-discriminatory and write code for the server to handle all the tunnels. During our research we learned about the communication restrictions imposed by Softbank Robotics causing us to reconsider our solution entirely. This reconsideration is responsible for the structure of our current solution. The product we delivered in the end connected the robot to the server and the server handled all connections. We tested our product by using it with real robots in real life scenarios. The company intends to integrate our solution into their system, but as of now the system is not yet ready for it so it was planned for later this year.

Members of the project team:

Name: Roy Graafmans.
Interests: Robotics, Back-end development.

Name: Millen van Osch.
Interests: Robotics, Domotics, front-end development

Since there were only two of us, we both participated to all aspects of this project.

Client:

Name: Joost Broekens.
Affiliation: Interactive Robotics.

Coach:

Name: Joachim de Greeff.
Affiliation: department of intelligent systems, interactive intelligence group.

Contacts:

Roy Graafmans: roy.graafmans@gmail.com
Millen van Osch: millen_vanosch@hotmail.com

The final report for this project can be found at: http://repository.tudelft.nl

This thesis is confidential and cannot be made public in any form until July 05, 2019 until further notice

Bibliography

[1] Lusa. Pepper, o robot que não nos vai tirar o emprego, 2016. URL https://www.noticiasaominuto.com/tech/685174/pepper-o-robot-que-nao-nos-vai-tirar-o-emprego.

[2] Interactive Robotics. De robotutor, n.d.. URL http://robotsindeklas.nl/.

[3] Softbank Robotics. About us, n.d.. URL https://www.ald.softbankrobotics.com/en/about-us.

[4] Softbank Robotics. Who is nao, n.d.. URL https://www.ald.softbankrobotics.com/en/cool-robots/nao.

[5] SoftBank Robotics. Nao: Construction, 2016. URL http://doc.aldebaran.com/2-4/family/robots/dimensions_robot.html.

[6] SoftBank Robotics. Nao: Motherboard, 2016. URL http://doc.aldebaran.com/2-5/family/robots/motherboard_robot.html.

[7] Softbank Robotics. who is pepper, n.d.. URL https://www.ald.softbankrobotics.com/en/cool-robots/pepper.

[8] SoftBank Robotics. Pepper: Construction, 2016. URL http://doc.aldebaran.com/2-4/family/pepper_technical/dimensions_pep.html.

[9] Rapid Electronics. Pepper robot academic edition, n.d. URL https://www.rapidonline.com/pepper-academic-edition-robot-3-year-warranty-70-8870.

[10] SoftBank Robotics. Pepper: Motherboard, 2016. URL http://doc.aldebaran.com/2-5/family/pepper_technical/motherboard_pep.html.

[11] SoftBank Robotics. Naoqi os - getting started, 2016. URL http://doc.aldebaran.com/2-5/dev/tools/opennao.html.

[12] Softbank Robotics. Connecting choregraphe to a robot, n.d.. URL http://doc.aldebaran.com/2-4/software/choregraphe/connection_widget.html#chore-connection-management.

[13] Softbank Robotics. Key concepts, n.d.. URL http://doc.aldebaran.com/2-4/dev/naoqi/index.html.

[14] Softbank Robotics. Sdks, n.d.. URL http://doc.aldebaran.com/2-4/dev/programming_index.html.

[15] Python Software Foundation. Low-level networking interface, n.d. URL https://docs.python.org/2/library/socket.html.

[16] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, fourth edition, 2009. Dutch translation.

[17] Imran Ahmed Rafai. Chapter 3: Models, 2012. URL http://ccna-cme.blogspot.nl/2012/06/day-4-models.html.

[18] S. N. Srirama and M. Liyanage. Tcp hole punching approach to address devices in mobile networks. In *2014 International Conference on Future Internet of Things and Cloud*, pages 90–97, Aug 2014.

[19] A. Wacker, G. Schiele, S. Holzapfel, and T. Weis. A nat traversal mechanism for peer-to-peer networks. In *2008 Eighth International Conference on Peer-to-Peer Computing*, pages 81–83, Sept 2008.

[20] P. Konstantin, M. Weinert, N. Liebau, and R. Steinmetz. Flexible framework for nat traversal in peer-to-peer applications. Technical Report PWLS07-1, Technische Universität Darmstadt, 2007. Most recent version.

[21] Oliver Haase. Punching holes with java rmi, 2009. URL `http://www.drdobbs.com/jvm/punching-holes-with-java-rmi/217400127`.

[22] Inc Docker. What is docker?, n.d. URL `https://www.docker.com/what-docker`.

[23] MongoDB. What is mongodb?, n.d. URL `https://www.mongodb.com/what-is-mongodb`.

[24] S. Jordán, T. Haidegger, L. Kovács, I. Felde, and I. Rudas. The rising prospects of cloud robotic applications. In *2013 IEEE 9th International Conference on Computational Cybernetics (ICCC)*, pages 327–332, July 2013. doi: 10.1109/ICCCyb.2013.6617612.

[25] J. Wan, S. Tang, H. Yan, D. Li, S. Wang, and A.V. Vasilakos. Cloud robotics: Current status and open issues. *IEEE Access*, 4:2797–2807, 2016. doi: 10.1109/ACCESS.2016.2574979.

[26] ROS. About ros, 2017. URL `http://www.ros.org/about-ros/`.

[27] RoboEarth. A world wide web for robots, 2017. URL `http://roboearth.ethz.ch`.

[28] Rapyuta Robotics. About, 2016. URL `https://www.rapyuta-robotics.com/about`.

[29] Kelly Walters. Prioritization using moscow, 2009. URL `http://www.allaboutagile.com/prioritization-using-moscow/`.

[30] J. Highsmith. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House eBooks. Pearson Education, 2013. ISBN 9780133489484. URL `https://books.google.nl/books?id=CVcUAAAAQBAJ`.

[31] Jim Highsmith. Messy, exciting, and anxiety-ridden: Adaptive software development, 1997. URL `http://web.archive.org/web/20080820120517/http://www.jimhighsmith.com/articles/messy.htm`.

[32] Thomas Lotze. Low-level networking interface, n.d. URL `https://pythonhosted.org/tl.testing/tl.testing-thread.html`.