An Autonomous Bicycle

GUI, Database, Communication

Christian Kreipl

September 28, 2017

No previous work to built upon. Everything has to be done from scratch.

Contents

1 Goal **4**

2 Choice of the mobile device **5**

3 GUI **10**

4 Database **13**

5 Communication **17**

6 Student project **23**

1 Goal

2 Choice of the mobile device

At first we are supposed to choose a suitable mobile device to display data, send commands to the bike and display recorded data. To be able to decide on a certain device, we need to specify its requirements. We consider the following points:

- Display size and resolution
- Connection options
- Operating system
- Input type

With respect to:

- Mobility
- Usability
- Effort to maintain and setup
- Price

To rate the options we introduce the rating system shown in table 2.1.

Very positive	Positive	Neutral	Negative	Very negative
+ +	+	0	-	- -
Very important	Important	Neutral	Unimportant	Very unimportant
VI	I	0	U	VU

Table 2.1: Ratings and weights

We rate with and without respect to the price. In the latter case the price is matched to a reasonable device. E.g. if there is a 600€device and another device, that fits our purposes, for 200€, the cheaper device would be chosen. Usually there are cheaper devices available then the minimal price that is stated in the following tables. This devices are usually to weak to be considered as relevant candidates and therefore left out.

	Small ($< 5''$)	Medium ($5'' - 10''$)	Large ($> 10''$)	
Usability	fairly hard to see details	easy to see details	easy to see details	VI
Price	80€+	150€+	200€+	I
Resolution	up to 2160p	720p - 2160p	up to 4K	U
Weight	low	low - medium	medium - high	0
CPU	low - medium	medium	medium - high	I
Rating w/o €	-	0	+	
Rating with €	-	+	0	

Table 2.2: Display size and resolution

The "small" devices are smartphones, the "medium" sized devices are usually tablets or netbooks and the "large" devices are laptops. Desktop PCs are ruled out because of their lack of mobility. The smaller the device is, the higher is the mobility of the user, but also it is problematic to display a lot of data at the same time and allow control. The price scales with the computational power of the device.

	Touch display	Mouse & keyboard	
Mobility	high (smartphone, tablet)	medium - high (laptops)	I
Usability	high	high	VI
Price	0	0	U
Effort to setup and maintain	high if OS $\neq$ Android, otherwise low	low	VI
Rating	+ if OS = Android, otherwise - -	+	

Table 2.3: Input devices

In the considered devices the input type is already built in. Therefore no additional costs are added. This applies to other parts aswell. For instance are almost any mobile devices equipped with WLAN and Bluetooth. If a touch display is chosen, the operating system should be Android (iOS devices are generally to expensive for our purposes).

	WLAN	Mobile network	Bluetooth	
Range	70m	global	5-10m	I
Price	0	2×10€ + devices to setup the connection, e.g. LTE hotspot 80€	0	0
Volume	∞	1GB Data / Month	∞	I
Latency	1-80ms	GSM: latency 500 ms+ EDGE: latency 300-400 ms UMTS: latency 200 ms LTE: 35-40 ms (advertised with 10 ms)	40-250 ms (40ms was the lowest latency advertised)	I
Effort to setup and maintain	low	high	high	VI
Rating w/o €	+ +	0	-	
Rating with €	+ +	-	-	

Table 2.4: Connections

The best option for the connection type is WLAN. It has a sufficient range, is free is fairly easy to use and maintain. It is assumed, that the bike is in an area that is free of disturbing signals. The bike supports both 2.4 Ghz and 5 Ghz. However it is not able to open a 5 Ghz hotspot due to legal restrictions. The WLAN adapter is not region locked and therefore can't open an access point. It can however connect to a network that is spanned by a router or the mobile device. Bluetooth would only be an option if the usecase would be restricted to someone moving in close distance to the bike, e.g. riding on it. The global range of the mobile network comes at the price, that a mobile device must be attached to the bike be able to open the connection. This increases the risk of failures, limits the data that can be sent and is a huge effort to setup. In the current usecase the global range is not even useful, as the bike is not expected to move far from the controller. hier test einfgen!

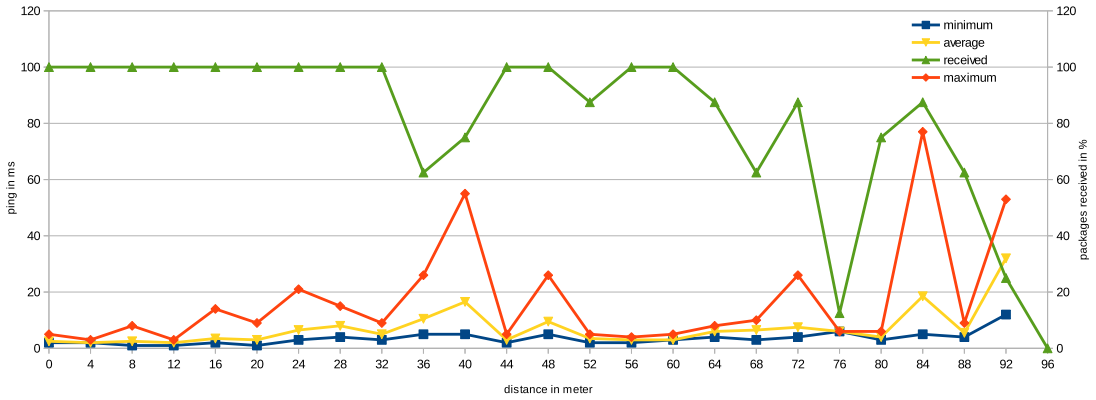

Figure 2.1: Measurement of the latency of a 2.4 Ghz WLAN connection to the bike in relation to the distance of the bike and the mobile device. It was tested close to the parking lot behind the MI building, without interfering networks. The measurement was carried out by Jorge Jiménez and Christian Kreipl and the underlying data can be found in the file: "wlanMeasurements.xlsx"

	Android	Linux	Windows	
Usability	good support for touch, allows easy access to sensors	possibly bad driver support	usually good driver support	VI
Price	0	0	0-280€	0
Effort to setup and maintain	high	low	low	I
Rating w/o €	0	0	+	
Rating with €	0	0	0	

Table 2.5: Operating system

The operating systems are all fairly equal with their own strengths and weaknesses. None of them has a huge advantage against the others. A touch display probably has the best support under Android and Windows 8. The additional access to the built in sensors would make the effort to set it up and get an application running valuable. If no special hardware is required, Linux provides an open environment and a lot of freedom. The operating system should therefore be chosen with respect to the underlying hardware.

	Android	Linux	Windows
Smartphone	-	N/A	-
Tablet	+	+	+
Laptop	0	0	0

Table 2.6: Final comparison

As shown in table 2.6, there is no clear favorite. Depending on personal preferences, every combination of hardware and operating system can be viable. In the authors opinion a tablet suits the purpose best. It has a decent amount of computational power and other resources. Further it is very mobile and allows the extension of the communication through a mobile network by LTE support. A suitable device is the "Lenovo IdeaPad MIIX 310-10ICR, 64GB Flash, 4GB RAM, LTE" that can be bought for 300€(by end of September 2017) and in some special offers even for less. It allows good mobility and brings all considered types of connections. The keyboard allows to use it like a laptop, what makes it easier to program directly on it. If it is not required, the weight can be halved to only 580g. The battery provides a running time of up to 10 hours, which is more then enough.

Because of the free availability of Windows and Linux laptops, the final decision was to implement everything platform independent and to design it touch friendly. This allows further groups to make their own decisions, if the requirements changes.

3 GUI

The GUI is created with QT-Designer and the .ui files are directly loaded into the main GUI file. That way everything can be redesigned with a few clicks as long as all elements are preserved. If elements are replaced, it is necessary to update the file: "mainGUI.py". This file connects all buttons to actions and fills the widgets with content. All changes to the visible part of the GUI are either made in the ".ui" files or in "mainGUI.py". To allow the usage of tablets, it is designed to be touch friendly. The resolution is $1051px \times 490px$, can be displayed at almost any modern mobile device and is easily resizeable to the exact size of a device. The main view of the GUI, in its current state, is shown in figure 3.1.

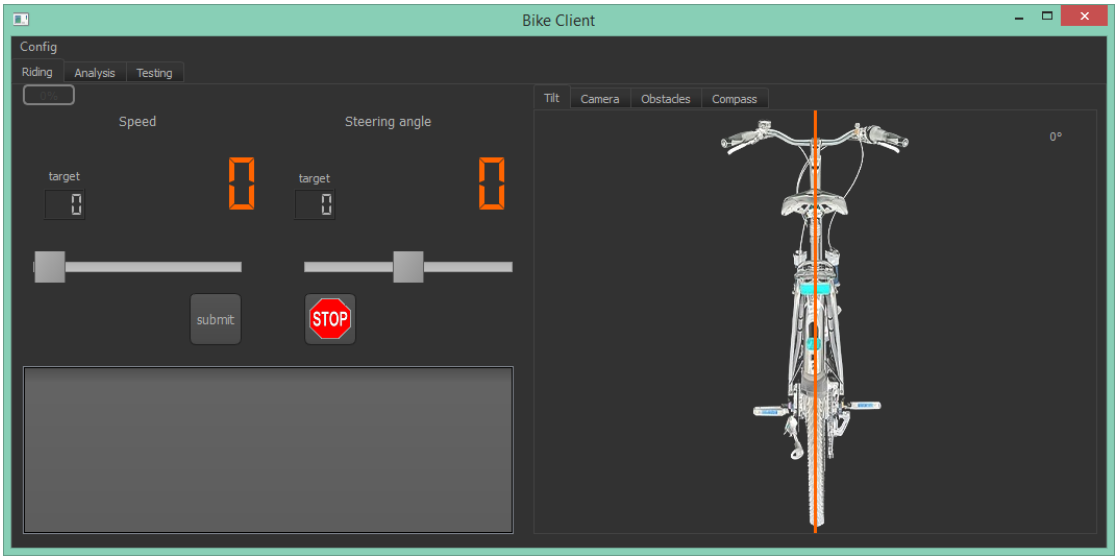

Figure 3.1: Main view of the GUI

The GUI is divided in three parts. This parts correspond to the tabs "Riding", "Analysis" and "Testing". The "Riding" tab is supposed to show relevant information about the bike while it is running and to allows to send control commands to the bike. This includes the battery state, the current speed, the steering angle, a news feed, the current tilt level of the bike, a live-stream from the camera, a graphic displaying detected objects and a compass. The battery state and the compass are currently disabled. It was not yet possible to get the state of the battery and the laptops lacked a compass to be able to present the position of the bike relative to the mobile device. The live video consists of grayscale images. It supports rgb images as well (implemented, but

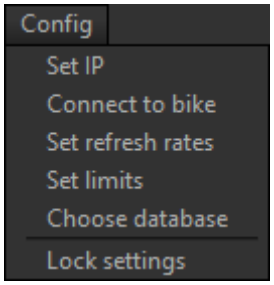

Figure 3.2: Configuration menu

some small changes are necessary). The grayscale images were chosen because of there smaller size. To control the target speed and steering angle, two touch friendly sliders allow to set the values. A click on the "submit" button will send the values to the bike if possible. The "stop" button sends a top priority signal to reduce the speed to 0. Figure 3.2 shows the configuration menu. It allows to set the IP address and port, that the bike uses and then to connect to the bike. The result will be displayed at the news feed. Further it allows to set the rates in which data is requested from the bike and to set the maximal and minimal tilt angle. "Choose database" allows to select a database from the file system and connect to it in order to display the stored data. The final option disables the menu except of the "Connect to bike" entry. This mode allows interaction with the bike without the user being able to interfere with important settings. This might be useful in case of a demonstration. To re-enable the settings, the GUI has to be restarted. "Connect to bike" is always enabled because it is safety relevant to be able to gain back lost control.

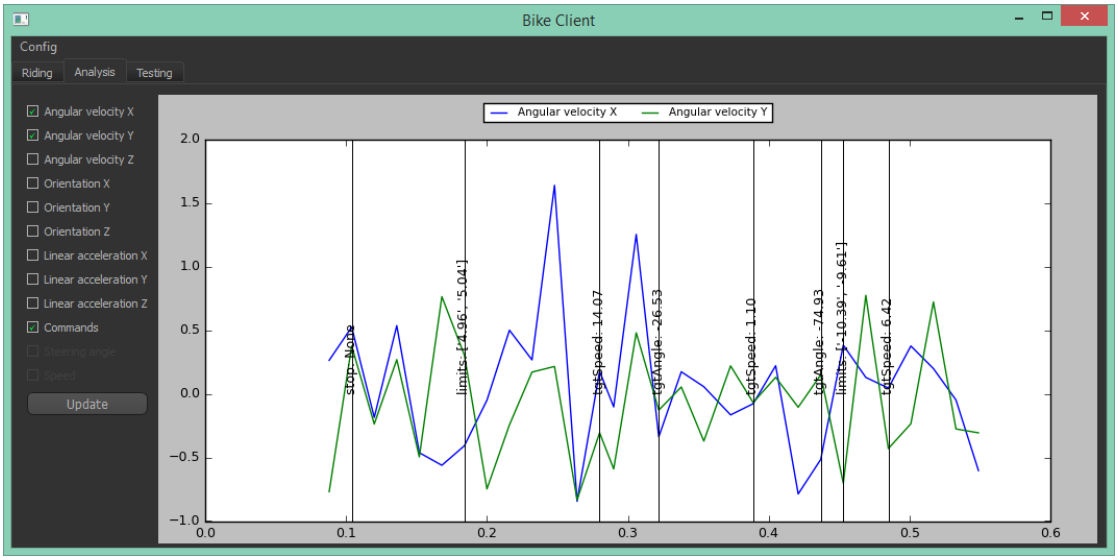

Figure 3.3: Analysis tab with some dummy data displayed.

Figure 3.3 shows the analysis tab after some dummy data was selected to be displayed. The axes and colors change accordingly to the data shown. The vertical lines indicate commands sent to the bike. If more then 7 data sets are chosen, the colors will be reused. It is strongly advised against displaying that many graphs, as it gets very hard to recognize something. The options "Speed" and "Steering angle" are disabled because this data is not recorded.

The third tab "Testing" is currently not in use, but is intended to give developers space to display custom data while testing their application. Especially in the case that it is not yet in a format to be displayed in the main application or is not intended to be displayed at all. When this project is finished, the tab should be removed with QT-designer.

4 Database

MongoDB	sqlite3	
multiple processes can write at once	locks if multiple processes write at once (potentially slower)	U
runs on almost any little endian system	python built in	I
NoSQL (very simple structure & low data volume)	SQLite faster execution & JOIN clause	I
some effort to setup and use)	easy to use and doesn't require a separate process	0

Table 4.1: Comparison of two database systems

With the inter process communication between Control, Obstacle Detection, the Core module and the GUI using ROS, all data can be aggregated at one ROS-node and be written to the database. This reduces the maintain effort for future groups as it provides a better modularity. Therefore there will always be only one process writing to the database. Thus it isn't necessary for the database to provide multiple user support in terms of writing. As seen in table 4.1, the python inbuilt database has some more advantages against external database systems and especially against NoSQL-databases. Figure 4.1 displays the information flow. The only method writing to the database is "databaseWriter()". It gets its input from a modified version of "listenerCombined.py" and adds timestamps to the data as soon data is handed over. By this it is not necessary to sync the clocks, but the timestamps may be delayed. It is possible for every process to create its own "databaseReader()" object and call all the reading functions. However there should be only one "databaseWriter()"-object at any time, as it can get very slow if two or more objects try to write at the same time. This is especially true, if the amount of data is always small and committed in every step to be written to the disk. This doesn't mean it isn't possible to write from multiple processes. In fact, the "databaseWriter()"-object can be passed around and every process can call the write methods at the same time. The elements will be added to a queue and periodically be written to the disk. To pass the object around the segregation between the modules has to be broken or a way figured out, how to pass the intact object through ROS. Figure 4.2 visualizes how the "DB writer" works. The classes "DB writer" and "DB reader" can be found in the file "database.py". To analyze the database it has to be copied to the mobile device or the GUI has to be executed on the bike. A remote access can be implemented, but wasn't important yet, as the data is only analyzed after a test drive.

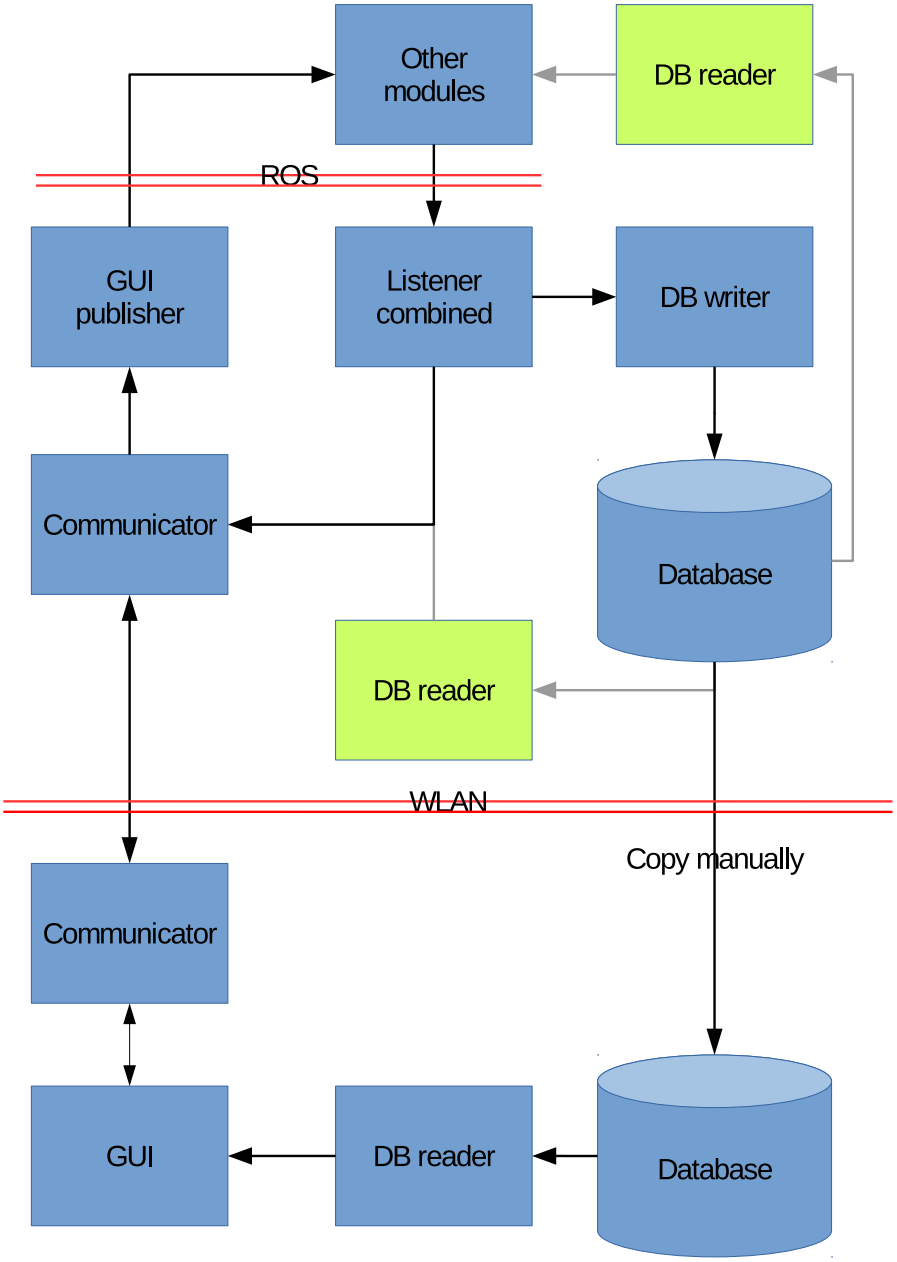

Figure 4.1: Information flow. All blue parts are implemented and working, the green parts are possible (but not yet implemented parts) and arrows indicate the flow of information. The red line indicates the separation of bike and mobile device.

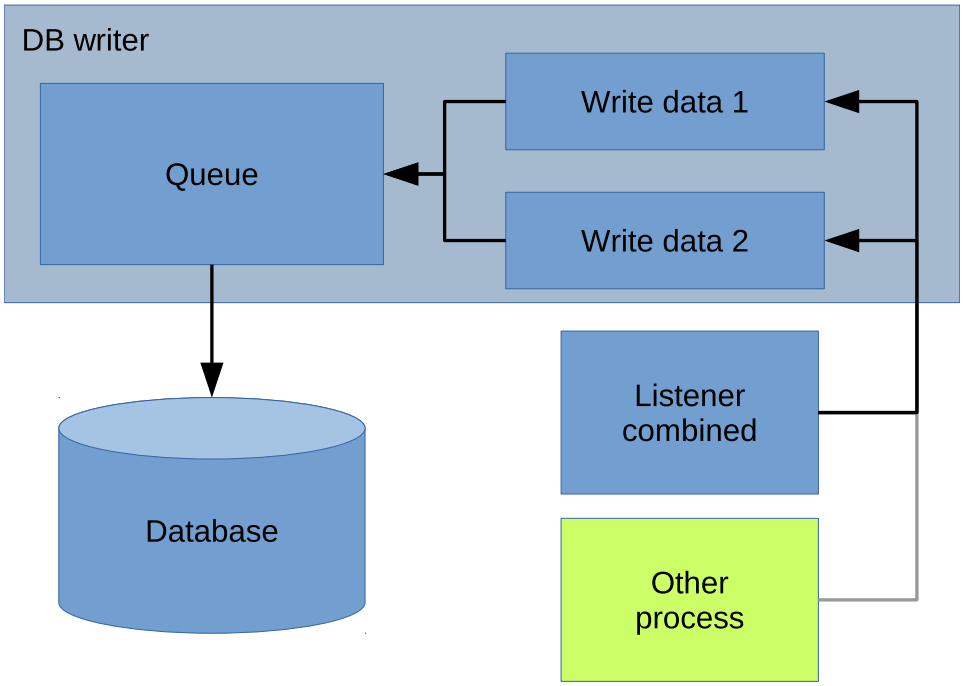

Figure 4.2: Visualization of the database writer. By calling the "write data" methods, SQL statements are created, that are added to a queue and added to the database by a separate process. By this construction the writing process is not blocking and writing to the disk can be efficient as it only happens if a lot of write statements are executed or the queue is empty.

Method	Arguments
addIMUData(acceleration, orientation, rotation)	[**float** accX, **float** accY, **float** accZ], [**float** orX, **float** orY, **float** orZ], [**float** rotX, **float** rotY, **float** rotZ]
addGPSData(longitude, latitude)	**float** longitude, **float** latitude
addCommand(commandType, value=None)	**str** type, **None** or [**float** min, **float** max]
addSpeed(speed)	**float** speed
addSteeringAngle(angle)	**float** angle
addObstData(obstacles)	[(**float** xL_1, **float** xR_1, **float** z_1), (**float** xL_2, **float** xR_2, **float** z_2), ..., (**float** xL_n, **float** xR_n, **float** z_n)]

Table 4.2: Implemented methods for "DB writer"

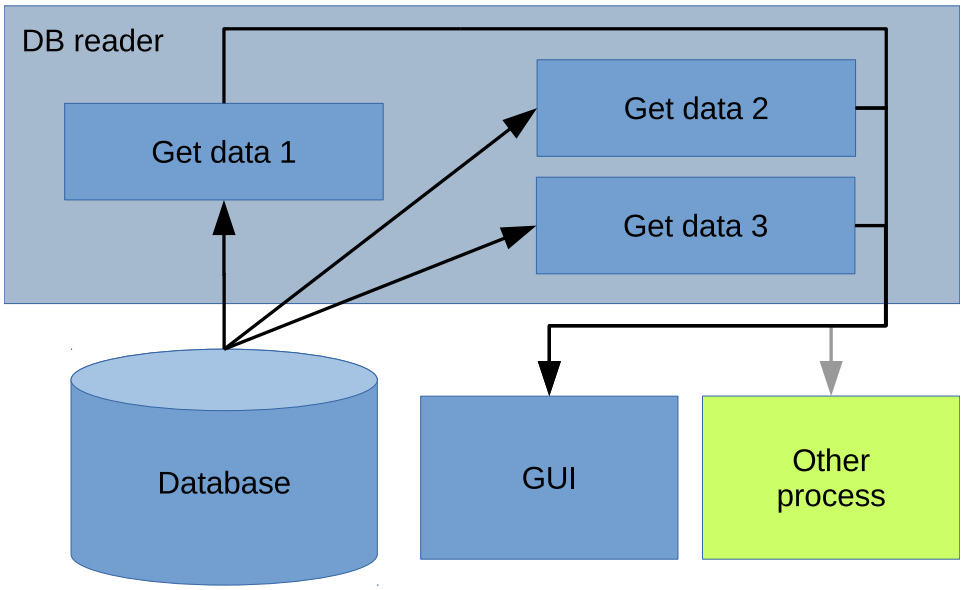

Figure 4.3: Visualization of the database reader. The scheme is simplified.

Method	Return values
getSpeed()	[**float** $datum_1$, ..., **float** $datum_n$], [**float** $speed_1$, ..., **float** $speed_n$]
getSteeringAngle()	[**float** $datum_1$, ..., **float** $datum_n$], [**float** $angle_1$, ..., **float** $angle_n$]
getImuData()	[**float** $datum_1$, ..., **float** $datum_n$], [**float** $angVelX_1$, ..., **float** $angVelX_n$], [**float** $angVelY_1$, ..., **float** $angVelY_n$], [**float** $angVelZ_1$, ..., **float** $angVelZ_n$], [**float** $orientX_1$, ..., **float** $orientX_n$], [**float** $orientY_1$, ..., **float** $orientY_n$], [**float** $orientZ_1$, ..., **float** $orientZ_n$], [**float** $linAccX_1$, ..., **float** $linAccX_n$], [**float** $linAccY_1$, ..., **float** $linAccY_n$], [**float** $linAccZ_1$, ..., **float** $linAccZ_n$]
getCommands()	[**float** $datum_1$, ..., **float** $datum_n$], [**str** $type_1$], ..., **str** $type_n$], [**None** or **float list** $value_1$, ..., **None** or **float list** $value_n$]
getGps()	[**float** $datum_1$, ..., **float** $datum_n$], [**float** $longitude_1$, ..., **float** $longitude_n$], [**float** $latitude_1$, ..., **float** $latitude_n$],

Table 4.3: Implemented methods for "DB reader"

5 Communication

To keep the system as flexible as possible, it is import to keep the communication independent from the underlying operating system. ROS already provides a strong communication tool but offers only good support on Linux, bad support on Windows and no support for Android. With the python built in networking packages it is possible to create a communication system that suits our purposes best.

An overview over the communication mechanism is given in figure 5.4. The "Communicator" class in "communicator.py" implements both ends of the inter network communication. On the bike it is run as server and on the mobile device as client. The bike and the mobile device have to be in the same wireless network. It is possible to run the bike as 2.4 GHz hotspot and to connect the bike to it. Then the GUI can be connected to the server. Figure 5.2 shows the process of sending two different messages. The messages are transmitted using the TCP. This ensures that the messages are received. A serial number and a timestamp is added to prevent the reuse of old messages. This not yet useful, as the clocks of the bike and mobile device are not synced, but allows an easier cryptographic extension. To achieve a secure connection, only a cryptographic procedure must be added to "Pack Msg" and to "Unpack Msg".

The sensor data is not sent to the GUI by default. The GUI is requesting the data, that shall be displayed. That way data that is not displayed can be excluded. The frequencies in which the data is requested can be set at the GUI, while it is running. The "Refresher" class in "communicator.py" periodically adds request messages to the priority queue. It uses a sleep command to wait the specified time. This can cause a problem if the time is set to a very high value by accident. Then this time will be waited before the higher frequency is used. It is visualized in figure 5.1. Currently all displayable data is updated all the time, but the system is designed to support pausing it. In the case that the traffic induces costs, the not displayed data should not be updated. The messages are sent with different priorities (see table 5.1) to guarantee that certain commands are received quickly. The currently supported message types are shown in table 5.3.

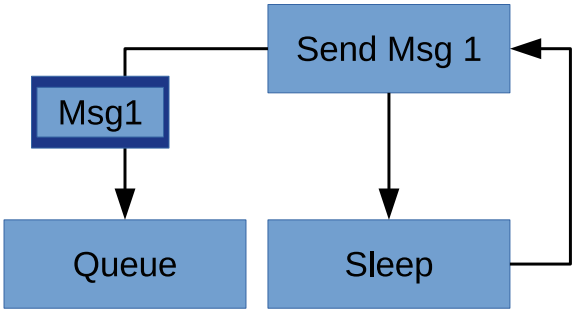

Figure 5.1: Refresher

Priority	Description
TOP	Highest priority will always be sent first. Currently only used for STOP messages.
HIGH	Currently only used for SETTARGETS messages.
NORMAL	Used for almost any message.
LOW	Currently not used but might be used to transmit the video stream, if the network is overloaded.

Table 5.1: Priorities

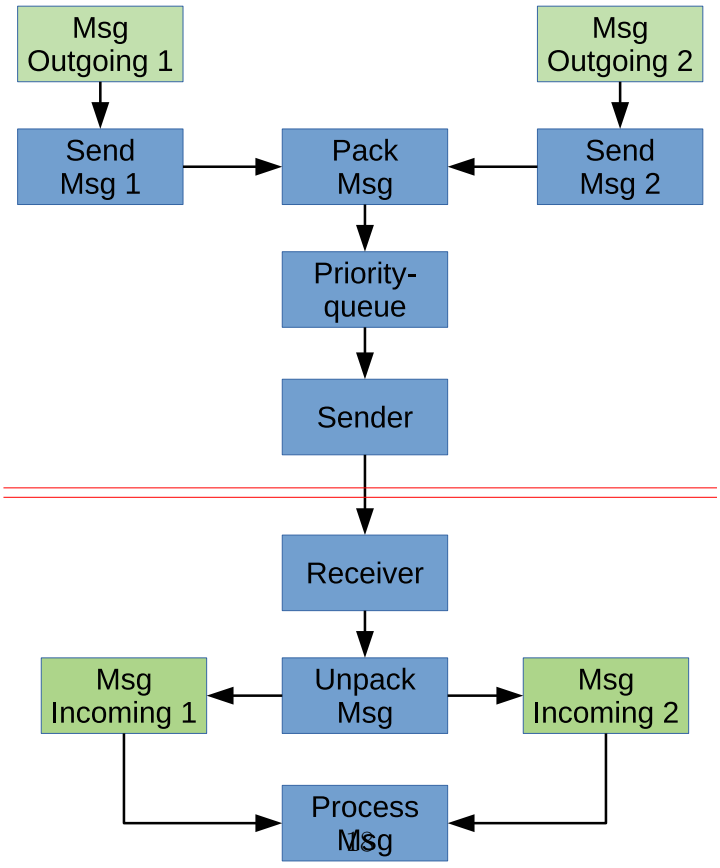

Figure 5.2: Sending of two different messages. See table 5.2

Send Msg	Fetches important data from the system, adds a priority and forwards the message to "Pack Msg"
Pack Msg	Adds a timestamp, a serial number and a number encoding the message type. The it uses pickle to serialize the data
Sender	Locks until the priority contains an element, then sends it to the corresponding "Receiver". More then one message may be sent at once, based on the usage uf TCP.
Receiver	Locks until data is received, then splits it to packed messages.
Unpack Msg	Unpickles the message and checks if it is to old or the sequence number was already used. In case of a failure (e.g. a broken message) the returned message type is **False**.
Process Msg	Checks for the message type and initiates, according to it, the following reaction.

Table 5.2: Short description of the methods involved in the sending process. See figure 5.2

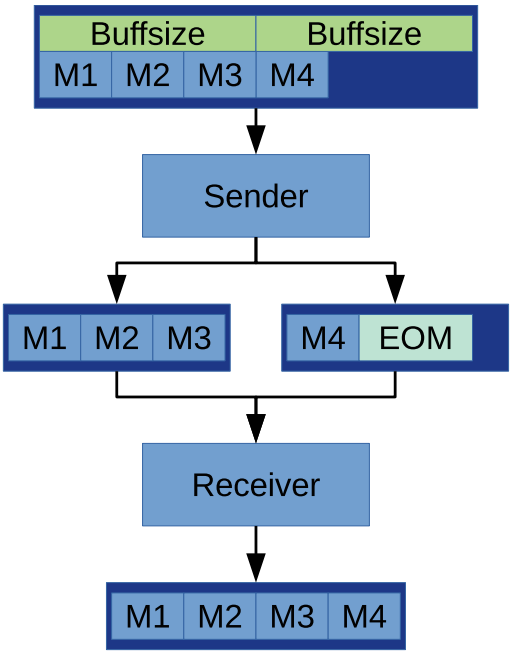

Figure 5.3: Process of sending a single message. If its size exceeds Buffsize, it is split into parts of the length Buffsize. Before splitting, a "End Of Message' string is added at the end of the message. If this string is sent through the network (which is very unlikely), the message containing it will be corrupted.

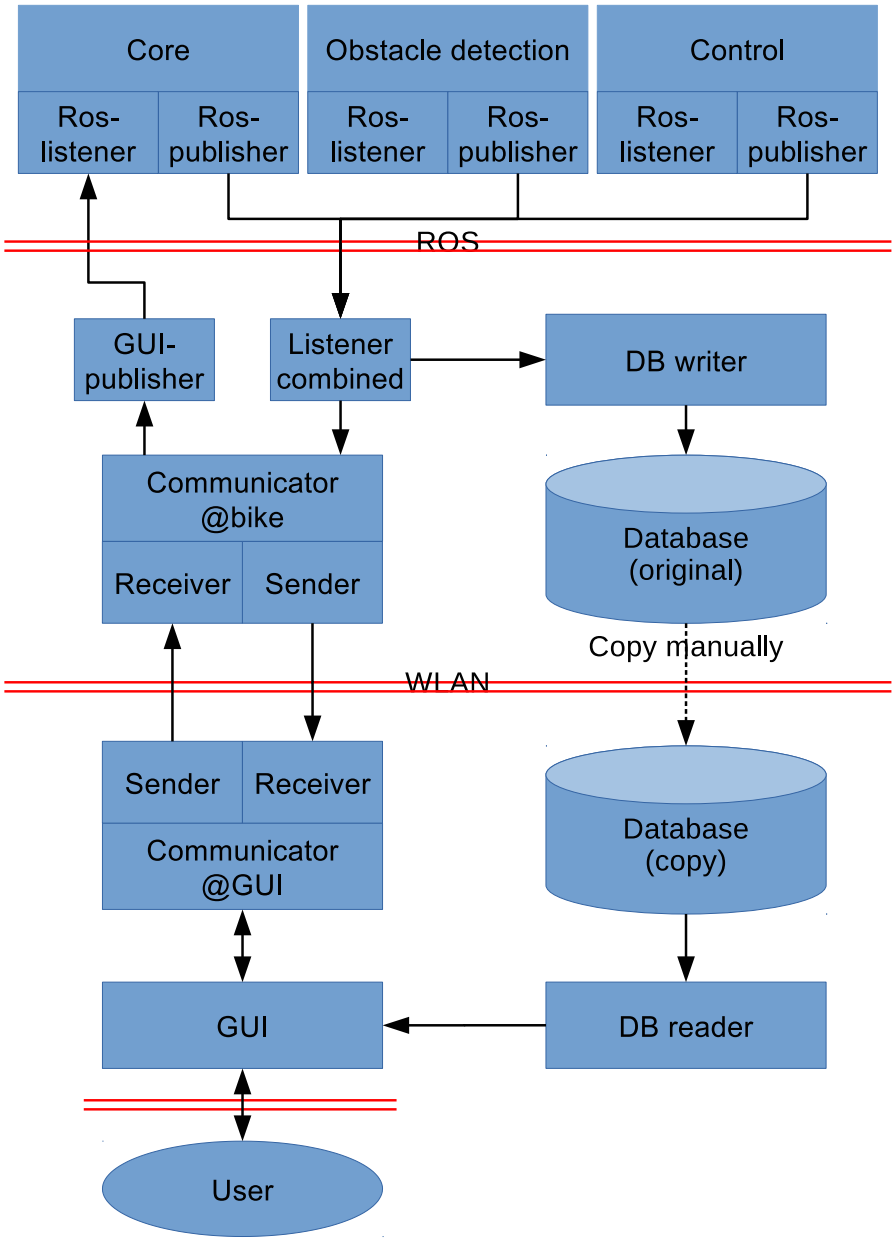

Figure 5.4: Scheme of the inter process communication from the GUI up to the ROS nodes "GUI publisher" and "Listener combined". Adjacent modules are part of the biggest one. Arrows indicate the direction of the information flow.

Message type	**int**	Description
DUMMY	0	does nothing
OK	1	acknowledge the income of a certain message
SHUTDOWN	2	stop the bike as soon as possible
GETSPEED	30	requests the current speed
RETURNSPEED	31	return for GETSPEED
SETTARGETSPEED	32	sets a new target speed value
GETTARGETSPEED	33	requests the target speed value set by Control
RETURNTARGETSPEED	34	return for GETTARGETSPEED
GETTILT	40	request the tilt level of the bike (orientationY)
RETURNTILT	41	returns for GETTILT
GETOBSTACLES	50	requests the list of detected obstacles
RETURNOBSTACLES	51	return for GETOBSTACLES
GETBATTERY	60	requests the battery state
RETURNBATTERY	61	returns the battery state
WARNING	70	sends a message to the GUI to be displayed
GETSTEERANGLE	90	requests the current steering angle
RETURNSTEERANGLE	91	returns for GETSTEERANGLE
GETTARGETSTEERANGLE	92	requests the value of the current target steering angle
RETURNTARGETSTEERANGLE	93	return for GETTARGETSTEERANGLE
SETTARGETSTEERANGLE	94	sets a new value for the target steering angle
GETORIENTATION	100	requests the orientation of the IMU
RETURNORIENTATION	101	return for GETORIENTATION
GETCAMERAIMAGE	110	requests the current camera image
RETURNCAMERAIMAGE	111	return fpr GETCAMERAIMAGE
UPDATELIMITS	120	sends new values for the maximum and minimum tilt angle
SETTARGETS	130	sends new values for target speed and target angle

Table 5.3: List of supported message types. Not all types are in use. New message types can be added easily.

The tables 5.4, 5.5, 5.6 and 5.7 are based on the file "Interfaces.pdf" and the content of them were provided by the groups providing the data. E.g. the content of table 5.6 was provided by the obstacle detection group. All received data is expected to be of the stated types and in the specified units.

	Type	Additional information
Target speed	**float**	km/h [0:35]
Target steering angle	**float**	degree [-90:90]
STOP	**bool**	Emergency stop signal for control.

Table 5.4: Datatypes provided by the Visualization group

	Type	Additional information
Speed	**float**	km/h [0:35]
Steering angle	**float**	Radiant [-100:100]

Table 5.5: Datatypes provided by the Control group

	Type	Additional information
Obstacles	$[(xL_1, xR_1, z_1), (xL_2, xR_2, z_2), \ldots, (xL_n, xR_n, z_n]$	Coordinates relative to bike in world units (m/cm)

Table 5.6: Datatypes provided by the Obstacle detection group

	Type	Additional information
Orientation	**float**	values are in degrees
Linear Acceleration	**float**	values are in meter per second per second ($\frac{m}{s^2}$)
GPS data	**float**	Longitude and Latitude Coordinates
Angular Velocity	**float**	values are in rad/s

Table 5.7: Datatypes provided by the Core group

6 Student project

List of Figures

2.1 WLAN latency measurement 8

3.1 Main view of the GUI 10
3.2 Configuration menu 11
3.3 Analysis tab 11

4.1 Information flow 14
4.2 Database writer 15
4.3 Database reader 16

5.1 Refresher 18
5.2 Sending of two different messages. See table 5.2 18
5.3 Send a single message 19
5.4 Communication 20

List of Tables

2.1 Ratings and weights 5
2.2 Display size and resolution 6
2.3 Input devices 6
2.4 Connections 7
2.5 Operating system 8
2.6 Final comparison 9

4.1 Comparison of two database systems 13
4.2 Implemented methods for "DB writer" 15
4.3 Implemented methods for "DB reader" 16

5.1 Priorities 18
5.2 Short description of the methods involved in the sending process. See figure 5.2 19
5.3 List of supported message types. Not all types are in use. New message types can be added easily. 21
5.4 Interfaces I 22
5.5 Interfaces II 22
5.6 Interfaces III 22
5.7 Interfaces IV 22