Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

A flexible visual framework for debugging complex robotic systems

Felix Kaser

Masterarbeit im Elitestudiengang Software Engineering

Software Engineering
Elite Graduate Program


Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

A flexible visual framework for debugging complex robotic systems

Matrikelnummer:	1174068
Beginn der Arbeit:	21. Juni 2012
Abgabe der Arbeit:	14. Dezember 2012
Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Betreuer:	M.Sc. Andreas Angerer
	Assoc. Prof. Bruce A. MacDonald, Ph.D.

Software Engineering
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 14.12.12 Felix Kaser

Acknowledgements

I would like to thank everyone who supported me in the last months either by providing professional advice and guidance or just being a good friend and listen to my ideas and thoughts. This work would have never been possible without the support from Prof. Dr. Wolfgang Reif, who established the connection to the University of Auckland in New Zealand, Dr. Dominik Haneberg, who made the exchange possible and Andreas Angerer who helped me to find my way and write this thesis. I would also like to thank the Robotics Group at the University of Auckland under the supervision of Assoc. Prof. Bruce MacDonald, where I was able to work on the project. You inspired me with the work you do and were of great help whenever I needed it.

No words can express how thankful I am for the support of my parents Annemarie and Martin, and my sister Lisa. Thank you for your never ending support. I can count on you in good times and in bad.

There are so many people who helped me, it is hard to name them all. I thank you, if you ever had a coffee with me and talked with me about work and the life beyond work.

Abstract

Data collected during debugging is traditionally rendered as text. The special application field of robotics faces problems with this approach, since a typical robotic system constantly gathers and processes data from the surrounding environment through sensors. Developers of robotic applications are confronted with large amounts of data during debugging, which becomes hard to interpret if the data is represented as text and large amounts of data need to be interpreted at once. Robotic applications are also hard to interrupt during debugging, since robots generally do not run in a deterministic and suspendable environment. This makes it hard to reproduce the situation in which a bug occurred and makes it necessary to support the debugging process without interrupting the robot program.

This thesis partially solves this problem by introducing a visual debugging system to support debugging of robotic applications: It takes into account the special requirements for a debugging tool in a robotic development environment, especially the uninterrupted rendering and the need for better visualization of debugging data to support a faster interpretation of this data. The goal of the developed system is to help developers understand the data during debugging more quickly and improve overall productivity during robot development. To achieve this goal, a system was designed and developed where developers can choose how they want to visualize data collected during debugging of a robotic application. This work presents the design and a prototypical implementation of the proposed system. Evaluating the hypothesis that the developed system can improve productivity during debugging is out of scope for this work but initial experiments were promising and the developed tool provides a solid base for further evaluations.

Contents

List of Figures

List of Listings

1 Introduction

Current debugging tools for robotics are mostly based on techniques and interfaces developed for traditional non-robotic applications. Those techniques and interfaces were developed specifically to suit the requirements of traditional systems: First, they assume that a program can be interrupted during its execution, which in a robotic environment is not possible most of the time. Second, the data handled in traditional applications is usually discrete and based on user input, as opposed to the sensory data a robot handles. The data a robot handles is in general closely related to the real world environment and comes from sensors that provide a continuous data stream of readings. Although the computing world has changed in recent years, many tools still rely on the original principles developed for different requirements. Additional tools had to be developed to complement or extend traditional tools with robotic specific functionality [1].

Traditional debugging tools usually render data collected during debugging as text and it is the developer's task to interpret the data in order to get a better understanding of the program and find possible bugs. A traditional software environment is usually suspendable and the execution of a program can be interrupted to give the developer enough time to interpret and analyse this textual data. On the other hand, it can be a problem to interrupt the execution of a program in a robotic environment, because robots generally do not run in a deterministic and suspendable environment [2]. Interrupting a robotic application would destroy the continuity in which the sensors collect data, the environment of the robot would change substantially and thus the robot's behaviour would change. This is an example of the "probe effect" and it makes it hard to reproduce a fault unless it has a single cause [2]. Thus it is necessary to collect data during debugging without interrupting the execution of the program, which means the developer has significantly less time to interpret and analyse the debugging data.

Murphy et al. found that despite the existence of tools to support debugging, many novice developers still use "printf-style" debugging [3]. They only examined

novice debuggers, but older studies indicate that both expert and novice debuggers use the same strategies for debugging [4]. Debugging with print or logging statements is a basic approach, which does not require external tools, but to use it the source code must be modified. Source code modification can be a problem with so-called "Heisenbugs", software faults that disappear because the observation affected the bug [5]. The combination of textual data from print or logging statements with an uninterruptible environment requires the developer to constantly parse and interpret the debugging messages. This results in a significant cognitive load for developers during debugging [6].

Since the robotics field has different requirements for debugging tools than the traditional software development field, many new tools have been developed and existing tools have been adapted to suit the special requirements. Despite the effort to create specialized tools for robotics, many developers still develop their own tool because existing tools are not flexible enough for their application area [7]. To fulfil the special requirements for robotics and to counter the "one-tool-per-project" trend this work proposes a new kind of debugging system. The proposed system visualizes data during the debugging process in robotics and is flexible enough to be used for many different application areas. The visualization focuses on simple data which would usually be printed on a development console or logged to a file. Existing debugging tools and methods were to be analysed first, to identify possible strengths and weaknesses of those tools and methods.

This work is based on the assumption that a flexible visual debugging system can support the developer during the debugging of a robotic application and reduce the cognitive load during debugging. It thus improves the development speed and developer satisfaction. Such a hypothesis can only be evaluated with an extensive user study, which is out of scope for this work. The purpose of this work is to design and develop such a flexible visual debugging system, which can be evaluated in the future. The development of such a system is a necessary step for further evaluations.

The remainder of this work is structured as follows. Chapter 2 presents related tools and methods that are used to debug robotic applications. At the end of Chapter 2 the presented tools and methods are summarized and existing problems are outlined. Chapter 3 presents the design of a generic visual debugging system for robotics. It is generic because it does not target a specific framework for robotics and can be implemented for any of the existing frameworks. The chapter contains goals and requirements for the debugging system which have driven the design and development. It also presents the architecture and system design of the proposed debugging tool in more detail. The presented design has been implemented in the ROSDashboard prototype which is described in detail in Chapter 4. ROSDashboard is a prototypical implementation of the proposed design for ROS (Robot Operating System) which

represents one of the existing robotics frameworks. Chapter 5 contains a case study that shows the possibilities of the developed tool presented in Chapter 4. It also shows how the existing tool can be extended with more visualizations. The last chapter of this work, Chapter 6, summarizes the possibilities to extend and improve ROSDashboard in the future and a summary of this project concludes the thesis.

2 Debugging in Robotics

Due to the specific problems developers face when developing robots, specialized tools have been developed. Some of those tools are adaptations of existing traditional debugging tools to make them usable for robotics, other tools have been specifically designed and developed for the use in robotics. The first section in this chapter presents instrumented tools which use low level system calls to hook to the debugging target. They allow developers to collect data without modifying the source code. As addition to the classic breakpoint approach, the more recent approach of tracepoints is presented. Tracepoints focus on data gathering without interrupting the execution of the debugging target, which is particularly important in robotics [2, 1, 8].

To use logging and print statements for debugging is still popular amongst developers [3]. Section 2.2 covers this basic form of debugging. Logging and print statements are the simplest form of debugging, often referred to as "output debugging" [1]. While ease of use and availability make this method attractive, it also has its downsides. It is an intrusive method which requires source code modification, type information is lost since everything is converted to a String and in bigger projects it is hard to keep an overview where logging and print statements were placed.

The data in robotic applications usually comes from sensors that capture a real world image of some sort. Common data in robotics is for example data from laser and sonar scanners. This data can be quite complex and difficult to understand in its raw representation, for example quaternions to specify 3D positions. In order to help the developer understand the data, graphical tools have been developed to support this task. Those tools help bridge the gap between the robot's perception of the world and the developer's understanding of the collected data. Section 2.3 presents some of these graphical tools, which influenced this work.

2.1 Instrumented Debugging Tools

2.1.1 GDB

The GNU Project Debugger (GDB) is a general purpose debugger, originally designed to debug C and C++ programs, but other languages have been added later on. The debugging tool "allows you to see what is going on 'inside' another program while it executes" [8]. GDB is open source and free software, released under the GNU General Public License (GPL). Although GDB is a highly sophisticated tool with many features, this section will summarize only the features that are most important for robotics. A more detailed overview of GDB's features can be found on the project website[1] and in the book "Debugging with GDB: The GNU Source-Level Debugger" [8].

The emerging market of embedded systems and robotics triggered the development of new features like the support for debugging remote targets and collecting data with tracepoints instead of breakpoints. These features are particularly important for robotics, since robotic applications are in general not interruptible and often mobile, which means the source code is executed on a different machine from where it is developed.

Remote Debugging

Some times it is impossible or infeasible to run GDB on the target platform itself. In robotics this might be due to limited processing power or due to a special operating system which does not support debugging. GDB can connect to remote targets running on different machines, if the program on the remote machine has been executed with `gdbserver`, an auxiliary program which implements a debugging stub. The debugging stub contains a set of subroutines which take care of the communication with the GDB host through a GDB-specific remote serial protocol [8]. Once the remote target is connected to the host, it can be controlled remotely: the developer can use GDB to specify breakpoints and control the execution of the program, almost as if it would run locally.

The debugging stub is specific to the architecture of the remote target. GDB

[1] http://www.gnu.org/software/gdb/

distributes a couple of already implemented debugging stubs for certain architectures, but if another target platform is used the stub needs to be re-implemented.

GDB Tracepoints

The most common technique used for debugging are breakpoints. They allow developers to specify at which point in the execution of a program they want the program to halt in order to have a closer look at variables and the state in which the program is currently in. As alternative to breakpoints, tracepoints were introduced to collect data without interrupting the execution of the program. GDB's `trace` and `collect` commands can be used to specify locations in the program code where data should be collected [8]. GDB internally handles tracepoints similar to breakpoints: they mark a location in the source code the developer wants to inspect to gain insight on the execution flow or the value of a variable. The data can be collected with arbitrary expressions which are evaluated when a tracepoint is hit, without interrupting the execution.

The collected data can only be analysed after the program has stopped its execution. The data is stored in a buffer on the remote target and represents snapshots of the data values that have been specified when the tracepoint was created. Each snapshot can be analysed closely using the normal GDB commands (e.g. `backtrace` and `print`) which are also used in traditional environments [8].

GDB tracepoints are currently only available for remote targets. If developers want to collect data with tracepoints in a local program, it has to be started with `gdbserver` and handled like a remote target.

2.1.2 Realtime Debugging Tracepoints

The GDB tracepoints approach presented above has one major downside: data can only be accessed and analysed after the program has stopped. This might be necessary for some systems, where transmitting the data over the network is too much overhead and can cause the system to misbehave. On the other hand there are systems that take a long time to properly start up and shut down. In such systems the GDB tracepoints approach will make development cumbersome and slow.

To counter this problem, a realtime debugging framework for robotics was developed [2]. The framework is originally based on the Player/Stage robotic middleware but can be used with other robotic frameworks as well, since it operates on a low level

and does not rely on specific features of a robotic framework or middleware [2]. It uses GDB to collect data when a tracepoint is hit, but transmits the data immediately to the user interface. The data can be accessed live during a debugging session and the developer does not need to wait for the application to stop to evaluate the data.

A plugin for the NetBeans IDE[2] integrated the novel debugging tool to make it easier to use. The user interface to set traditional breakpoints was re-used for tracepoints and presents a well known interface for developers. The user interface can be extended with further plugins to visualize the data collected during debugging. The first plugins created for this tool rendered laser and ultrasonic rangefinder data [2].

2.2 Logging

The previous section presented tools to collect data without the need to modify the source code. This section presents tools and methods from the other end of the spectrum: Using logging and print statements is a popular method to collect data during debugging. Although logging frameworks originally were developed for long term collection of data, they can also be used excessively during the development of a particular feature or module. Logging statements help developers to understand the flow of execution in a program and can be used flexibly to output data values during the execution of the program.

ROS is a widely used robotic framework [9] with a publish/subscribe communication middleware [10]. ROS supports developers of robotic systems in many ways and will be presented in more detail in Section 4.1, this section only presents the logging facilities in ROS as one representative for logging support in robotic frameworks. ROS modules can publish messages on so called topics to communicate with other modules. This communication principle is also used for the logging facilities in ROS. Apart from robotics there are also other more general logging frameworks available: For example Apache's Log4j[3] for Java and log4cxx[4] for C++. ROS internally uses log4cxx for the logging implementation in the C++ client library. The first part of this section gives an overview of the logging mechanism in ROS and the second part introduces the most basic form of logging: print statements.

[2] http://netbeans.org/
[3] http://logging.apache.org/log4j
[4] http://logging.apache.org/log4cxx/

2.2.1 Logging in ROS

ROS has a dedicated API for logging purposes, which facilitates the emission of log messages with different severity levels. The severity of a log message defines where the message is displayed: Debug and Info messages usually appear on the console where a ROS module is running (stdout), in the modules' log file and on the special purpose publish/subscribe topic */rosout*. Warn, Error and Fatal messages are displayed on stderr, stored in the log file and published to */rosout*. The log messages are human readable strings intended to tell the developer about the status of the module. Listing 2.1 shows an example how to log a message using Python.

```
rospy.logerr("The topic %s has no field with the name %s",
             topic, datafield)
```

Listing 2.1: ROS logging example in Python.

The implementation of the logging mechanism is slightly different in the two target languages C++ and Python. When logging is disabled, the emission of logging messages can be compiled out in C++ to improve the performance. Since Python is a scripted language, this is not possible and a different solution has been chosen: Debug messages in Python are only published on */rosout* if the module is specifically initialized with the log level set to debug.

Since all log messages are published on the */rosout* topic if the log level is set to debugging, it is easy to access them even if a module is running on a different machine. The graphical user interface rxconsole can be used to display and filter log messages by severity and full-text search [11].

2.2.2 Print Statements

Although print statements are not part of a proper logging "framework", they are often used in the same way as other logging statements. Print statements are quite popular amongst developers because no additional knowledge is required to use them and they are a core part of every major programming language and thus do not require to link external libraries. Print messages are simpler than logging statements, they print a String message to a console and have no feature for filtering or visualizing the printed data.

The use of print statements is a flexible way of collecting data during debugging. The major downside of print statements is that it is easy to loose the overview where print statements have been used. If they are not excluded from the build before the software is released or deployed, this can lead to a weaker performance of the software. Another problem with print statements is the need for the source code, to debug a program. If only the executables of a module are distributed, print statements can not be used for debugging.

The text-only representation of messages printed on the console restricts the use of print statements to the most basic types of data, since more complex data would be too hard to understand in its raw representation. This is especially true in robotics, since the data in robotics usually comes from sensors and complex algorithms. Despite the text-only limitation, developers can pre select what they want to print and can use the full power of the programming language they are using to aggregate values or compute simple metrics and properties for complex data which might be useful during debugging.

2.3 Graphical Debugging Systems

This section gives an overview of graphical tools to support debugging in robotics. Those graphical tools are often used to e.g. get a better understanding of the robots position in the real world and to visualize sensor readings and control values. Visualization is an important factor in robotics, because developers usually cannot interrupt the execution of a robot and take as much time as they need to understand data. They need to get as much information out of the data stream as possible in a very short time to detect problems and their causes.

2.3.1 LabVIEW

LabVIEW (Laboratory Virtual Instrumentation Engineering Workbench) is a graphical programming environment developed by National Instruments[5]. It features *Front Panels* which allow a developer to combine several widgets to a graphical interface. These widgets can display the status of the graphical program or act as control interface (buttons, knobs, etc.) for the program. For each widget on a panel an element is

[5] http://www.ni.com/labview

created in the block diagram of the program which can be connected to other parts in the block diagram. LabVIEW delivers a high number of pre-defined visualization widgets such as text displays, sliders and progress bars, which can be used to create a dedicated user interface for an application. This user interface can be used to control as well as monitor and debug the application. Figure 2.1 shows the LabVIEW user interface with a front panel in the background.

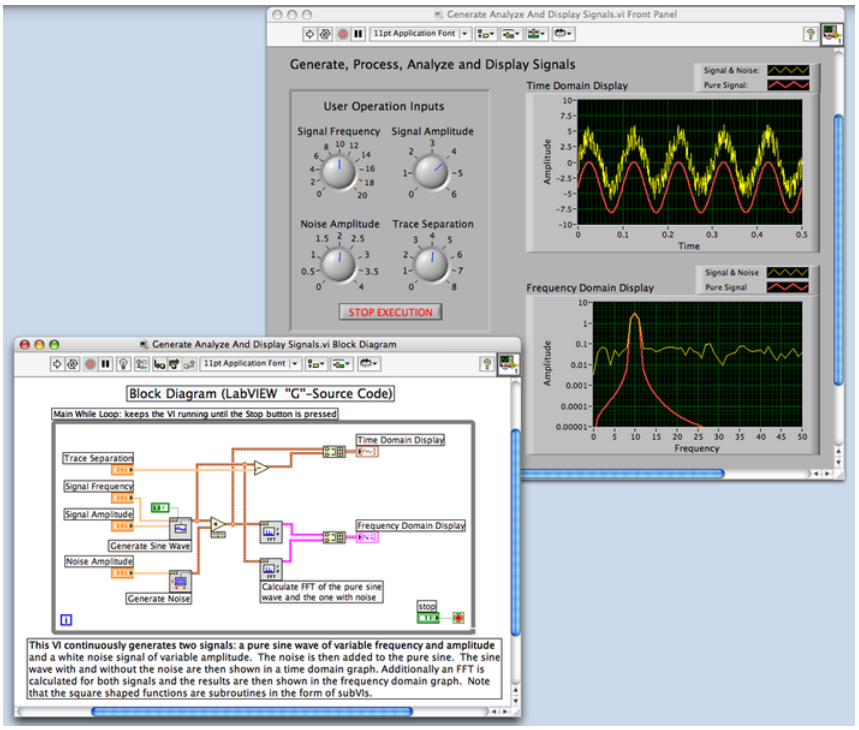


Figure 2.1: Screenshot of the LabVIEW user interface with front panels in the top right.

The front panel feature was described as "LabVIEW's biggest advantage," its "best feature," or its "main power" in the open-format answers of a survey of LabVIEW programmers [12]. It would be possible to design concrete front panels also for the purpose of debugging a specific application. This however requires integrating each front panel element directly into the visual program and is thus highly intrusive. Though a more transparent debugging possibility would be desirable, LabVIEW shows the potential of a flexible tool for visualization of abstract data in form of graphical widgets.

2.3.2 Augmented Reality Debugging System

ARDev is an Augmented Reality (AR) debugging system based on Player/Stage [7]. The AR approach to debugging helps the developer to understand the global state of the robot by augmenting a video feed of the robot with information of the robots' perception. Like the tools presented above, it focuses on pre-defined data such as laser and sonar sensors. The visualization of abstract data has been identified as possible future work [7]. Initial evaluation studies were promising that the visualization can help developers significantly during debugging [7].

As with most AR systems the set up of ARDev is quite complicated. A sophisticated tracking mechanism is required and some form of viewing device needs to be used. For ARDev a fixed viewpoint video see-through solution that tracks the robot with fiducial markers was chosen [7]. The single camera and viewpoint approach ARDev uses is sufficient for a small room, but might not be sufficient for different application areas. Thus the choice which AR technology is used highly depends on the targeted environment.

2.3.3 RViz

Most of the currently available robot frameworks have their own tools for data visualization, this section presents RViz as representative for the class of visualization tools in robotics. RViz is a 3D visualization tool for ROS. Other visualization tools with similar features are playerv from the Player/stage framework [13], OrcaView2d and OrcaView3d from ORCA [14] and robotgui from CARMEN [15] which provides a more general user interface.

RViz[6] is a highly sophisticated graphical interface to render 3-dimensional data. The tool visualizes a 3D model of the robot (Figure 2.2) and can visualize additional data like point clouds (Figure 2.3), maps, robot poses, trajectories, etc. [10]. During the development of robot applications, it is hard to understand how the robot perceives his surroundings, because the accuracy of the robot's sensory data can vary under different circumstances. This leads to problems during the execution which are hard to identify, reproduce and investigate due to the indeterministic nature of robotic applications. RViz helps developers to understand how the robot sees the world and makes it easier to understand what caused a problem.

[6] http://www.ros.org/wiki/rviz

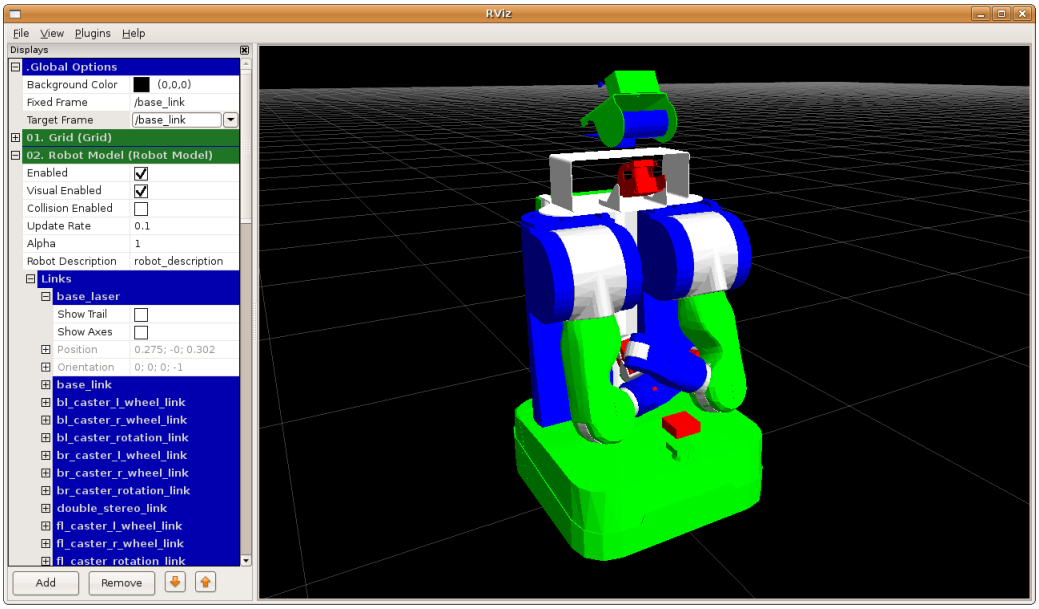


Figure 2.2: 3D model of a PR2 robot in RViz.
Available under a CC BY 3.0 license at http://www.ros.org/wiki/rviz/DisplayTypes/RobotModel

RViz requires a 3D model with the exact proportions of the robot to visualize the robot and its movement correctly. The data visualized in RViz is mostly pre-defined and collected from known interfaces such as laser sensors, mapping algorithms and computer vision modules. RViz makes extensive use of the existing communication channels in ROS since the same data is also often used as input for other modules in a robotic system and can be re-used. Although it is possible to visualize arbitrary data in RViz, it is hard to find a good place to visualize the data without occluding valuable information in the 3D view. Arbitrary data does not have to be directly related to the real world, it can also be intermediate values from a computation algorithm which can be hard to position in the 3D view.

The plugin functionality in RViz can be used to extend RViz with further visualizations and modules. The current plugin interface is not documented yet, since it is under active development (as of September 2012) and the plugin API is expected to change significantly before it will be released with the next major ROS release planned in late 2012 [16].

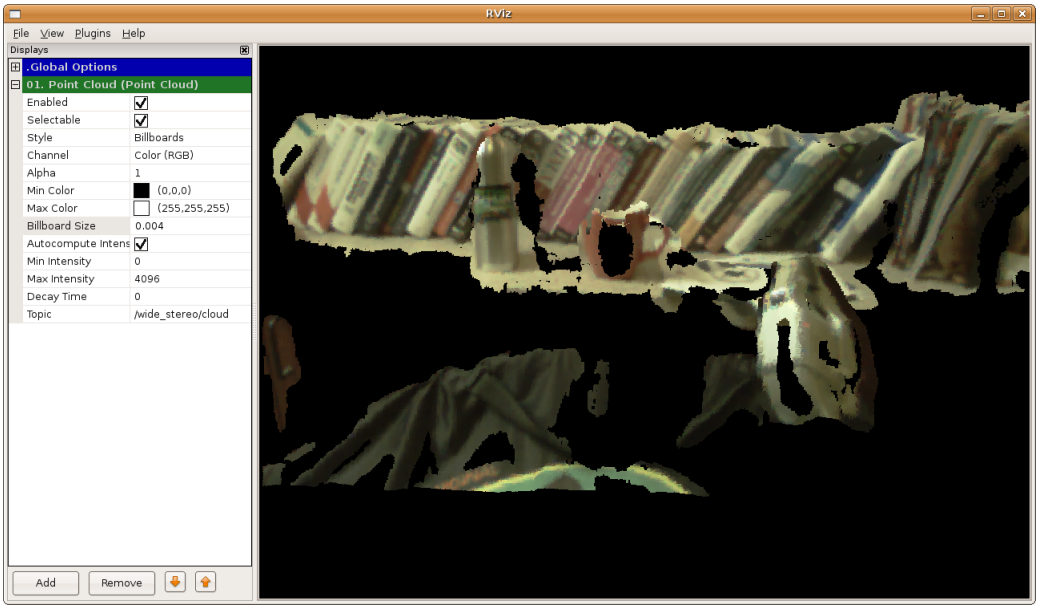


Figure 2.3: Point cloud visualization in RViz.
Available under a CC BY 3.0 license at http://www.ros.org/wiki/rviz/DisplayTypes/PointCloud

2.3.4 rosbag and rxbag

In a complex robotic system there is usually a lot of communication happening at once. Trying to keep track of the different topics and messages is not an easy task and recording the messages for analysis at a later stage is necessary. rosbag is a command line tool to dump communication data from selected topics to a file for later use or to analyse the communication after the application has terminated. Listing 2.2 shows an example usage of the rosbag tool to record in this case the two topics */foo* and */bar*. The data is stored in a so-called bag-file, which contains all the messages published to selected topics during the runtime of the application. rosbag can be used to play back the recorded messages in the same order and with the same time offset as they were recorded. This can be used to examine the behaviour of one particular subsystem without the need of running the full stack of subsystems in a real environment.

```
rosbag record foo bar
```

Listing 2.2: Example usage of rosbag.

The bag-file can also be used to examine a flow of events in the system after the execution has terminated. rxbag is a graphical tool to analyse bag-files, it visualizes the content of a bag-file on a timeline. The tool has controls to play back, pause and rewind the stream of messages, which allows to have a closer look at some events on the timeline. rxbag visualizes image messages as thumbnails on the timeline which makes it easier to understand what the robot was facing when the messages were recorded. Developers can use rxbag to inspect messages in more detail, since the raw messages were recorded and can be accessed from the tool. Figure 2.4 shows a screenshot of rxbag's user interface.

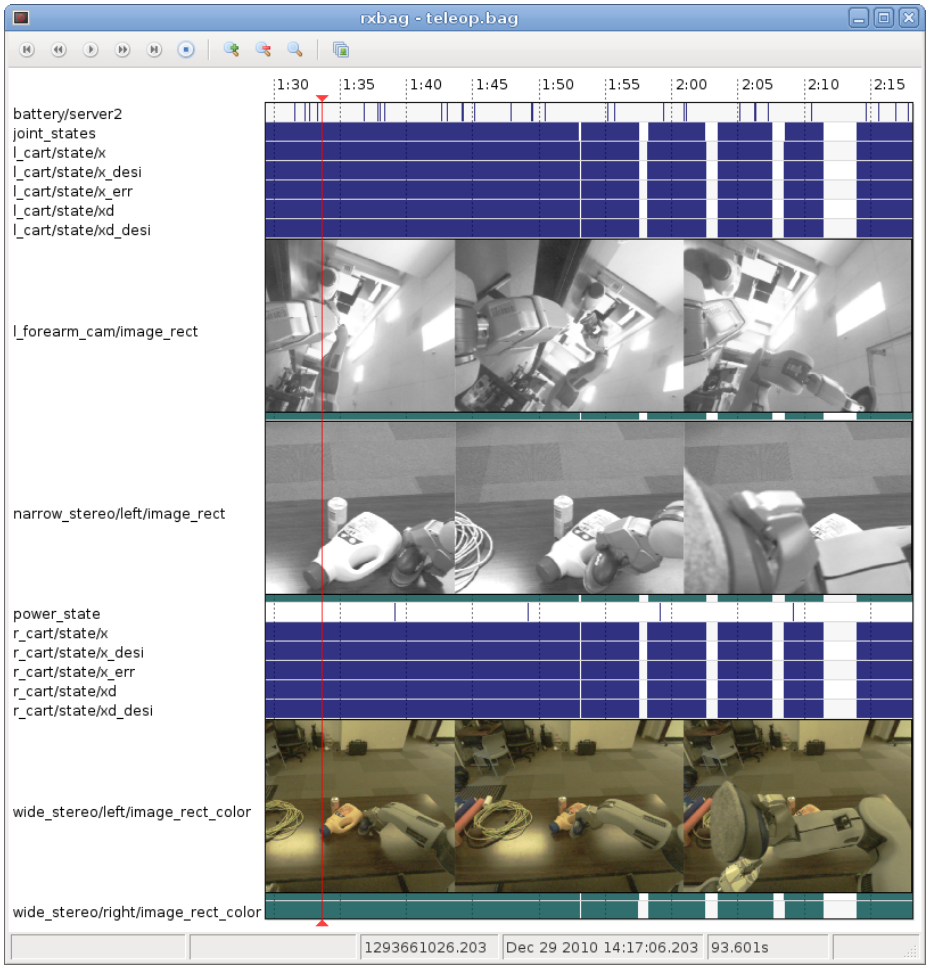


Figure 2.4: Screenshot of the rxbag user interface.
Available under a CC BY 3.0 license at http://www.ros.org/wiki/rxbag

While rxbag mostly focuses on replaying recorded messages from a bag file, it can also be used to visualize the data stream while it records it. This can be done with the command line option *–record*. rxbag can be extended with more visualizations through a plugin API. The plugin API is still experimental and under development, but it should become more stable in future versions of ROS [17].

2.4 Summary

The tools presented in this chapter cover a wide range of topics and can be categorized into two families: tools to collect data and tools to analyse and visualize data during debugging. While GDB collects data for later post-mortem analysis, the tracepoints approach in the realtime debugging system can collect data and display it live. GDB has no tools to visualize the data collected and developers have to examine the data manually. Although the tracepoints approach from the realtime debugging system [2] could be used, many developers still rely on "printf-style" debugging or other logging mechanisms. The ad-hoc logging approach is much simpler and does not require knowledge of external tools, but the source code must be modified which can lead to so-called "Heisenbugs" [5].

The data collected with print or logging statements is usually text-only, which requires the developer to constantly parse and interpret logging messages. Due to the large amount of data that is processed this often means developer consoles are filled with logging messages that sometimes contain more complex content such as quaternions for 3d coordinates, laser scans and point clouds. Because a lot of data is collected and the data is too complex to understand in its raw representation, visualization tools have been developed for specific purposes to bridge the gap between the collection of data and the developer's comprehension of data.

The visualization tools available for robotics usually visualize pre-defined spatial data collected from known interfaces like laser sensors, mapping algorithms and computer vision modules. Most of them can be extended with plugins for more visualizations, but the user interfaces are rather static and difficult to adapt to different scenarios. The tools were not developed to visualize arbitrary and abstract data which is often used for debugging.

3D visualizations such as rviz and ARDev are used during debugging to understand the robot's view of the world. Both of them require a substantial amount of set up work. For rviz a 3D model of the robot must be created, which has the exact proportions of the real robot. ARDev requires an even more complex setup for AR. During

the development of the tool an intelligent debugging space (IDS) was permanently installed in a laboratory, where the robot under development can be constantly tracked with markers and cameras [7]. This setup is often unavailable, especially in outdoor situations, and is also often not applicable for debugging tasks where only abstract data is available or required. A general problem with 3D visualization tools is that abstract data often does not fit into the 3-dimensional rendering and in the case of AR can hide important information of the real world [7].

3 Design of a Flexible Visual Debugging System

Debugging in robotics has unique requirements which are often not met by traditional debugging systems. The special requirements lead to a number of different tools to support debugging in robotics. The previous chapter summarized some of those tools and issues of existing solutions were identified: a) The tools either focus on visualization of pre-defined spatial data or render data as text messages. b) The graphical interfaces are rather static and difficult to adapt to different usage scenarios.

This chapter introduces a new system for debugging in robotics, tailored to tackle the challenges developers face when debugging robotic systems. The system combines the ease of use and flexibility of logging statements with a graphical visualization. The visualization makes it easier to comprehend the large amount of data collected during debugging of robotic systems.

A possible reason why many developers still use print or logging statements to debug their programs is because these methods can be immediately used when required. Developers do not need to undergo a lengthy setup and configuration process before they can start debugging and they do not need to gain extensive knowledge about the methodology and tools either. The overhead to set up and configure a debugging system should be reduced to make the system easy to use without extensive knowledge about the debugging methodology and tool.

The main issue with print or logging statements is the text-only representation of data. The system proposed in this chapter aims to support the developer during debugging by visualizing data in a graphic way and thus eliminate the cognitive effort needed to parse and interpret text based logging messages. The cognitive effort required during debugging can be further reduced by a flexible system which can be adapted to different preferences of different developers. Every developer can choose a visualization that fits their mental model of the debugged data best.

The chapter introduces the goals of this work in the first section. Based on the goals specific requirements for this system are identified in Section 3.2. The system design section presents the design for a flexible visual debugging system in the last section of this chapter.

3.1 Goals

The main goal of this work is to design a debugging system which is suitable for debugging in robotics and can be used to evaluate the hypothesis stated above. The design of the system should be independent of a specific robotic framework so that it can be implemented for any available framework. While most of the currently available visualization tools in robotics focus on pre-defined and spatial data to help understand the robot and the environment in which it runs [7, 10], rendering of abstract data is still uncommon. The design of the debugging system should be flexible enough to allow visualization of arbitrary and abstract data. Although abstract data will be the main focus of the debugging system since tools to visualize pre-defined data are already widely available, the designed system should not be restricted to abstract data.

The system should provide a graphical user interface which can be used by developers to visualize all kinds of data from the robotic application. The developers should be able to customize the visualization according to the current robot hardware, development stage and personal preferences. The system should be adaptable to many different use cases and should reduce the cognitive effort during debugging by visualizing the data according to the mental model of the developer and meaning of the data.

To preserve the configuration of visualizations a developer is using, it should be possible to save the state of the visual debugging system to file. This file could be used to continue working at the same point where work was stopped previously. It could also be used to share the visualization configuration amongst developers working in the same team.

3.2 Requirements

The requirements for the flexible visual debugging system are mostly dominated by the special requirements of robotics. Some requirements are derived from the hypothesis and focus more on development performance and speed. This section presents the elicited requirements for a flexible visual debugging system.

3.2.1 Distributed Live Debugging

Robotic applications are usually distributed systems, because they often run on mobile robots or are so complex that multiple computers are used to distribute the modules across multiple machines and thus improve performance. This means the system must be able to handle communication distributed on a network. There are many different robotic frameworks currently available and the design of the debugging system must account for different middlewares and possible changes in the future. It is out of scope for this work to make the debugging system compatible with multiple robotic frameworks at once, but it should be considered for the architecture of the debugging system. Since the execution can not be easily interrupted to collect data, the debugging system has to visualize all the data live.

3.2.2 Adaptable Tool

Due to many different application scenarios in robotics and the diverse environment of available frameworks for robot development, many researchers and developers have built their own tools to support them during debugging [7]. Developing your own debugging tool is extremely time consuming and the developed tools are often one-time-only tools, because they do not fit the use case of other applications and are too hard to adapt to a new project. In order to allow the use of the proposed debugging system in many different scenarios and use cases, flexibility has a high priority. Flexibility not only means the system can be adapted easily to fit different problems, it also means the system can be adapted to suit different developer's preferences. Each developer might prefer a different kind of visualization of the collected data, thus loose coupling of the collected data and the visualization realization is necessary.

3.2.3 Low Configuration Overhead

Debugging robotic applications is a highly iterative process, where small changes are made and immediately deployed to the robot. When new visualizations are needed or old visualizations need to be updated, the configuration overhead should be as minimal as possible. The configuration task should not distract the developer from the problem analysis task.

3.3 System Design

The design of the developed system reflects the requirements identified in Section 3.2. The system is designed to run in a distributed environment and visualize data live, as it is collected and not post-mortem. The system design is not tailored to a specific robotic framework: Since the underlying component based architecture is similar in many modern robotic systems [10, 13, 14, 15], implementing the system for any robotic system is possible. The need for an adaptable system influenced especially the object design, which takes into consideration the future extendibility and adaptability during the debugging process.

This section shows how the system is designed according to the requirements: The user interface design, the system architecture and the object design are presented. The user interface is presented first, because the concept of a developer dashboard influenced both the architecture and the object design.

3.3.1 Graphical User Interface

In order to be adaptable to many different use cases and visualization preferences of developers, a central dashboard approach was chosen for the user interface. The dashboard allows developers to add and remove visualizations easily and arrange them however they want. The visualizations are wrapped in widgets and can thus be positioned freely on the dashboard. Adding new visualizations to the dashboard can be done through a "Drag&Drop" mechanism. Once the widget is on the dashboard it can be resized and repositioned on the canvas. The initial mockup of the proposed graphical user interface is shown in Figure 3.1.

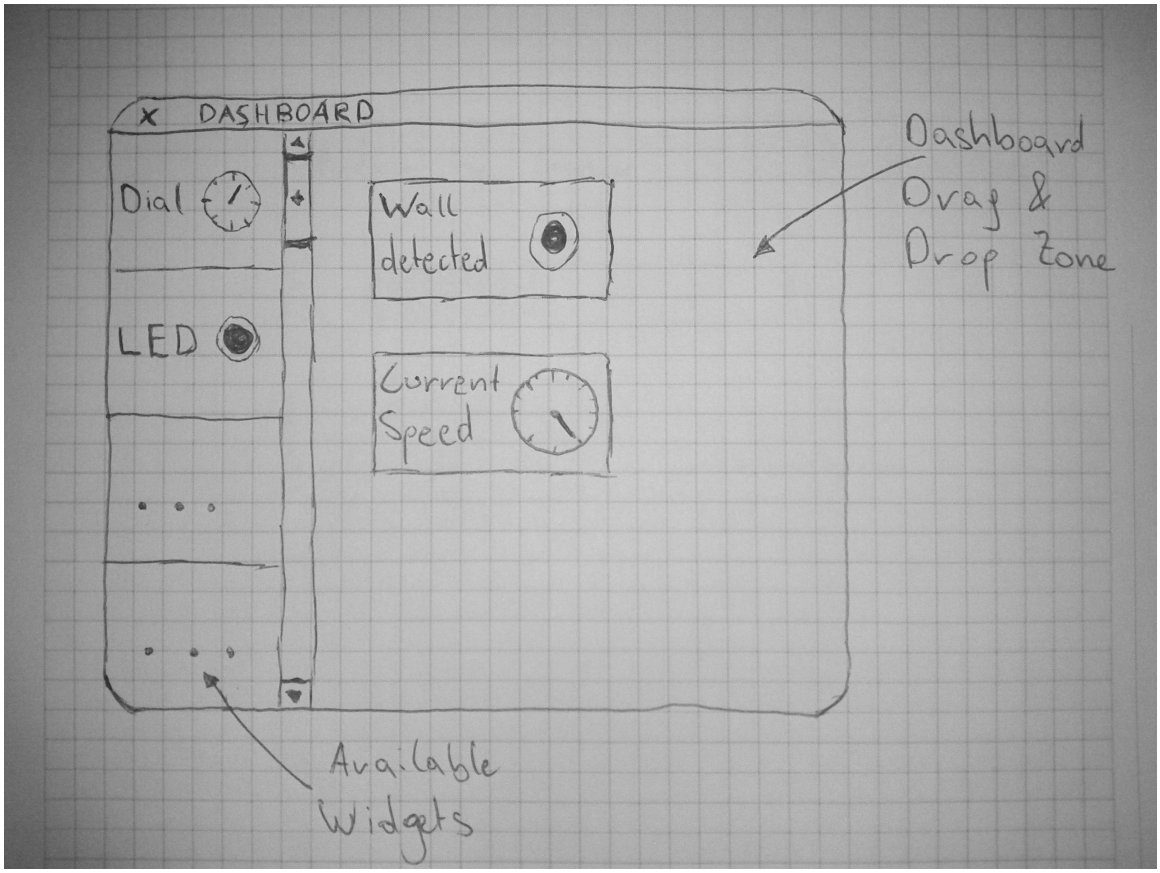


Figure 3.1: Paper mockup for the graphical interface.

Each item in the list on the left (Figure 3.1) represents one type of visualization, this can be as simple as a dial for numeric values or more complex visualizations for more concrete data like a map visualization. Although the design of the user interface does not restrict the types of visualizations, it is intended for simple and abstract data, since other tools like RViz (see Section 2.3.3) are more specialized for complex and spatial data required in other use cases.

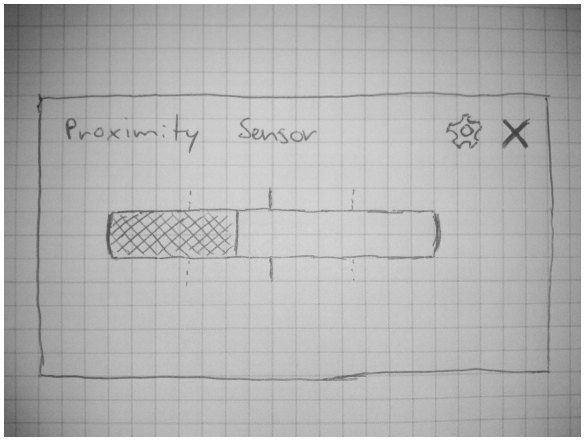


Figure 3.2: Paper mockup for a visualization widget.

Figure 3.2 shows an early mockup of a visualization widget on the dashboard. This exemplary visualization widget has the form of a progress bar where numeric values can be visualized. Dials, meters, thresholds, compasses and LEDs could be other simple visualizations widgets. A more complex example for a visualization widget is the laser scan visualization as shown in Figure 3.3.

3.3.2 Architecture

The currently available robotic frameworks mostly rely on a communication middleware that abstracts from the concrete communication channels. This communication infrastructure can be used by the proposed debugging system to access data without the need to develop a dedicated communication layer for debugging.

The user interface mockup shown in Figure 3.1 already indicates the modular design of visualizations that can be added to the dashboard canvas. The modular approach makes it easy to extend the system with more visualizations in the future and it gives developers the choice how data is visualized. Each visualization widget contains a

Figure 3.3: A possible laser visualization, as shown in Luke Gumbley's paper on realtime debugging (Fig. 7). [2]

different form of visualization. This visualization widget handles the graphical part of the visualization, stores information about the data connection and other settings and encapsulates the communication with the middleware. The connection with the communication middleware is established through an adapter which notifies the widget if the data has changed or new data is available. This makes it easy to exchange the communication adapter if the communication protocol changes or the implementation targets a different robotic framework which has a different communication structure. This reduces the coupling between the visualizations and the data collection. Although the dashboard needs to know which widgets are currently on the dashboard, the widgets itself are independent of the dashboard. The decentralized approach allows easier integration of a plugin infrastructure since the dashboard itself does not need to be modified to show additional visualization widgets.

Since the visualization widgets can connect directly to the communication middleware, existing data used for inter-module-communication can be visualized as well as dedicated debugging data. What kind of data is visualized is irrelevant for the visual debugging system and allows to also use it as pure monitoring tool to keep track of existing communication data between modules. Figure 3.4 shows how different robot modules communicate with each other through the communication middleware. The adapter instances in the visualization widgets can connect to the communication using the same principle used for inter-module-communication. It is transparent to

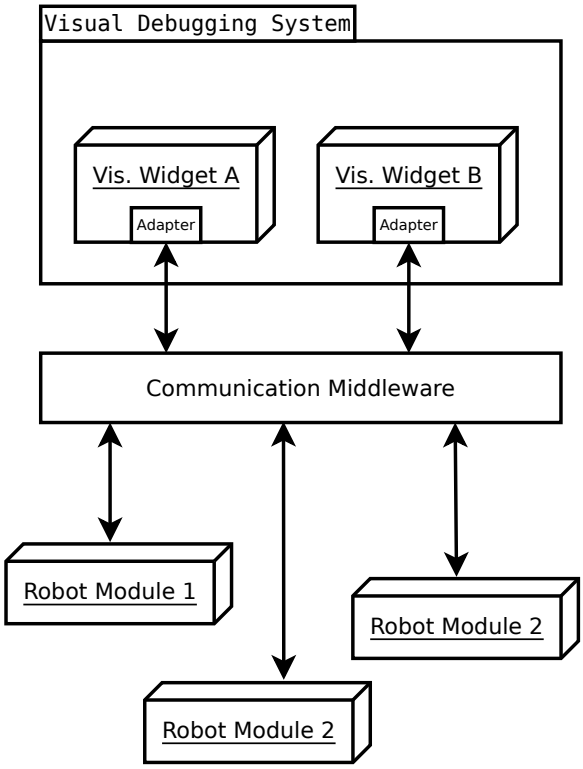


Figure 3.4: Communication diagram

the visualization widgets whether the visualized data comes from eavesdropping on communication between modules or is dedicated debugging data.

ROS has been examined as an example for a robotic framework with a communication middleware, since some of the ROS tools were already presented in Chapter 2. The publish/subscribe style communication in ROS successfully decouples the sender from the receiver of messages [18]. Since this principle also appears in other frameworks it should be simple to implement an adapter that connects to the communication middleware of a specific robotic framework. If the communication structure is fundamentally different in the target robotic framework, some more work might be required to decouple the visualization from the data collection.

3.3.3 Object Model

The object model shown in Figure 3.5 was designed to make it easy to extend the debugging system with more visualization widgets. The abstract **DashboardWidget** class implements the general methods that are the same for every widget. Its internal method structure allows subclasses to overwrite only some parts of functionality, without the need to rewrite most of the other methods. This allows easy integration of a plugin framework at a later stage, where third party widget developers only need to implement the specific parts of the new widget they want to provide and the common parts are handled by the default implementation in **DashboardWidget**. The class **ConcreteVisualizationWidget** stands for a possible visualization widget that implements the abstract methods from **DashboardWidget**. Each visualization widget has to subclass **DashboardWidget** to make use of **DashboardWidget**s default implementation framework.

The data provider setup and the properties management are also implemented in the **DashboardWidget** class, because they will likely be the the same for most widgets. The main reason is to hide the technical details from a third party plugin developer to make his life easier. The plugin developer only needs to specify which properties his widget has in `initProperties()`, the default implementation of `updateProperties()` saves the new properties and notifies the widget to update the user interface through `updateWidget()`. The **DashboardWidget** automatically supports numeric, text and float properties. If more specific properties are needed the plugin developer can overwrite the properties-specific methods in **DashboardWidget** to provide an implementation tailored to this specific widget.

The **Adapter** class in Figure 3.5 can be seen as a black box that takes care of the communication with the communication middleware from the respective robotic

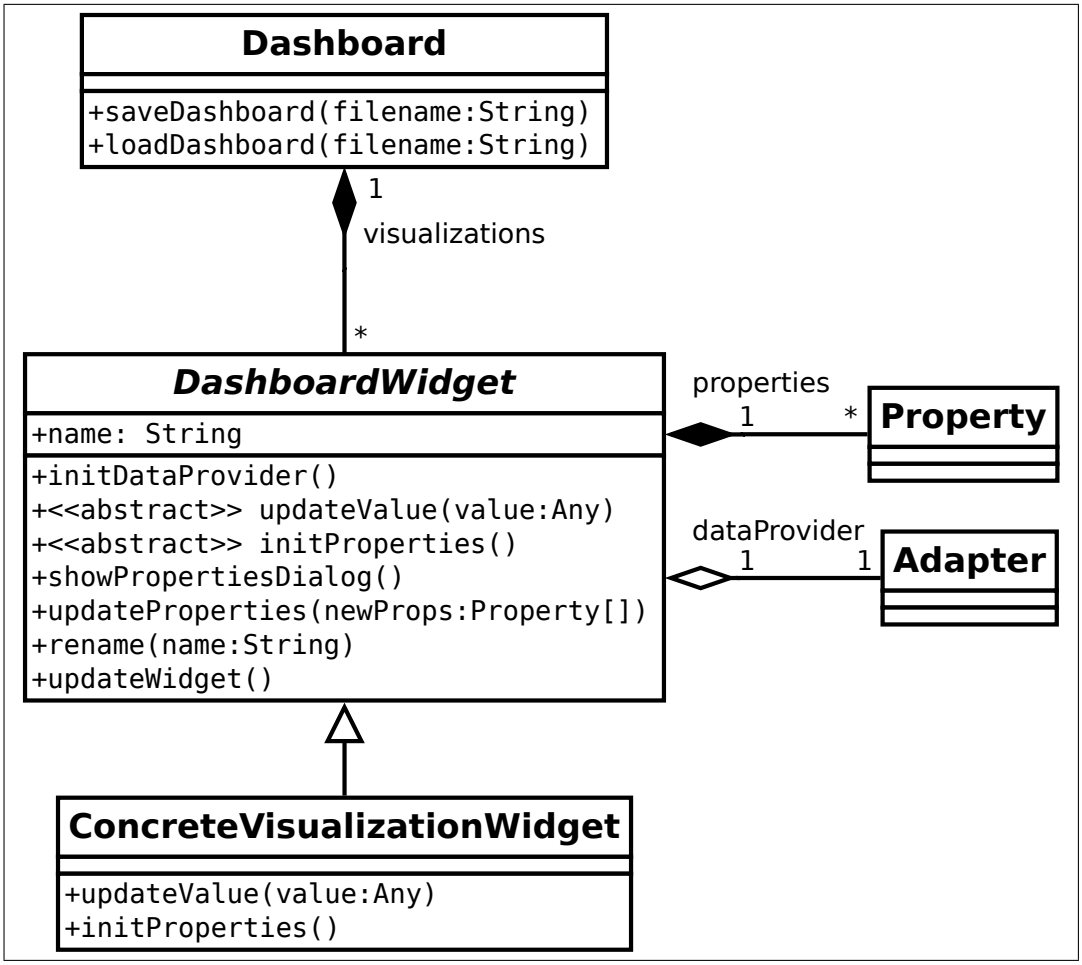


Figure 3.5: Extendible object model.

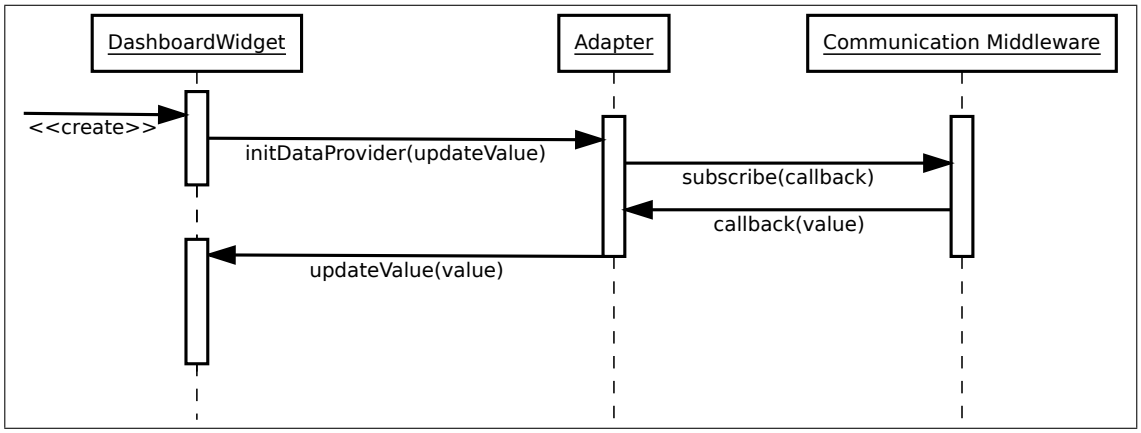


Figure 3.6: Decoupling the user interface from the application logic with callbacks.

framework. The **Adapter** connects to the communication middleware and waits for new data. If new data is published it notifies the visualization widget to update the value of the widget through `updateValue(value: Any)`. The diagram in Figure 3.6 shows how the `updateValue` method gets passed to the **Adapter** upon initialization to decouple the user interface from the application logic. When the **Adapter** receives a new value through the publish/subscribe communication middleware, it passes the value on to the **DashboardWidget** which updates the user interface.

3.3.4 Plugin Architecture

The architecture and object model presented in the previous sections aim to make it easy to extend the visual debugging system with further widgets. To allow third party developers to add their visualizations a plugin architecture will be presented in this section. The plugin architecture allows to add new visualization widgets in form of plugins, without the need to modify the source code directly.

The diagram in Figure 3.7 shows the object design of the plugin architecture. Plugins can be loaded and unloaded through the **PluginManager** class. The **PluginManager** class keeps a list of all plugins and populates the toolbox where plugins can be selected from (see Figure 3.1). The plugin itself contains the implementation of a visualization widget which is subclassed from **DashboardWidget** and some additional meta information about the plugin.

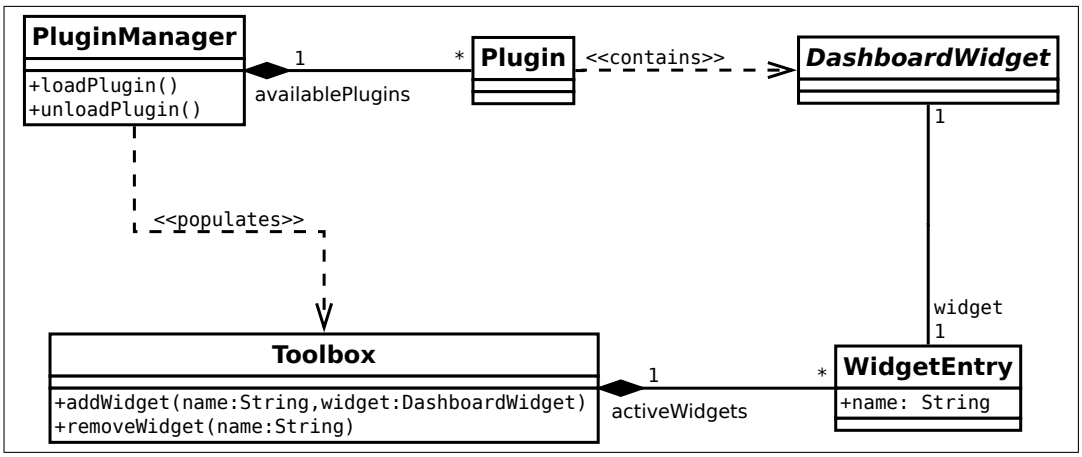


Figure 3.7: Plugin architecture overview.

4 ROSDashboard: A Visual Debugging System for ROS

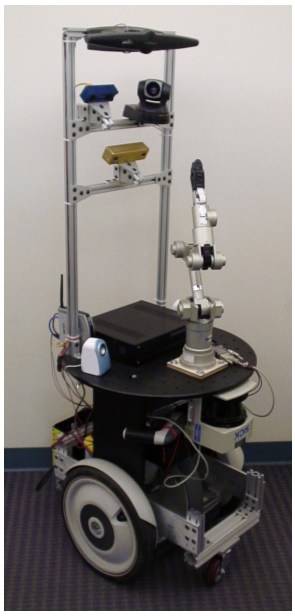
ROSDashboard is a prototypical implementation of the system designed in Chapter 3. The implementation makes use of the ROS publish/subscribe communication infrastructure to gather data and visualizes the data published by the respective ROS nodes. The data can be taken either from existing node-to-node communication or from explicit logging statements that publish data for visualization only.

Although the design in Chapter 3 does not imply the use of a specific robotic framework, some of the architectural decisions were driven by the ROS architecture which was examined during the initial survey of existing debugging tools and practices in robotics. Especially the callback-style to get new values from the communication middleware to the user interface (see Figure 3.6) was inspired by ROS. As a result, the implementation of ROSDashboard did not require a lot of glue code to connect to the communication infrastructure. If another robotic framework is chosen as target platform for an implementation, more work might be required to access the data from the robotic application. This highly depends on the existing communication infrastructure of such a robotic framework and whether it can be accessed transparently without the robot module knowing about the visualization.

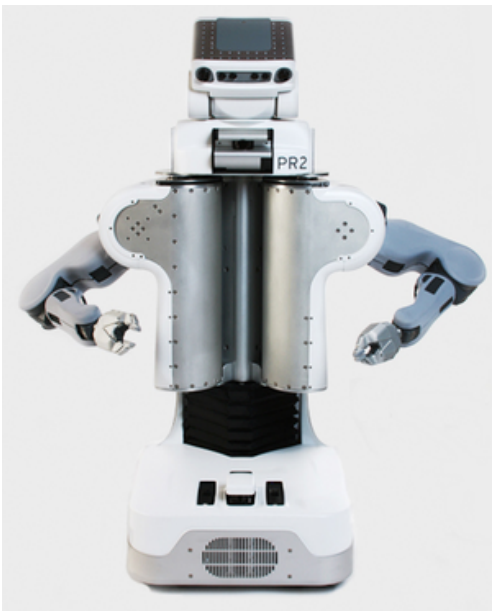
This chapter gives a detailed introduction to ROS with the tools related to debugging and visualization, which influenced the development of ROSDashboard. The second section in this chapter presents ROSDashboard's implementation details, ranging from the user interface level down to the ROSDashboard API and the topic introspection that makes the use of ROSDashboard easier.

4.1 ROS (Robot Operating System)

The ROS (Robot Operating System) is an Open Source framework for complex robotic systems. The first work on ROS was done as part of the STanford Artificial Intelligence Robot (STAIR) in 2007 [19]. The original software library was called *Switchyard* and had been developed at Stanford University. Later the library was refined and generalized to also suit the requirements of the Personal Robot Program at Willow Garage[1] [10]. The resulting general framework has been released as Open Source [10] and this section gives a short overview of the most important principles in ROS.



(a) Stanford's STAIR



(b) Willow Garage's PR2

ROS has grown significantly in the last years with an active community backing the project and the support for many of the currently available robots [9]. It was developed to abstract from the hardware of the robot and make it easier to create modular robot software which can run on different robots and on different machines. The modular approach makes development easier, because the work can be divided amongst different developers or development teams. This also allows the developer to

[1] www.willowgarage.com

change only small parts of a complex system, without the need to build and re-deploy the whole system. The modules in ROS are called *nodes* and several nodes executed together are called a *stack*. ROS *packages* bundle nodes and stacks and are used to make software modules available to other developers. Everyone can create their own package which can be indexed by ROS so that their software modules can be found, downloaded and used by other developers. There are many packages, nodes and stacks available bundling implementations of algorithms for some of the most common problems in robotics (e.g. navigation, localization, joint movement, etc.) and they can easily be (re-)used [20].

The communication between ROS nodes is either asynchronous through a publish/subscribe mechanism or synchronous through services. With the asynchronous approach nodes can send messages by publishing a message on a topic and receive messages by subscribing to that topic. This mechanism is flexible and decouples the sender from the receiver: A publisher node does not need to know if there are other nodes listening and vice versa. The routing for those publish/subscribe messages is established during runtime through the ROS core. For synchronous communication and guaranteed delivery of messages, services can be invoked. Since the communication with services couples the sender to the receiver, services were not a relevant data source for ROSDashboard. The communication between nodes is one of the main sources for debugging data when debugging a ROS application. The same communication framework is also used for the logging mechanism in ROS, which publishes messages to the special purpose topic */rosout* (see Section 2.2.1).

ROS was chosen as a target platform because it has become a stable and popular robotic framework in recent years [9]. Many tutorials and code samples made it easy to learn how to program for ROS and the ROS community was helpful and quick to respond if questions arose during the implementation of ROSDashboard. This includes help for ROS beginners as well as valuable feedback on the design and implementation of ROSDashboard. The publish/subscribe mechanism in ROS provides an easy communication layer that can be accessed using the ROS client API. This communication style allows ROSDashboard to easily connect the visualization widgets to the data source without the source knowing about the visualization. Since ROS is a modular framework and encourages to encapsulate functionality in several small nodes rather then a single big node, the data on communication channels between existing nodes can be accessed and visualized. ROSDashboard fits well into the existing ROS tool suite and fills the gap between visualization of complex data in RViz (see Section 2.3.3) and text only debugging with the ROS logging mechanism (see Section 2.2.1). This section presents further ROS tools which were not introduced in Chapter 2 since they are not debugging specific.

4.1.1 Related ROS Tools

While most of the initial tools developed for ROS were mostly command line based, many graphical tools have been developed recently. This surge of graphical tools shows that ROS has become more wide spread and developers started working on ROS projects which are probably not as familiar with command line tools as the core developers of ROS are. Most of the graphical tools were created to help developers understand the data flowing back and forth between modules and the topics through which the data was transported. This section only presents the tools which were not already introduced in Chapter 2.

`rxplot` is a graphical tool which can plot values from topics on a Cartesian coordinate system (see Figure 4.1. The tool takes data from a published ROS topic and prints it on a time graph. The tool can be configured to visualize several graphs in one go, which makes it easy to compare values and data streams.

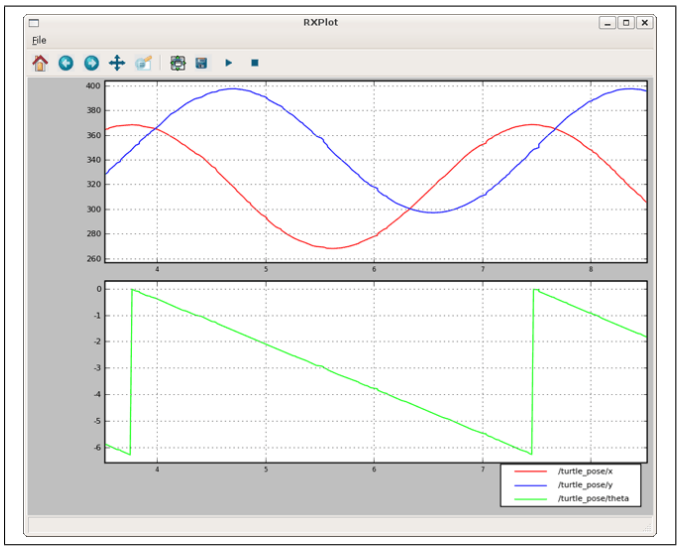


Figure 4.1: rxplot screenshot. Available under a CC BY 3.0 license at http://www.ros.org/wiki/rxplot

`rxconsole` provides a graphical interface for logging messages. ROS logging messages are published on the special purpose topic */rosout* and can be viewed either with a normal console or with the graphical rxconsole interface. The graphical interface can filter logging messages by text and by severity (see Figure 4.2).

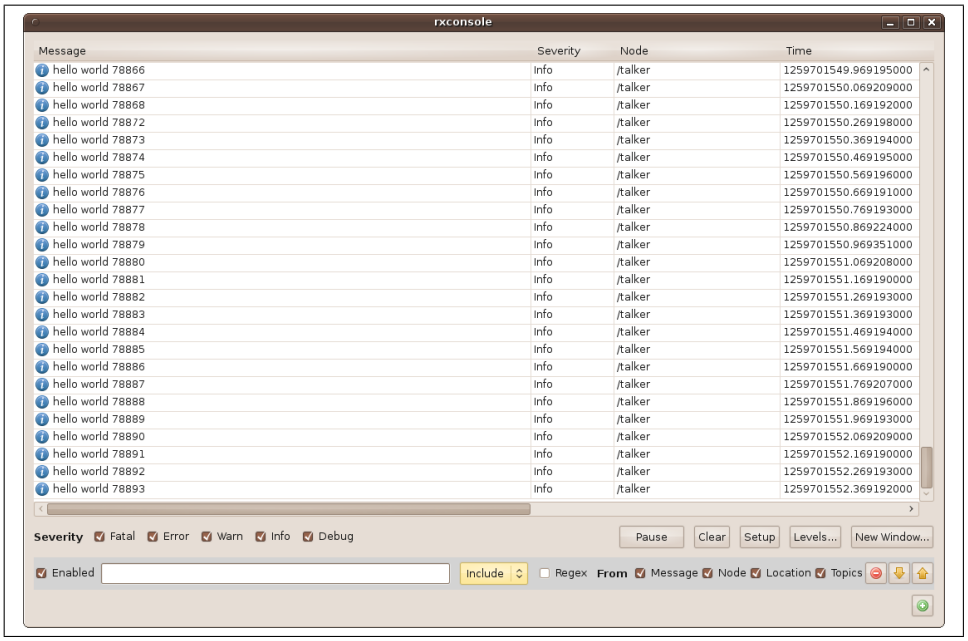


Figure 4.2: rxconsole screenshot. Available under a CC BY 3.0 license at http://www.ros.org/wiki/rxconsole

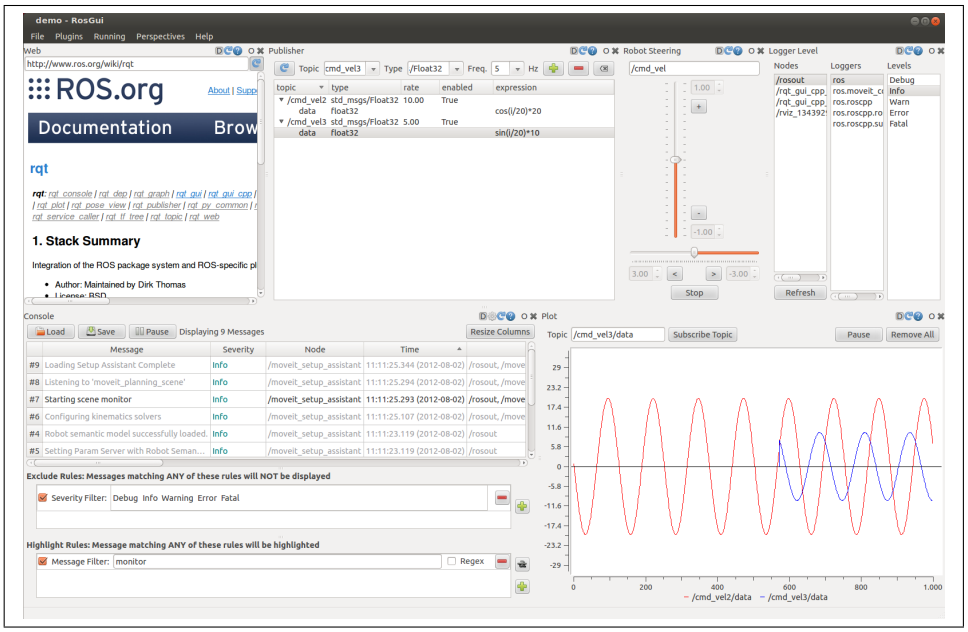


Figure 4.3: rqt screenshot. Available under a CC BY 3.0 license at http://www.ros.org/wiki/rqt

`rqt` (formerly rosgui) is a project that aims to unify the graphical user interface for ROS tools. It provides a hull where other tools can be loaded through plugins (see Figure 4.3). rqt's interface is based on the Qt window toolkit which is currently the recommended window toolkit for ROS. The interface provides a single point of access to all other graphical ROS tools by exposing them as plugins. The plugins can be enabled and positioned inside rqt. This allows developers to create perspectives for special use cases, where a set of graphical ROS tools are used. rqt manages the life cycle of the plugins by exposing a plugin API which has to be implemented by every plugin. When a perspective is saved the configuration of every plugin is saved as well. This makes it easy to pick up work where left off.

ROSDashboard fills the gap between highly sophisticated visualizations in `RViz` and `rxplot`, which only offers one kind of visualization. It distinguishes itself from other graphical tools by giving the developers full flexibility to choose themselves what visualization they prefer. `rqt` aims to bring together all the different graphical tools in ROS. ROSDashboard deliberately does not offer a record and replay functionality, since this is the main functionality of `rosbag`. ROSDashboard can be combined with `rosbag` in `rqt` to give developers the possibility to use record and replay together with visualizations. Although `rxbag` already contains some support for visualizations, it mainly focuses on replaying data and looking at data in detail after the execution has stopped. This distinguishes `rxbag` from ROSDashboard, which focuses on visualizations during the execution of a program.

4.2 Implementation Details

This section presents the implementation details of ROSDashboard. The first part shows in detail how the visualization widgets are connected to a ROS topic and what difficulties had to be resolved. Part two of this section presents the user interface details and shows screenshots of ROSDashboard in action. The third part introduces the data format in which dashboards can be saved and restored from a file. The ROSDashboard API is a set of convenience methods to publish data on topics and is shown in the last part.

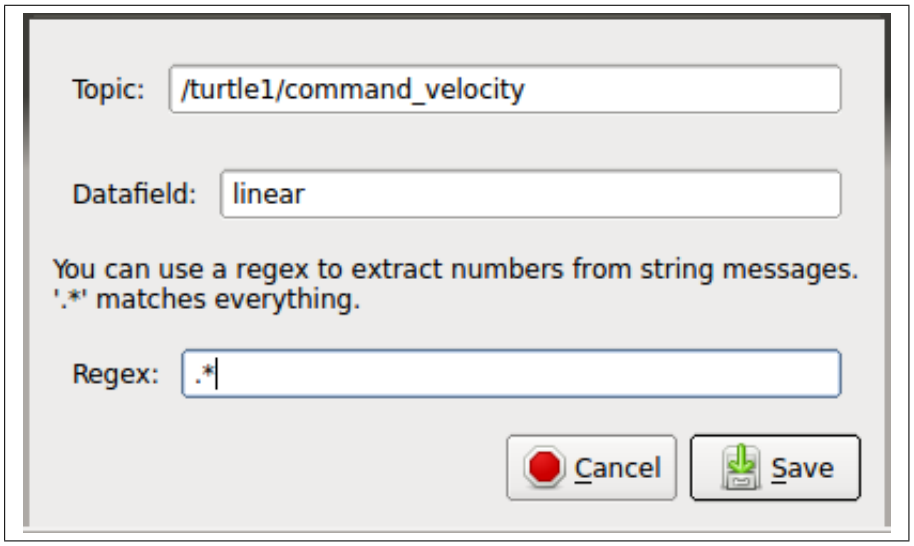


Figure 4.4: Screenshot of the topic setup dialog.

4.2.1 Topic Introspection

ROS topics were originally not designed and developed as something the user or developer chooses graphically: They are usually created, configured and used in the source code. ROSDashboard exposes the topic setup in a graphical user interface every time a new widget is added to the dashboard. To make this as easy as possible and without much overhead, a technical solution was chosen to reduce the number of fields to be set during the topic subscription setup. Normally you have to select a topic name and a message data type. The data type can be one of the standard message types like Float, Integer, String and Boolean or a more complex message type which contains more information in a structured message. To access one data element of a message the "datafield" field was introduced in the graphical interface. Figure 4.4 shows an exemplary topic setup configuration to access the linear velocity of the */turtlesim/Velocity* message published to the topic */turtle1/command_ velocity*. Using Python's duck typing and the *rostopic* module it was possible to avoid the complexity of dynamically binding message type classes during runtime and detect the message type automatically. If a topic is not yet published and thus the message type of this topic is not defined yet, the method call to *rostopic* will block until the message type becomes available. To avoid blocking of the user interface a listener thread was implemented to wait until the message type for a topic becomes available (see Figure 4.5). Avoiding to manually ask the user for a message type makes the

configuration of widgets easier and faster for the user, it also keeps the implementation significantly simpler, because no dynamic binding of message type classes during runtime is needed.

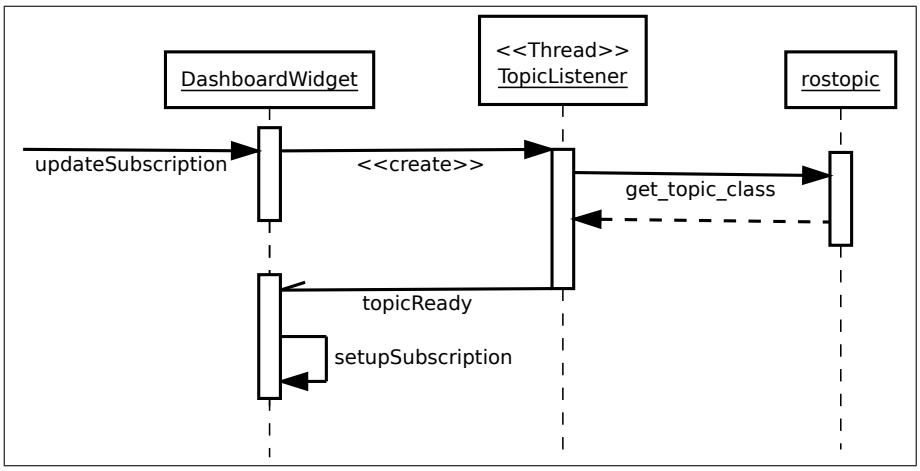


Figure 4.5: Exemplary flow of events for asynchronous topic subscription setup.

4.2.2 Transparent Data Collection with Regular Expressions

Figure 4.4 shows a third field in the user interface to set up a topic subscription. The "Regex" field can be used to apply a regular expression on a message to filter the message or extract numerical data from a String message. As mentioned in Section 2.2.1, the logging messages in ROS are String messages. Those messages often contain numerical data which can currently not be used for visualization because of the lack of type information. Since many existing ROS modules make extensive use of the logging framework it is important to provide access to this information. The regular expression gives developers transparent access to the data embedded in logging messages, without the need to have access to and modify the source code of a module they want to debug.

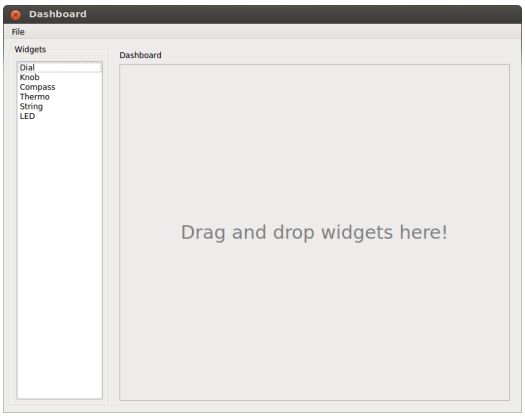
The code to check the regular expression is contained in the **Adapter** class, which parses the value it receives in the `callback()` method from the communication middleware (see Figure 3.6). This hides the regular expression handling from the **DashboardWidget** and **ConcreteVisualizationWidget** (see Figure 3.5). The widget can thus focus on the graphical part and expect polished data which can be visualized straight away, without further pre-processing. Encapsulating the regular expression

matching in the **Adapter** means future plugin developers do not need to care about the regular expression handling unless they want to, in which case they can overwrite the subscription setup methods in their visualization widget implementation to circumvent the integrated regular expression evaluation.

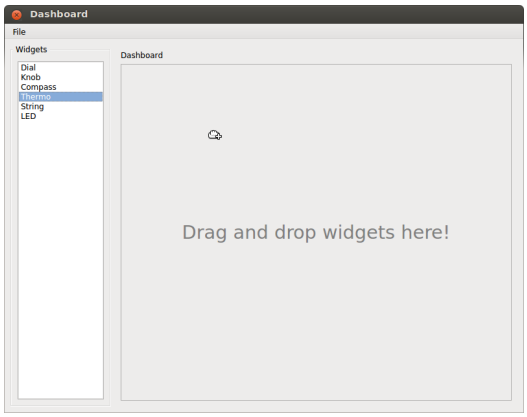
4.2.3 User Interface

The user interface was implemented based on the early interface designs from Section 3.3.1. The main window consists of the dashboard canvas where visualization widgets can be positioned freely and a toolbox where all the available visualization widgets can be selected. The visualization widgets can be positioned on the dashboard by "Drag&Drop" from the toolbox. Figure 4.6 shows a series of screenshots, depicting every step necessary to put a visualization widget on the dashboard: First the desired widget needs to be selected in the toolbox and dragged to the dashboard. When the widget is dropped on the dashboard, the configuration dialog opens automatically where the developer can select to which topic and datafield the visualization widget should connect to. This dialog can be re-opened later from the context menu of the widget, as well as the rename dialog and the properties dialog. It was deliberately chosen to separate the dialog for setting up a topic connection and the dialog for other settings of the widget. This separation of concerns keeps the data provider setup encapsulated in its own dialog, which makes it easier to exchange the data provider if necessary. To remove a widget from the dashboard, simply drag it to the bottom of the dashboard where a special drop zone deletes the widget from the dashboard.

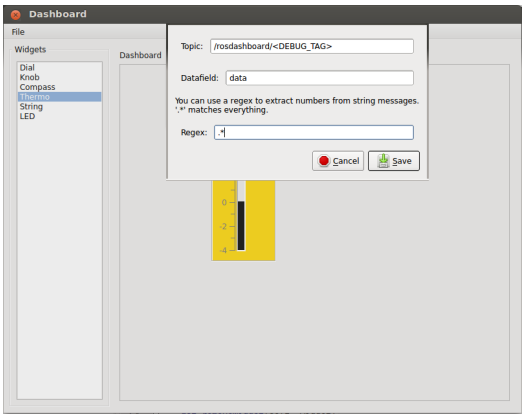
The widget's background has two different colours to provide additional feedback to the developer whether the visualization is connected to a topic or not. When the configured topic for a visualization widget is not available yet, the background of the widget is coloured in orange. Once the topic becomes available the widget changes its background colour to green. This mechanism only checks the availability of a topic on the communication middleware, it does not time out if a topic has been without any messages for a while. Figure 4.7 shows the two different colour states of a widget.



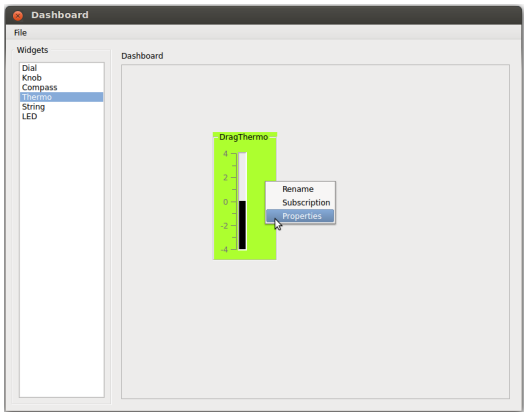
(a) Empty dashboard



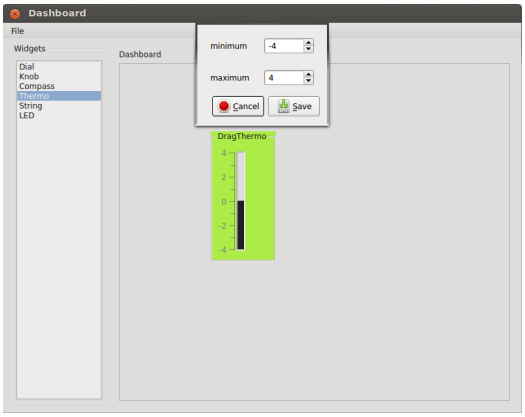
(b) Dragging widget to dashboard



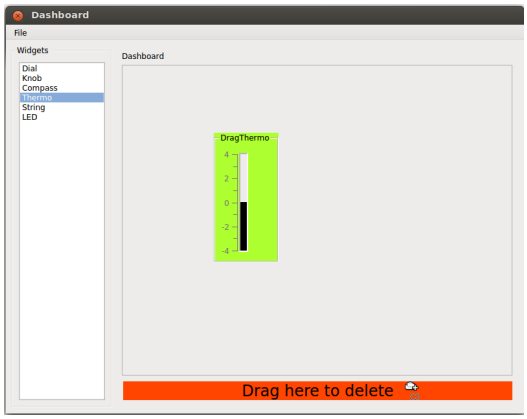
(c) Configure subscription on drop



(d) Context menu



(e) Properties dialog



(f) Drag to bottom to remove

Figure 4.6: Screenflow when adding and removing widgets from the Dashboard.

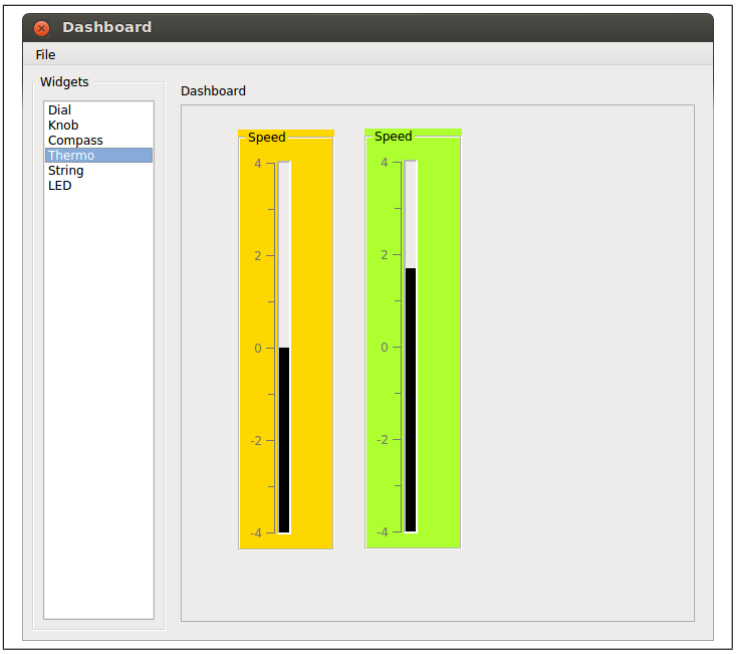


Figure 4.7: Different colour cues provide additional information.

Figure 4.8 shows a dashboard with every currently available widget on it. The currently implemented widgets are:

Dial A dial widget similar to the speedometer on the dashboard of a car or motorbike.

Knob The knob is similar to the dial but is usually smaller. While it is originally often used as input widget, we use it only to visualize numeric data.

Compass The compass is a 360 degree display which can be used e.g. to visualize orientation.

Thermo The thermo is a horizontal progress bar with a scale which is similar to a thermometer.

String The string widget is a text field to show logging messages or other textual information.

LED The LED widget can visualize boolean data, the LED is either green or red.

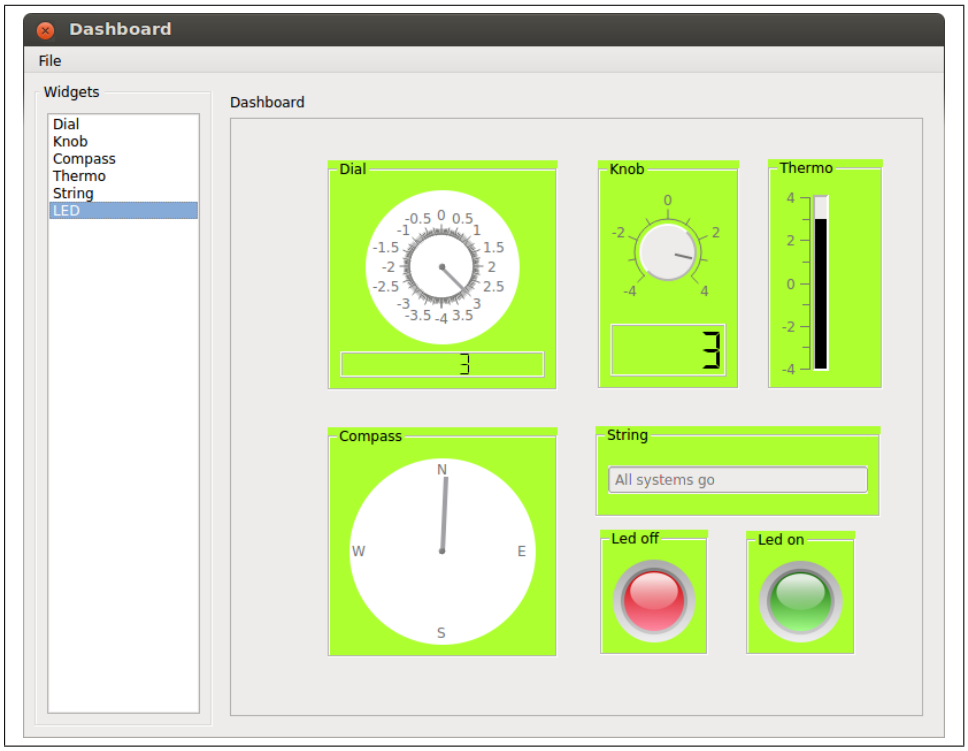


Figure 4.8: Dashboard with all currently available widgets.

4.2.4 Data Format

To share the configuration of a dashboard a save and restore mechanism was integrated. ROSDashboard can save a current configuration of the dashboard to a file which can be used to save the current work and to provide other developers with the same interface. For the prototype all data is stored in a text-based file in JSON (JavaScript Object Notation) [21] format. Listing 4.1 shows an example dashboard configuration represented as JSON, which can be stored in a file. The example contains one visualization widget which is a thermometer widget and is set up to visualize data from the linear field of the */turtle1/command_velocity* topic.

This feature could also be used by robot software and hardware manufacturers to provide a dashboard for a specific module or piece of robot hardware. The dashboard configuration can be made available as part of the module's documentation and represents a easy way to explore the possibilities of a new hardware or software module.

```
{
  "widgets":[
    {
      "width":105,
      "height":197,
      "name":"Linear Velocity",
      "posX":50,
      "posY":25,
      "subscription":{
        "topic":"/turtle1/command_velocity",
        "datafield":"linear",
        "regex":".*"
      },
      "type":"DragThermo",
      "properties":[
        {
          "type":"numeric",
          "name":"minimum",
          "value":-4
        },
        {
          "type":"numeric",
          "name":"maximum",
          "value":4
        }
      ]
    }
  ]
}
```

Listing 4.1: Example dashboard configuration in JSON.

```
1  # log arbitrary data, try to find the data type with
       introspection
2  api.log(tag, data)
3
4  # log string message
5  api.logstring(tag, msg)
6
7  # log integer value
8  api.logint(tag, value, msg_type=Int32)
9
10 # log float value
11 api.logfloat(tag, value, msg_type=Float32)
12
13 # log bool value
14 api.logbool(tag, value)
15
16 # log complex data with datatype
17 api.logdata(tag, data, msg_type)
```

Listing 4.2: ROSDashboard API methods.

4.2.5 API

The current ROS logging framework publishes the log messages on the special purpose topic */rosout*, but converts everything to a String before the transmission. Since those logging messages are text based, valuable information about the type of the data is lost. To use the data for visualization, the type information must be kept. Thus a set of convenience methods were exposed in the ROSDashboard API (Listing 4.2) to provide a simple interface to publish data for the visualization. The interface is similar to the logging API exposed in the ROS client libraries and basically wraps the ROS specific methods to publish data on a topic. Listing 4.3 shows the simple implementation of the `logint` API method.

The API methods shown in Listing 4.2 can be used like logging statements to emit data for visualization. The `TAG` parameter is used to identify the data when the visualization widgets are set up. Internally the data is published as a message on a ROS topic. The topic name consist of the ROSDashboard specific prefix */rosdashboard/* and the tag given as parameter. Listing 4.4 shows an example use of the API to

```
1  def logint(tag, value, msg_type=std_msgs.msg.Int32):
2      """
3      publishes a integer value to the /rosdashboard/<tag>
           topic
4      Be careful to use only valid tags, ROS does not allow
5      dashes and dots in the topic name.
6
7      The default integer message type is std_msg.msg.Int32
8      """
9      pub = rospy.Publisher('/rosdashboard/' + tag, msg_type)
10     pub.publish(msg_type(value))
```

Listing 4.3: Implemented logint API method.

publish a integer value with the tag "proximity". The value "17" will be published on the ROS topic */rosdashboard/proximity* and can be visualized with a widget in ROS-Dashboard that points to the respective topic name and accesses the "data" datafield of the message.

```
1  import rosdashboard
2
3  rosdashboard.logint('proximity', 17)
```

Listing 4.4: Example API usage.

Although this solution proved to be a useful way to expose data during debugging, it might not be feasible for a bigger project because it introduces a new dependency to ROSDashboard. Since the API is only a set of convenience methods that publish data on ROS topics, data can also be published using the same ROS methods the API uses directly in the target application. Other possible solutions will be discussed as future work in Chaper 6.

5 Case Study

Although a thorough evaluation of the developed visual debugging system in general and ROSDashboard in particular was not possible during the course of this work, a case study was conducted to demonstrate the possibilities of such a system. The three cases in this case study show how ROSDashboard can be used with a simple example from the ROS tutorials, how it can be extended with further visualization widgets and how it was used in a real life project to investigate a problem during development.

The first case shows how to connect ROSDashboard's visualization widgets to existing topics in a ROS project. The ROS turtlesim tutorial was chosen as a simple scenario which can be easily reproduced. The example requires no in-depth knowledge of robot programming with ROS since it is created on top of one of the initial tutorials that explain the ROS concepts. The second case in this chapter introduces a new visualization widget for ROSDashboard: The plot widget. This widget can be used to plot data on a Cartesian coordinate system, similar to the dedicated ROS tool rxplot. This case shows how easy it is to extend the current dashboard implementation with further visualization widgets, which was one of the initial requirements for the visual debugging system designed in Chapter 3. The last case examines the use of ROSDashboard during the development of a real life robotics project. An existing problem that appeared during the development of a robotic system with the NAO humanoid robot was investigated using ROSDashboard.

5.1 Simple ROSDashboard Example

Figure 5.1 shows a simple example how ROSDashboard can be used during development. ROSDashboard is running alongside the *turtlesim_ node* node which is used in

many examples in the ROS tutorials[1]. It monitors the values for linear and angular speed which are published by the *turtle_teleop_key* node to control the turtle simulation. The String widget is configured to display logging messages published on the */rosout* topic, which in this example shows a warning when the turtle hits a wall. For the purpose of this simple example, there was no need to modify the turtlesim source code. The only topics used by this scenario are topics that are already used to control the turtle in the simulation and to display warnings.

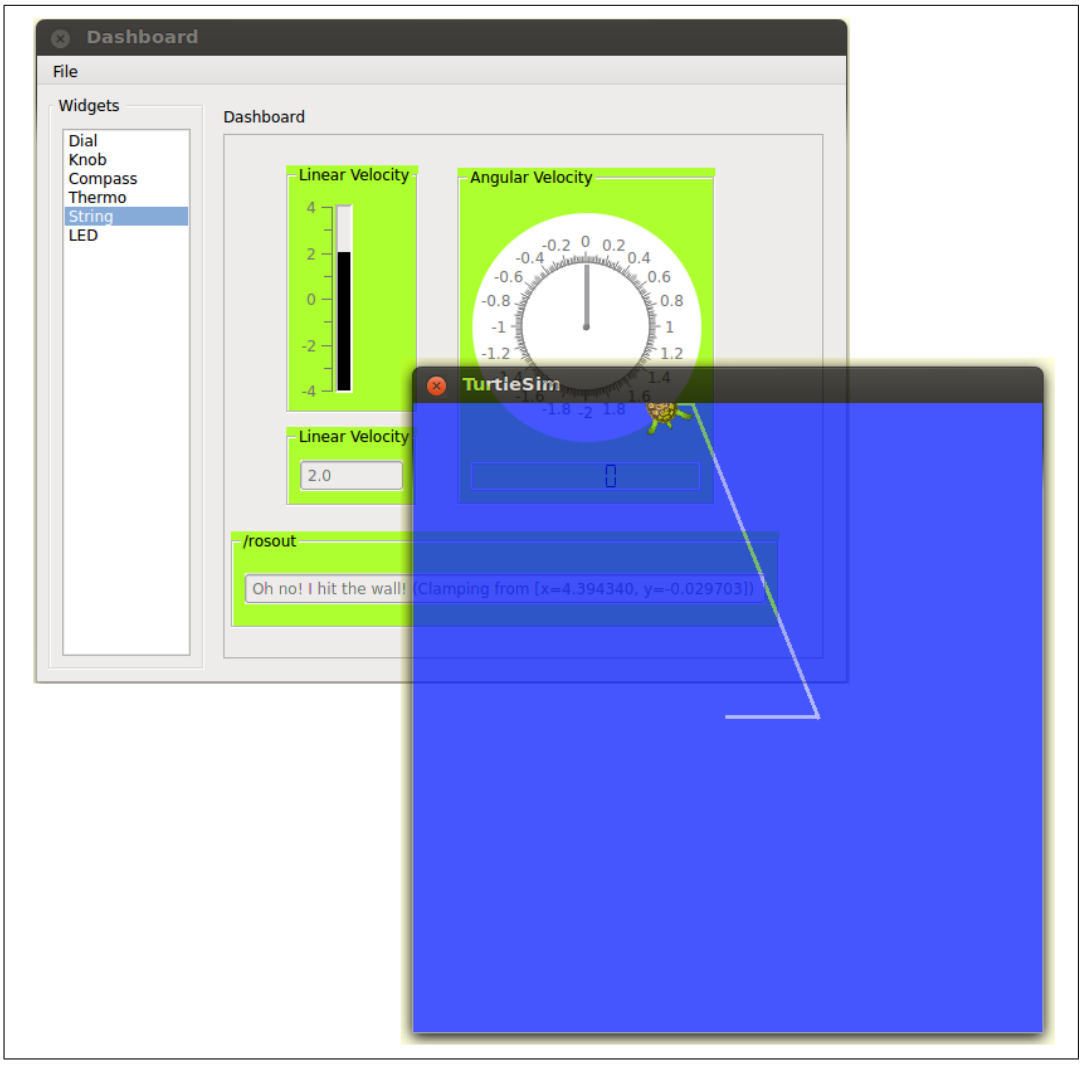


Figure 5.1: ROSDashboard running alongside turtlesim_node.

[1] http://www.ros.org/wiki/ROS/Tutorials

The ROS computation graph during the execution of this scenario is shown in Figure 5.2. It shows how ROSDashboard is connected to the nodes which are debugged. The topics needed for the example are */turtle1/command_ velocity* for the linear and angular velocity and */rosout* for warnings when the turtle hit the wall.

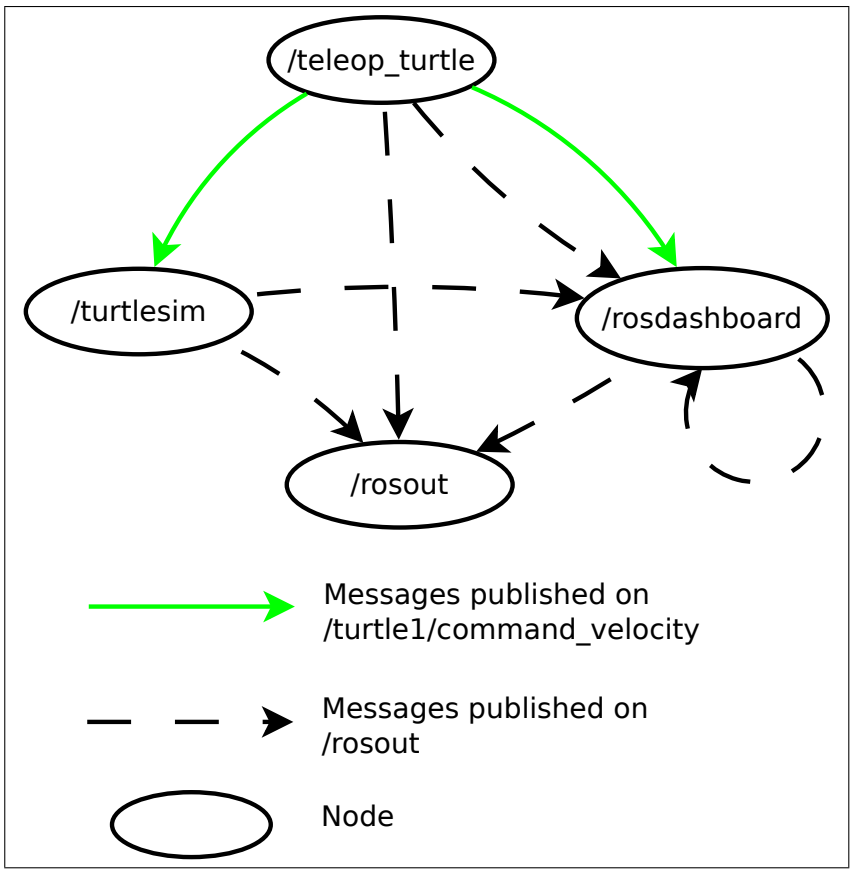


Figure 5.2: Simplified ROS computation graph with ROSDashboard.

5.2 Adding a New Widget

Adding new visualization widgets to ROSDashboard is easy. The abstract **DashboardWidget** class (see Figure 3.5) handles most of the generic things a widget should implement. As part of a first evaluation step a new visualization widget was

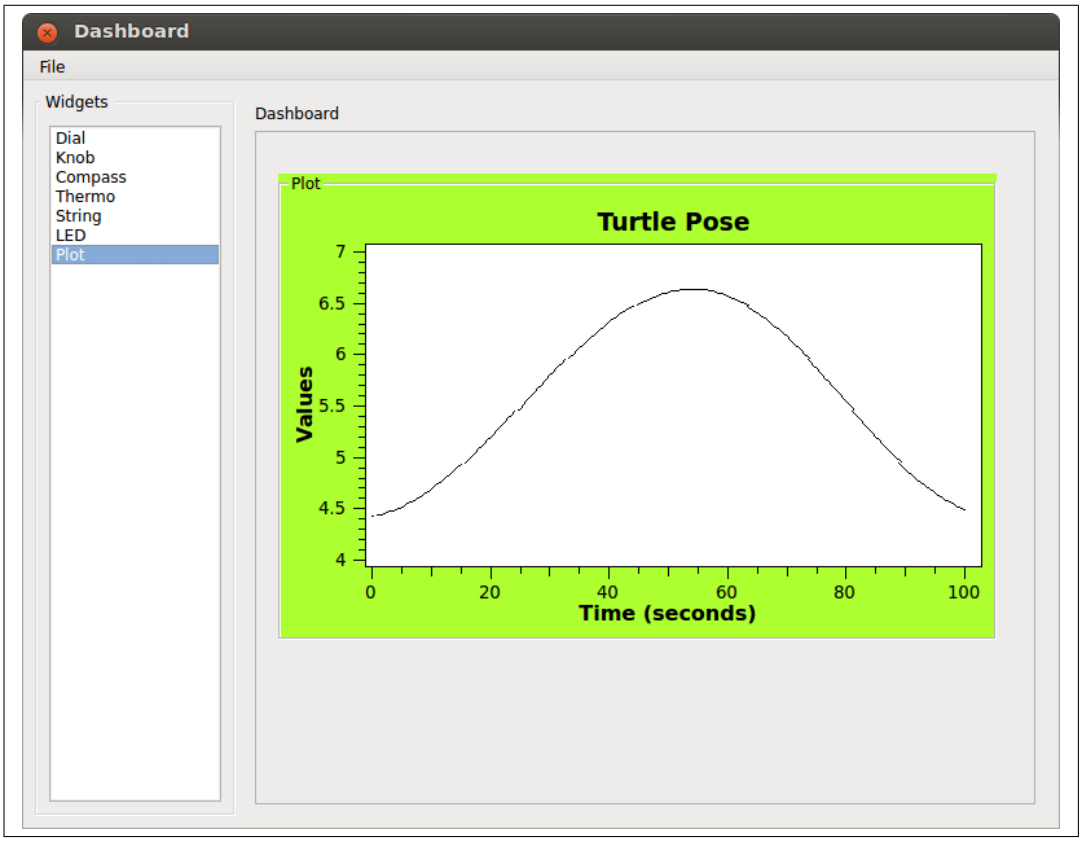


Figure 5.3: The newly created plot widget in action.

added to the dashboard: A plot widget that plots values on a Cartesian coordinate system. Figure 5.3 shows a screenshot of the new visualization widget in action and the full implementation is appended in Listing B.1.

To implement the widget, a new class called **DragPlot** was created. The class inherits the basic implementation from **DashboardWidget** and only needs to implement the plot specific methods. Upon initialization of **DragPlot** the user interface part of the widget is created. The class implements the two abstract methods `updateValue` and `initProperties` from **DashboardWidget**. `updateValue` updates the graphical part of the widget and `initProperties` adds two properties to the widget: plot name and update rate in milliseconds. This requires only two lines of code, see Listing 5.1.

The default implementation in **DashboardWidget** is capable of automatically generating a user interface for those simple properties. When the properties are updated in the properties dialog, a callback is used to notify the **DragPlot** class.

```
1 # property keys
2 TITLE = 'Plot title'
3 RATE = 'Refresh rate (ms)'
4
5 def initProperties(self):
6     self.props[self.TITLE] = WidgetProperty('text', 'Plot title
      ')
7     self.props[self.RATE] = WidgetProperty('numeric', 500)
```

Listing 5.1: Implementation of initProperties in DragPlot.

```
1 def updateWidget(self):
2     #update the widget properties
3     self.qwtPlot.setTitle(self.props[self.TITLE].value)
4
5     if (self.currentTimerId != 0):
6         self.qwtPlot.killTimer(self.currentTimerId)
7         self.currentTimerId = 0
8
9     self.currentTimerId = self.qwtPlot.startTimer(self.props[
        self.RATE].value)
```

Listing 5.2: Implemented updateWidget callback in DragPlot.

The callback implementation to update the widget is shown in Listing 5.2. It updates the title of the plot and starts a new timer which updates the plot in a given interval.

The plot itself is a **QwtPlot** widget from the popular QWT widget library[2]. To customize the plot widget, **QwtPlot** was subclassed in **DataPlot**. The **DataPlot** class exposes two methods: `addValue(value)` and `timerEvent()`. The `addValue(value)` method is used in the implemented abstract method `updateValue(value)` from **DashboardWidget** and adds a new value to the plot. `timerEvent()` is a callback that gets triggered regularly, how often depends on the update rate which can be set in the properties of the widget.

[2] http://qwt.sourceforge.net/

The last step to add the new visualization widget to ROSDashboard is to add it manually to the list of available visualization widgets which appears in the toolbox left of the dashboard canvas. This example shows how easy it is to add a new widget to ROSDashboard. A plugin architecture would make it even easier, since the last manual step is not required and widgets could be dynamically loaded as plugins.

5.3 Real Life Project

A recent problem during the development of an existing project was chosen to show how ROSDashboard can be used for debugging in a real life project. The project's goal is to use the NAO humanoid robot together with ROS to follow people and engage in conversations with them.

The NAO is a humanoid robot developed and sold by Aldebaran-Robotics[3] [22]. Figure 5.4 shows an image of the NAO robot used during this experiment. The robot comes with a comprehensive list of high level software modules that can be used: a text-to-speech module, a voice recognition module, a walking engine, sound localization, etc. Developers also have access to low level sensor information and can control the actuators directly. The robot is shipped with Choreographe, a graphical programming interface to combine the high level modules to behaviours and create new behaviours from scratch.

In the project targeted for this experiment the sound localization module was used to move the head towards the person talking to the NAO robot. The sound localization algorithm is part of the SDK and can be accessed through the robot drivers in ROS, which publish the estimated position of a sound source to a topic. The algorithm publishes the location where the sound comes from as a triplet of the horizontal location (azimuth), vertical location (elevation) and confidence. During the initial experiments with the "Sound Tracker" module in Choreographe it seemed like the NAO robot would sometimes point the head to a random direction unrelated to the real direction of the sound source.

The developer of this system tried to find out whether the problem lies with the accuracy of the sensor, the implementation of the sound localization algorithm or the "Sound Tracker" module moving the head. Since the data from the sound source is published as a triplet of numbers, debugging the issue by looking at numbers in a console was challenging. ROSDashboard was used to visualize the values from the

[3] http://www.aldebaran-robotics.com/

Figure 5.4: NAO robot during the experiment.

sound localization algorithm to determine where the problem of wrong head positions originated.

This section first presents the experimental setup for the tests with the NAO robot. The second part explains how the experiment was conducted with the NAO robot and ROSDashboard, which leads to the results in the third part of this section.

5.3.1 Experiment Configuration

The setup during this experiment was distributed on three different machines. The NAO robot runs on a built in Linux which communicated with the master node where the ROS core, Choreographe and the NAO drivers were running. The third machine was a laptop running ROSDashboard and connecting to the respective ROS topics over the network. All three machines were connected to a router by cable. The master node and the laptop with ROSDashboard were both running with Ubuntu 12.04 (64bit) and had ROS Fuerte, the latest ROS release installed.

While the robot was operated directly by Choreographe, a ROS node was executed as part of the NAO drivers for ROS which published the three values from the sound localization algorithm on the */sound_source* topic. The data from this topic was

visualized in ROSDashboard. A compass widget was used to visualize the azimuth value from the localization algorithm which represents the horizontal location of the sound source. Since the data published by the localization algorithm is in radiants, it first had to be transformed into degrees for better visualization. Apart from the compass widget a dial was used to visualize the raw radiant value from the algorithm and a thermo widget was used to visualize the confidence as a vertical bar ranging from zero to one. Figure 5.5 shows the configuration of widgets during the experiment, which is also available in its JSON representation in Listing B.2 in the appendix.

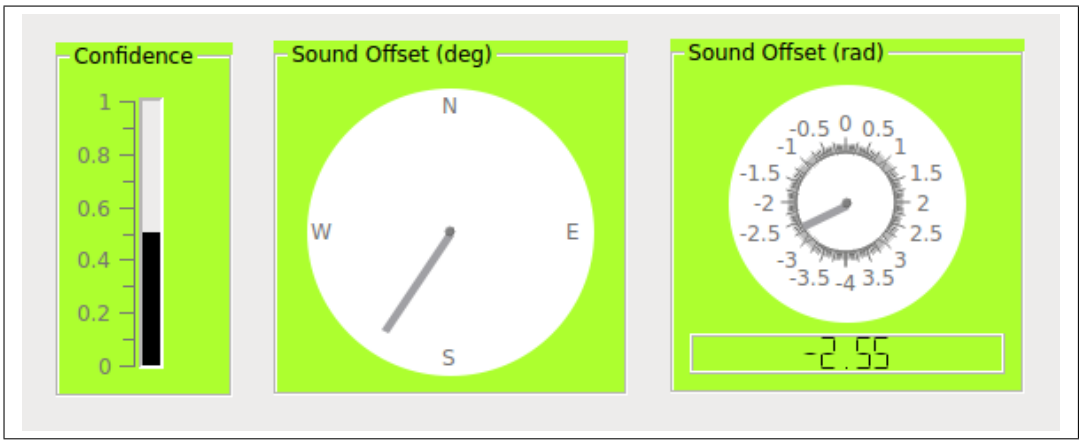


Figure 5.5: Widget configuration during the NAO experiment.

5.3.2 Experiment

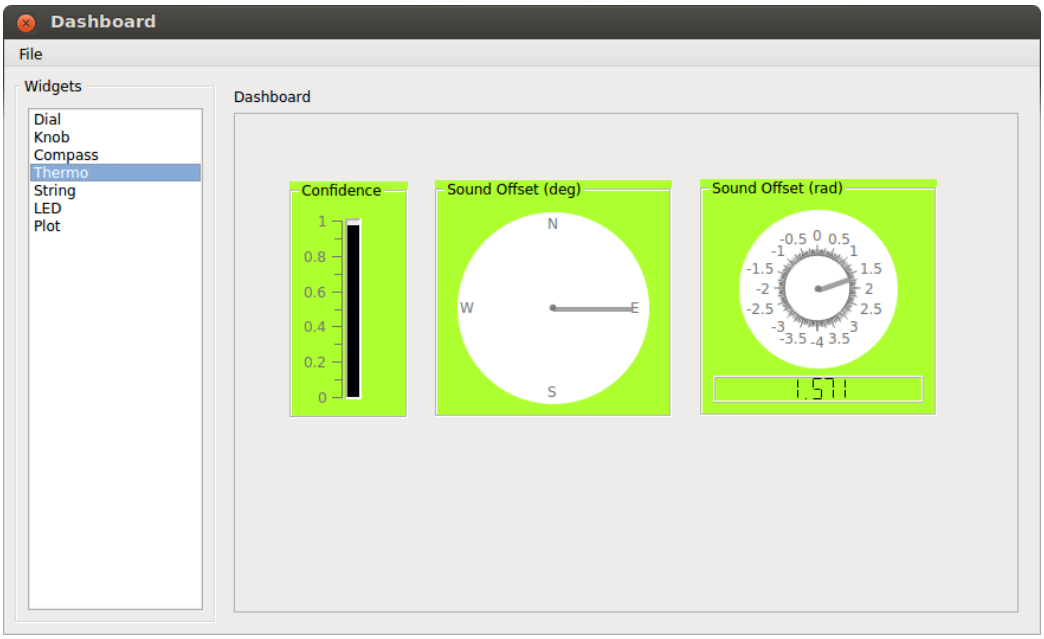
A series of tests were conducted to understand what the values from the sound localization algorithm meant. Several sound sources were created by clapping in close proximity to the NAO robot or simply by talking towards the robot. The first set of test were done with an activated "Sound Tracker" module in Choreographe, which in result moved the head immediately once the NAO robot detected a sound source with a confidence level above a certain threshold. The second set of test was conducted without the "Sound Tracker" module and thus with a static head fixed to its origin. First visualizations of the azimuth value were misleading, but during the experiment it became obvious that the position of the sound source is calculated relative to the position of the head and not relative to the position of the body of the robot which was not moving while the tests were performed. Further research explained this ob-

servation, since the microphones used to determine the location of the sound source are all positioned on the head of the NAO robot.

The tests performed during the experiment also showed that the sound localization algorithm usually detects the location of the sound source correctly, but while the head is moving to the desired position more readings come in from the algorithm. These estimated positions were most of the time less confident than the initial sound localization. If the threshold to move the head was set too low, the robot was interrupted during the movement of the head to the position estimated at first and moved the head to a different position. This made the head movement seem random and not related to the actual source of the sound, which was the original problem examined in this experiment.

5.3.3 Results

This experiment described in this section is based on an existing problem that happened during the development of a small but real life robotics project. ROSDashboard was successfully used to visualize data and helped to understand the data published by the sound localization algorithm. In general it also contributed to a better understanding of the robot under development. The data published by the localization algorithm is much easier to interpret and understand using ROSDashoard than looking at the numbers scrolling down in a console. Figure 5.6 shows the dashboard interface visualizing the data and the raw output printed in a console with the command line tool `'rostopic echo /sound_source'`.



(a) ROSDashboard

(b) Console

Figure 5.6: Side by side comparison of ROSDashboard and console output during the experiment.

6 Future Work and Conclusion

The visual debugging tool presented in this work and more specifically the ROSDashboard implementation of such a system is capable of visualizing simple abstract data that is being published on a topic in the ROS communication middleware. Although the tool presented in this work has not been fully evaluated yet, it shows the possibilities and potential of simple visualizations during debugging of robotic applications. Further evaluations need to be conducted in order to quantify the improvements the tool has on a reduced cognitive effort and thus debugging performance.

This chapter summarizes the future work that has been identified during this work and some final words conclude the thesis.

6.1 Future work

The implemented tool is a first prototype of a simple visualization tool which can be used during debugging. The existing visualization widgets are relatively basic and further work can be done to create more visualization widgets and explore more complex types of visualization. Ultimately there should be a plugin engine where third party developers can easily add widgets for new and more complex kinds of data. The extensibility of the tool was one of the initial requirements and the object structure presented in Section 3.3.3 allows easy integration of such a plugin engine.

This section gives an overview of the possible improvements for ROSDashboard and the visual debugging system in general which have been identified.

6.1.1 User Interface Improvements

Apart from more visualizations, the graphical interface can be improved in different ways. The "Drag&Drop" mechanism can be improved to provide more feedback to the user while dragging. For example an outline could indicate where the widget would be dropped on the dashboard. Currently the widgets can be freely positioned on the dashboard canvas, a snap-to-grid function could help to structure the widgets on the dashboard.

While the initial prototype's implementation of the topic setup dialog (Figure 4.4) features simple text fields where the developer can enter arbitrary Strings, a more sophisticated solution can be implemented to improve the user interface. Using the existing ROS tools a list of available topics can be accessed, which makes it possible to create smarter interfaces. For example type-ahead completion for the topic name and a drop down list to choose the datafield parameter of a message are possible options.

Another possibility to improve the current user interface is to change the widgets to be more general and thus make it possible to have both visualization widgets and control widgets. This would give developers the ability not only to monitor values during execution but also manipulate configuration values and give commands to the robot during a debugging session.

6.1.2 Plugin Framework

Although a plugin system has been designed in Section 3.3.4, it has not been implemented in ROSDashboard yet due to the time constraints of this work. Section 5.2 shows how easy it is to add the necessary code for a new visualization widget but the widget had to be added manually to the list of available widgets in the toolbox.

The example shows that the object model is flexible enough to allow the introduction of new widgets without much overhead. The newly added plot widget was able to re-use the existing skeleton from the abstract **DashboardWidget** class and only had to implement the abstract methods and related callbacks. With the proposed plugin framework it would be easy for third party developers to add their own widgets without modifying the ROSDashboard source code directly.

6.1.3 Exchangeable Data Providers

Currently the data for the visualizations is collected by either re-using existing communication between ROS nodes or publishing dedicated data for the visualization. Publishing dedicated visualization data can be done through the ROSDashboard API or directly with the ROS communication API. If the ROSDashboard API is used, a new dependency must be added to the node that is debugged.

While this approach is good enough for a prototype, other ways of collection data should be considered in the future. The realtime debugging approach presented in Section 2.1.2 could be a possible source for visualization data. It can be used to collect data in realtime and publish the data on the communication middleware of the target robotic framework. The system design presented in Section 3.3 makes it relatively easy to exchange the data provider since the communication with the middleware is encapsulated in the **Adapter** class.

6.1.4 Extend the ROS Logging Framework

Since the current logging mechanism in ROS transmits data as text, existing logging statements have to be parsed with a regular expression to be used as data source for the visualization (see Section 4.2.2). The other way to connect data in ROSDashboard is to expose a set of API methods (see Section 4.2.5) which allow easy publication of data. This solution makes it necessary to add ROSDashboard as a dependency to the node that is debugged. A more transparent approach to collect data for debugging could be to extend the current logging framework in ROS to provide methods that allow logging of typed data. The API in ROSDashboard is an example for such methods, but is pretty basic. A more sophisticated solution would allow to exclude the logging statements if the node is not in debugging mode.

Of course it is also possible to publish data manually. The ROSDashboard API only provides a convenient set of wrapper methods which take care of the data publishing for the developer. Having specialized API methods to publish data for debugging purposes makes it easier to emit debugging data and helps to distinguish which data is used for debugging only and which data is used beyond the debugging scope.

6.1.5 Automatic Dashboards

The current dashboard interface offers a flexible canvas where developers can drop the visualizations they need to investigate a problem during debugging. A possible extension of the current system could be to automatically generate debugging dashboards using the data that is currently available on the communication middleware. The dashboard could detect which topics are currently active and contain visualizable data. Based on that information it could recommend visualization widgets to the developer and thus raise the visibility of existing data and visualization possibilities.

6.1.6 rqt Integration

rqt[1] is a graphical interface that groups together many different graphical tools for ROS. To integrate a graphical tool in rqt it must be wrapped as a plugin and implement the rqt plugin API. Since RQT was under active development during the time of this project, it was chosen not to integrate ROSDashboard into rqt yet, but to keep it in mind for future work. The plugin interfaces to integrate a tool into rqt have been finalized recently and ROSDashboard can easily be integrated.

rqt not only gives you a way to create your own user interface by combining existing tools, it also raises the visibility of the various tools that can be used to develop robots with ROS. Thus it would be good to integrate ROSDashboard in rqt to raise the visibility of the tool and promote its usage.

6.2 Conclusion

As result of this work, a flexible visual debugging system has been developed and documented. The system itself is not bound to a specific robotic framework but has taken some inspiration from the ROS architecture. With ROSDashboard a first implementation of such a system was presented. The developed prototype has been announced to the ROS community and the source code is available on Github[2]. After the prototype has been announced to the ROS community, valuable feedback has

[1] http://www.ros.org/wiki/rqt
[2] http://github.com/kaserf/rosdashboard

been collected. This resulted in some smaller changes and improvements, which have been applied to ROSDashboard already. Other suggestions have been documented as future work, since they were out of scope for this work.

ROSDashboard has been successfully used to debug a real world problem during a project (see Section 5.3). The modular design and Open Source approach followed during the development of the tool makes it easy to extend and adapt the tool in the future. The visual debugging system also allows developers to choose a representation of data they find most helpful, with the only restriction being the number of available widgets and the data type compatibility. How important an individual choice of visualization is, needs to be evaluated in an extensive user study. The simple "Drag&Drop" principle to add and remove widgets on the dashboard makes the tool flexible towards changes during the debugging process and provides a universal canvas to create debugging dashboards for any kind of project and robot hardware.

The results of this work have also been summarized in a conference paper and submitted to the 2013 IEEE International Conference on Robotics and Automation (ICRA) where it is currently being reviewed.

Bibliography

[1] Luke Gumbley. *Real-World Robotic Debugging*. Master thesis, The University of Auckland, 2010.

[2] Luke Gumbley and Bruce MacDonald. Realtime Debugging for Robotics Software. In *Australasian Conference on Robotics and Automation (ACRA)*, Sydney, Australia, 2009.

[3] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*, 40(1):163, February 2008.

[4] Leo Gugerty and Gary Olson. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '86*, number April, pages 171–174, New York, New York, USA, 1986. ACM Press.

[5] Michael Grottke and KS Trivedi. A classification of software faults. In *Supplemental Proc. Sixteenth International Symposium on Software Reliability Engineering*, pages 4.19–4.20, 2005.

[6] Timothy Jacobs and Benjamin Musial. Interactive visual debugging with UML. In *Proceedings of the 2003 ACM symposium on Software visualization - SoftVis '03*, page 115, New York, New York, USA, 2003. ACM Press.

[7] THJ Collett and BA MacDonald. An Augmented Reality Debugging System for Mobile Robot Software Engineers. *Journal of Software Engineering for Robotics*, 1(1):18–32, 2010.

[8] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB: The GNU Source-Level Debugger*. Ninth edition, 2002.

[9] Tully Foote. ROS community metrics. http://pr.willowgarage.com/downloads/metrics/metrics-report-2012-07.pdf, July 2012.

[10] Morgan Quigley, Ken Conley, and Brian Gerkey. ROS: an open-source Robot Operating System. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 32, pages 151–170, Kobe, Japan, 2009.

[11] ROS Wiki: rxconsole. http://www.ros.org/wiki/rxconsole. Accessed: 26.10.2012.

[12] K.N. Whitley and Alan F. Blackwell. Visual Programming in the Wild: A Survey of LabVIEW Programmers. *Journal of Visual Languages & Computing*, 12(4):435–472, August 2001.

[13] B Gerkey, RT Vaughan, and A Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, Coimbra, Portugal, 2003.

[14] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. Orca: Components for robotics. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Beijing, China, 2006.

[15] Michael Montemerlo, Nicholas Roy, and S Thrun. Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 2436–2441, 2003.

[16] ROS Wiki: RViz user manual, plugins section. http://www.ros.org/wiki/rviz/UserGuide#Plugins. Accessed: 27.09.2012.

[17] ROS Wiki: rxbag_plugins roadmap. http://www.ros.org/wiki/rxbag_plugins. Accessed: 24.09.2012.

[18] Patrick Th. Eugster, Pascal a. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[19] Morgan Quigley, Eric Berger, and AY Ng. Stair: Hardware and software architecture. In *AAAI 2007 Robotics Workshop*, Vancouver, Canada, 2007.

[20] Steve Cousins, Brian Gerkey, Ken Conley, and Willow Garage. Sharing Software with ROS [ROS Topics]. *IEEE Robotics & Automation Magazine*, 17(2):12–14, June 2010.

[21] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, RFC Editor, July 2006.

[22] David Gouaillier, Vincent Hugel, Pierre Blazevic, Chris Kilner, Jerome Monceaux, Pascal Lafourcade, Brice Marnier, Julien Serre, and Bruno Maisonnier. The NAO humanoid: a combination of performance and affordability. *CoRR*, abs/0807.3, July 2008.

Listings

```
1  from python_qt_binding.QtBindingHelper import QT_BINDING,
       QT_BINDING_VERSION #@UnresolvedImport @UnusedImport
2  import QtGui #@UnresolvedImport
3  import QtCore #@UnresolvedImport
4
5  from rosdashboard.modules.props import WidgetProperty
6  from rosdashboard.modules.dashboardWidgets import
       DashboardWidget
7  from PyQt4.Qwt5 import Qwt
8  from PyQt4.Qwt5.anynumpy import *
9
10 # data plot class from http://pyqwt.sourceforge.net/examples/
       DataDemo.py.html
11 class DataPlot(Qwt.QwtPlot):
12
13   def __init__(self, *args):
14     Qwt.QwtPlot.__init__(self, *args)
15
16     self.setCanvasBackground(QtCore.Qt.white)
17     self.alignScales()
18
19     # Initialize data
20     self.x = arange(0.0, 100.1, 0.5)
21     self.y = zeros(len(self.x), Float)
22
23     self.curveL = Qwt.QwtPlotCurve("Data Moving Left")
24     self.curveL.attach(self)
25
```

```
26      mY = Qwt.QwtPlotMarker()
27      mY.setLabelAlignment(QtCore.Qt.AlignRight | QtCore.Qt.
            AlignTop)
28      mY.setLineStyle(Qwt.QwtPlotMarker.HLine)
29      mY.setYValue(0.0)
30      mY.attach(self)
31
32      self.setAxisTitle(Qwt.QwtPlot.xBottom, "Time (seconds)")
33      self.setAxisTitle(Qwt.QwtPlot.yLeft, "Values")
34
35    # __init__()
36
37    def alignScales(self):
38      self.canvas().setFrameStyle(QtGui.QFrame.Box | QtGui.
            QFrame.Plain)
39      self.canvas().setLineWidth(1)
40      for i in range(Qwt.QwtPlot.axisCnt):
41        scaleWidget = self.axisWidget(i)
42        if scaleWidget:
43          scaleWidget.setMargin(0)
44        scaleDraw = self.axisScaleDraw(i)
45        if scaleDraw:
46          scaleDraw.enableComponent(Qwt.QwtAbstractScaleDraw.
                Backbone, False)
47
48    # alignScales()
49
50    def timerEvent(self, e):
51      #repeat the current value
52      self.addValue(self.y[-1])
53
54    # timerEvent()
55
56    def addValue(self, value):
57      #shift array left and add new value
58      self.y = concatenate((self.y[1:], self.y[:1]), 1)
59      self.y[-1] = value
60
61      self.curveL.setData(self.x, self.y)
```

```
62
63          self.replot()
64      # addValue()
65
66  # class DataPlot
67
68
69  class DragPlot(DashboardWidget):
70      """ draggable plot """
71      TITLE = 'Plot title'
72      RATE = 'Refresh rate (ms)'
73
74      def __init__(self, parent = None):
75          super(DragPlot, self).__init__(parent)
76          self.setTitle('DragPlot')
77
78          self.currentTimerId = 0
79
80          self.initUI()
81
82      def initUI(self):
83          self.layout = QtGui.QVBoxLayout()
84          self.qwtPlot = DataPlot()
85
86          self.layout.addWidget(self.qwtPlot)
87
88          #initial size
89          self.resize(400,300)
90
91          #update widget according to properties
92          self.updateWidget()
93
94          self.setLayout(self.layout)
95
96      def initProperties(self):
97          self.props[self.TITLE] = WidgetProperty('text', 'Plot
                title')
98          self.props[self.RATE] = WidgetProperty('numeric', 500)
99
```

```
100   def propertiesDialogAccepted(self):
101     self.updateWidget()
102
103   def updateWidget(self):
104     #update the widget properties
105     self.qwtPlot.setTitle(self.props[self.TITLE].value)
106
107     if (self.currentTimerId != 0):
108       self.qwtPlot.killTimer(self.currentTimerId)
109       self.currentTimerId = 0
110
111     self.currentTimerId = self.qwtPlot.startTimer(self.props[
          self.RATE].value)
112
113   def updateValue(self, value):
114     self.qwtPlot.addValue(float(value))
```

Listing B.1: DragPlot implementation.

```
{
  "widgets":[
    {
      "width":200,
      "name":"Sound Offset (rad)",
      "posX":394,
      "posY":56,
      "subscription":{
        "topic":"/sound_source",
        "datafield":"azimuth",
        "regex":".*"
      },
      "type":"DragDial",
      "properties":[
        {
          "type":"numeric",
          "name":"minimum",
          "value":-4
        },
        {
          "type":"numeric",
          "name":"maximum",
          "value":4
        }
      ],
      "height":200
    },
    {
      "width":200,
      "name":"Sound Offset (deg)",
      "posX":170,
      "posY":57,
      "subscription":{
        "topic":"/sound_source",
        "datafield":"azimuth",
        "regex":".*"
      },
      "type":"DragCompass",
```

```
39          "properties":[
40
41          ],
42          "height":200
43       },
44       {
45          "width":100,
46          "name":"Confidence",
47          "posX":47,
48          "posY":58,
49          "subscription":{
50             "topic":"/sound_source",
51             "datafield":"confidence",
52             "regex":".*"
53          },
54          "type":"DragThermo",
55          "properties":[
56             {
57                "type":"numeric",
58                "name":"minimum",
59                "value":0
60             },
61             {
62                "type":"numeric",
63                "name":"maximum",
64                "value":1
65             }
66          ],
67          "height":200
68       }
69    ]
70 }
```

Listing B.2: Saved dashboard from the NAO case study.