INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin

www.informatik2011.de

Robbie: A Message-based Robot Architecture for Autonomous Mobile Systems

Susanne Thierfelder, Viktor Seib, Dagmar Lang,
Marcel Häselich, Johannes Pellenz, Dietrich Paulus
{sisuthie, vseib, dagmarlang, mhaeselich, pellenz, paulus}@uni-koblenz.de
Active Vision Group, AGAS Robotics
Institute for Computational Visualistics
University of Koblenz-Landau
`http://robots.uni-koblenz.de`

Abstract: Designing a generic robot system architecture is a challenging task. Many design goals, such as scalability, applicability to various scenarios, easy integration of soft- and hardware, and reusability of components need to be considered. The code has to be kept easy to read and maintainable by developers and researchers. In this paper we describe the message-based software architecture Robbie that was specifically designed to address these goals. It has been successfully applied to fulfill various and complex tasks for different robots and scenarios in the context of autonomous mobile systems. We also examine how Robbie is related to the widely spread robot operating system ROS.

1 Introduction

Software systems for autonomous mobile applications are increasingly complex as one has to deal with heterogeneous hardware components, which need to be integrated into a working system. The software should be easily adaptable to various robotic platforms and hardware configurations. Different levels of abstractions are required and need to be managed by the software. These requirements lead to a rapidly growing software size and complexity. At the same time the system has to remain maintainable and has to allow easy integration of new components e.g. for research purposes.

A broad variety of robot architectures already exist focusing on different tasks and applications. OpenRDK, a modular framework focusing on rapid development of distributed robotic systems, was presented in [CCIN08]. The main entity of the OpenRDK framework is a process, called agent. It contains several modules, each as a single thread inside the agent. The idea behind the OROCOS project [Bru01] aims at establishing a community to design and develop an open source robotics platform. The OROCOS code is divided into different types of modules to manage the huge complexity of a big software project. Information sharing is done using CORBA. Miro [USEK02] also uses CORBA for information sharing. Miro is a middleware for mobile robot applications and was designed with multi-platform support and interoperability in mind. Orca, on open-source component-based

INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin

www.informatik2011.de

software engineering framework for robotic applications was proposed in [BKM$^+$05]. It focuses on reusability and customization of software parts. Recently ROS [QCG$^+$09], an open-source meta-operating system for robots, was introduced. Like our proposed architecture, ROS aims at simplifying software development for robots by providing a modular, scalable, and easy extendable system. ROS has many similarities to our architecture, but differs in some key aspects. The relation between ROS and Robbie will be discussed in a later chapter.

In this paper we describe the robot system architecture Robbie, which was developed at the University of Koblenz-Landau to address the requirements mentioned above. Robbie is easily adaptable to various hardware and software configurations[1]. In addition to research purposes, focus was put on designing the architecture for teaching. Over the past years the Robbie architecture has proven its advantages in various student projects (for this year's projects please refer to [TGNL$^+$11] and [LHP$^+$11]). It enables the students to familiarize with a complex research subject in a short time and develop own ideas that can be easily tested in practice. We successfully use this software architecture for different types of robots participating at various robotic competitions: The team of Lisa participated successfully as finalists at the RoboCup@Home[2] World Championship in Suzhou, China (2008), Graz, Austria (2009) and in Singapur (2010), where it was honored with the RoboCup@Home Innovation Award. Robbie competes in the RoboCupRescue[3] league since the RoboCup competitions 2007 in Atlanta, USA. Since then our team was honored, among other prices, with the Inter-league Mapping Challenge award (2010) and became German Champion in Rescue Autonomy for the 5th time in 2011. The teams competing at the SICK Robot Day[4] placed 1st and 2nd in 2009 and 2010 in Freiburg, Germany (see Figure 1).

The system's design goals will be presented in Section 2, followed by a detailed description of the components of the system architecture in Section 3. Applications and scenarios will be shown in Section 4. Additionally, Chapters 5 and 6 will focus on the interface to ROS integrated in our architecture and compare both approaches. We close with a conclusion in Section 7.

2   Design Goals

In designing Robbie we focused on the following goals:

Modular structure and maintainability: Robbie consists of various independent modules that are connected to the system. Specific tasks are encapsulated in isolated program blocks and can be maintained or redesigned without the need to adapt large parts of the system.

Extensibility and adaptability: The modular structure also allows for a simple integra-

---

[1]ADAPT is a research focus of the University of Koblenz-Landau `http://adapt.uni-koblenz.de/`
[2]RoboCup@Home league `http://www.robocupathome.org/`
[3]RoboCupRescue league `http://www.robocuprescue.org/`
[4]SICK Robot Day `http://www.sick.com/group/EN/home/pr/events/robot_day/`

Figure 1: Our robots using the Robbie system architecture. From left to right: Rescue robot Robbie (2011), robots for SICK Robot Day (on commercial racing car (2007) and with a Pioneer3AT-platform (2010)), cleaning and disposal robot Waylon (2010) and GiGo (Roomba 531 from iRobot as platform, 2011), household robot Lisa for RoboCup@Home (2011).

tion of new components. For example making Robbie adaptable to new hardware or new fields of functions. Further, this structure allows for fast prototyping and testing of new algorithms which makes it perfectly suitable for research purposes.

Concurrency: Every module connected to Robbie is executed in its own thread. Thereby the architecture is perfectly suited for modern computer hardware that increasingly relies on multiple processors rather than one single CPU.

Dynamical configuration: Robbie is a loosely coupled system that can be configured via a special registry allowing to load different sets of modules. By this mechanism the system is easily adoptable to different scopes of tasks without the need of recompiling the program.

Different abstraction layers: Modules for Robbie can be designed on different abstraction layers. They can either directly communicate with hardware, encapsulate algorithms to perform specific tasks, or control the whole application, thus, realizing higher-lever behavior. Of course, if required, this abstraction layers can be mixed in one module. From the system's point of view all modules have the same interface disregarding their abstraction level.

Message-based: The communication between the modules is strictly message-based. All data that need to be transfered between modules is encapsulated into messages. Messages are passed to the system and forwarded to the right recipient.

Easy to learn: One focus of our research group is teaching robotic technology to students. As student projects usually extend over one term only it is very important that students can rapidly familiarize with the system and start developing own components. The Robbie architecture is perfectly suited for this purpose as it is confirmed by our successful participation at various competitions.
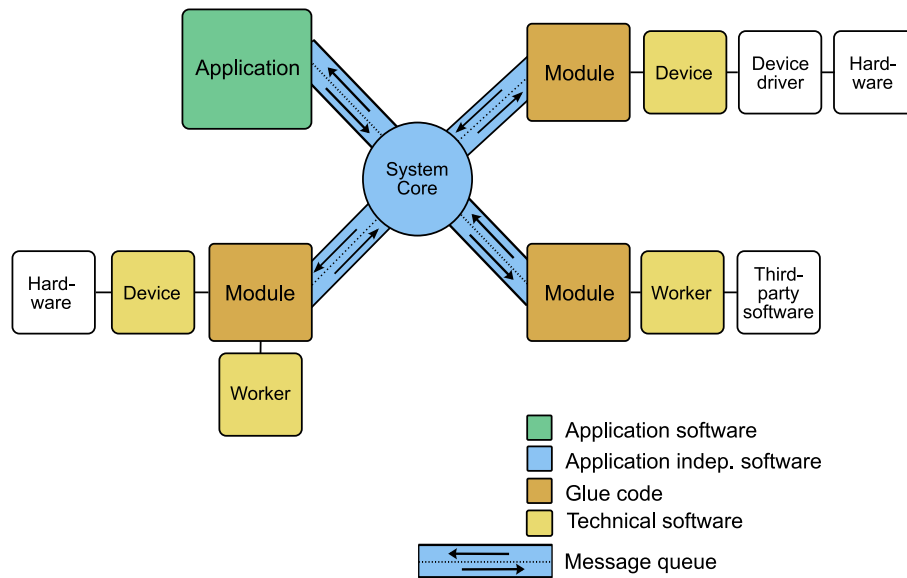
INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin

www.informatik2011.de

Figure 2: The Robbie architecture concept

3 Components and Architecture

The software architecture of Robbie follows a loosely coupled, strictly message-based, and event driven software design: the center of the architecture is a generic, application independent system core that serves as a dispatcher between the connected components. All types of data are transported as messages. So called modules register at the core and subscribe to messages relevant to their particular task. If a module wants to share information with another module it has to send a message to the system core which delegates it to all subscribers. All inter-module communication is realized via the system core. Thus, all modules are kept independent of each other. The only exceptions are tasks concerning the system as a whole such as recording log-files, profiling and logging of system notifications (see Sections 4.6 and 4.7).

Every module is executed in its own thread and is connected by two queues to the core: The *inbound queue*, containing the subscribed messages that are send from the core to the module and the *outbound queue*, containing messages send by the module to the core. The system can be configured dynamically by a central registry that contains various profiles that store the required modules and configuration settings for a certain task.

The concept of the system is presented in Figure 2. The idea of this architecture has its origin in the *Quasar windmill* [Sie02], but extends this concept by introducing a generic core and hiding the application specific details in one of the modules.

INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin

www.informatik2011.de

3.1 System Core

The core's main task is the exchange of messages between modules. The modules are loosely coupled with the system core: rather than defining the necessary modules in the program code, the required modules are listed in a configuration file. These modules are created on start up and automatically registered at the core. When registering, each module subscribes to messages it is interested in. The core manages the subscription information and forwards the messages to all subscribing modules, thus, waking the corresponding modules' threads.

There are two kinds of message subscriptions: either a module wants to receive all messages of the specified type or it is only interested in the newest message of that type. In the first case the messages are simply added to the inbound queue of the module. In the second case the system core checks whether the message type is already contained in the queue. If not, the message is added to the end of the queue. Otherwise, the former message of that type is replaced by the new one.

3.2 Modules

A module is a thin layer that connects workers (algorithms) and devices (hardware) with the system core: It can interpret and create messages for the communication with the core, but it also knows how to talk to a particular device or how to use the functionality of a particular worker. Thus, modules act as glue code between these two worlds. Using this glue code, the devices and workers do not have to deal with the system specific objects (e.g. messages), and are therefore very independent of the target (robotic) application. The use of the modules helps also to isolate the generic system core from the application specific parts of the software.

When creating new modules the user can choose between *passive* and *active* modules. A passive module sleeps until an incoming message from the system core wakes its thread. The module processes the information, generates one or several messages as response and sends them back to the system core. This kind of modules is usually used to process data generated by other modules. Like passive modules, active modules react when new messages arrive. Additionally, they are periodically activated (e.g. every 50 ms). This type of modules is used to poll and transmit data from sensors to the system core. When subscribing to a message there are two ways of dealing with them: `Message-Handling::AlwaysDeliver` is the standard case that lets the module process every new message that arrives. When this option is set to `MessageHandling::Keep-OnlyNewest` only the latest message of that type is stored in the queue of the module. This behavior is necessary if messages with a high frequency are subscribed and the module wouldn't be able to handle them all in time, which means that the module needs to queue them (bottleneck effect).

Apart from connecting workers and devices to the system core, several modules exist that cover specific tasks required by a robotics system. Further, an application control module

INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin

www.informatik2011.de

can be connected to the system core to control the whole system or to encapsulate higher-level behavior. Examples for this specialized modules and their respective role in the Robbie system will be presented in Section 4.

3.3 Messages

A robotic system has to deal with large amounts of data, such as color images or 2D and 3D laser range data. Wrapping the data into messages would be very inefficient, especially if more than one receiver subscribed to the data. Therefore, only the pointers to the messages are sent to the modules and the core takes care of the memory management: After a module is done with a message, it signals the system core that the data is not used anymore. When all modules notified the system core that the message is handled the data associated with the message is released and the message itself is deleted.

All messages have the capability to serialize and to deserialize themselves. This way, the messages can be written to a logfile or sent via a (wireless) network to an operator station that can be used to monitor and control the robot.

3.4 Devices

To perceive its environment and to interact with it, a robot needs certain hardware components. All these components like sensors, manipulators or the robot platform itself need to be connected to the software system. In the Robbie architecture the so called devices communicate with a certain piece of hardware. This communication can be either directly by encapsulating a custom hardware driver or by using external libraries. In order to make devices easily exchangeable, they do not know about modules or messages.

3.5 Workers

Other important components of the system are workers. A worker is a small set of program code which fulfills a certain task for the application by implementing a specific algorithm. Like devices, workers can use external libraries and do not know about modules or messages. Unlike devices workers never communicate with hardware directly. Therefore, they can be used independently of the underlying hardware platform.

4 Applications and Scenarios

In this Chapter we describe common use cases that arise while working with robot platforms. We explain how the Robbie architectures is adopted to handle these scenarios.

Various tools allow easy adaptation, configuration, visualization and debugging of software developed in this framework.

4.1 Dynamic Configuration

The setup of the Robbie architecture can be customized via a configuration file. The file contains a hierarchical XML structure describing profiles for different purposes. To select a profile its name has to be supplied as command line argument when starting the application. Each profile defines which modules are loaded at the system start. Since modules are used as glue code to devices, a profile also defines which hardware is used. Further, configuration values and thresholds for algorithms are defined in the configuration profiles. The appearance of the GUI can also be customized, displaying only the relevant information whereas unnecessary tabs and widgets are hidden. Additionally, objects, persons and speech commands can be loaded with each profile.

The parameters of the XML-file are represented by a `value`-Tag. The name of parameter has a prefix defining the types bool, int, float or string. A global `Config`-Object modeled as a singleton can access the data depending on its type (e.g. `Config::getInt`, `Config::getString`).

Some modules (such as the navigation module) might be needed in almost all profiles. To facilitate the definition of new profiles and avoid repeated inclusion of the same set of fundamental modules, a profile can include parent profiles. When a parent profile is included all of its properties are automatically added to the including profile. Thus, only the configuration values specific for the including profile need to be defined. This configuration structure also allows to test different system setups, hardware configurations and algorithm thresholds without recompiling the whole program code.

4.2 Higher-level behavior

Our robots already participated in different competitions with a broad variety of tasks. In the RoboCupRescue league the robot has to navigate through a simulated disaster environment, create a map of the area and find simulated victims. The RoboCup@Home league specialized on tasks that arise in a household environment. Here, the robot has to follow and recognize people, carry objects and react to speech commands from the user (Figure 3). The tasks the robot has to master at the SICK Robot Day usually change at every competition. So far, the robot had to drive through parcours as fast as possible or had to recognize numbers and navigate to targets in the order given by the identified numbers. Of course, these tasks were solved completely autonomously.

Though these fields of application are diverse the Robbie architecture was easily adopted to handle them. This easy adaptation is possible by integrating an application control module defining higher-level behavior. An application control module is basically a normal module with a state machine designed for a particular task (i.e. the desired higher-level

INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin

www.informatik2011.de

behavior). Depending on its state, the module triggers fundamental tasks (e.g. navigation, image processing) by sending the corresponding messages. A particular behavior can be activated on start up by loading a profile which includes the corresponding application control module.
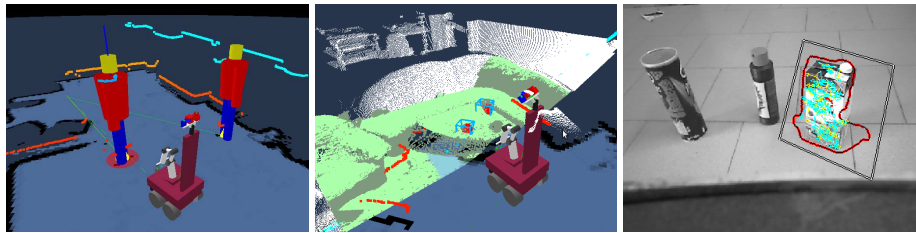

Figure 3: Higher level behaviors required to assist humans in daily life: detection, recognition and following of people (left), detection, recognition and grasping of objects (middle and right) for RoboCup@Home.

4.3 Scene Graph

The scene graph describes the geometric layout of the robot. It is defined in an XML-file and represents the robot using the tree data structure. The tree is modeled with globally unique nodes that contain transformations as accumulated rotations and translations. This mechanism allows a comfortable conversion between different coordinate systems (e.g. robot and world coordinates) for every part of the robot. Thus, object manipulation by actuators becomes easier as the position of the end effector can be calculated exactly.

In a multi-robot scenario (Section 4.5) the robots share a common map. All robots are included in the same scene graph. Therefore, every robots knows the positions of all other robots in the map.

The scene graph is managed by the `SceneGraphModule`. It processes all messages from actuators such as a pan-tilt unit or an arm and the robot's pose from mapping. After each change of the robot geometry, the module updates the scene graph and informs other modules by sending a `SceneGraphMessage`.

4.4 GUI

The Robbie framework offers a GUI that can be run directly on the robot or on a different computer via WLAN (please also refer to Section 4.5). The user interface is implemented using Qt 4 and OpenGL. It allows to monitor the system and control the most important modules and hardware components.

The 3D visualization included in the GUI evolved into a very important tool for understanding and improving the complex algorithms needed for a fully autonomous robot. It
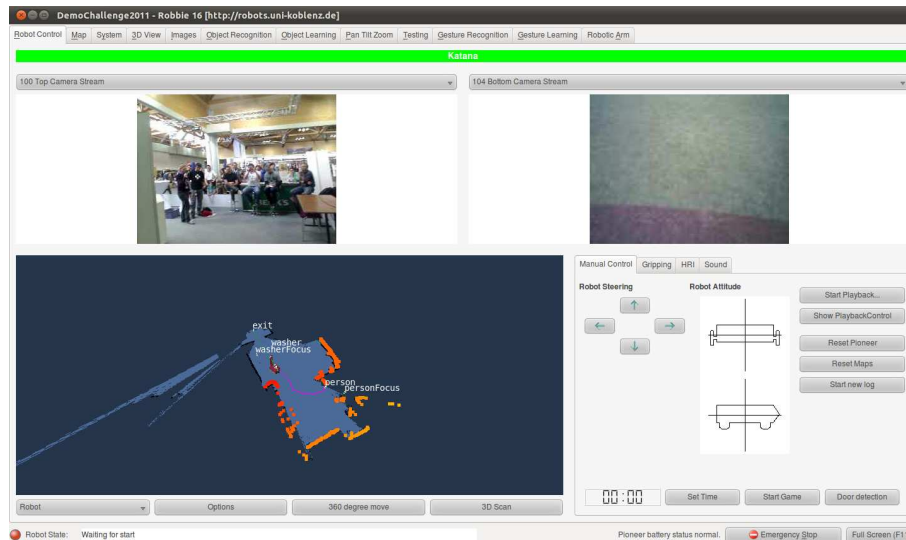
Figure 4: GUI during the Demo Challenge competition of the RoboCup@Home in Magdeburg (2011): top camera view (top left), bottom camera view (top right), 3D-view with scene graph of robot, map and navigation path, points of interest and laser data (bottom left). Control elements are located on the bottom right that allow to manually control the robot, gripping or human-robot interaction related tasks.

includes a model of the robot represented by the scene graph, obstacles detected by the laser range finder and optionally further relevant information such as the planned path or points of interest on the map (see Figure 4).

Further important features of our GUI are a 2D map of the explored region, a convenient interface to learn and recognize people and objects, manual control of the robot and its actuators and the possibility to create and send messages for debug purposes. The playback mode and the system monitoring also included in the GUI are described below in more detail.

4.5   Client and Server

The software can be used in a client-server scenario, where the server is running on the robot and clients can be connected from various other computers or robots. Messages are send over the network: storing and deserialization is accomplished as described in Section 4.6.

This mechanism allows to remotely monitor and control the robot. An application is found in the RoboCupRescue league. While the robot autonomously explores the arena and searches for simulated victims, it is monitored by a remote operator. When a potential victim is found, the robot asks the operator to confirm it. Further, the operator can stop the

robot if an emergency arises (e.g. if the robot enters an unstructured area).

Another application of the client-server scenario are multi-robot systems. As described in [TGNL+11] we are using a second robot for the participation in the RoboCup@Home league. The robots are programmed to show a cooperative behavior and exchange information via wireless LAN.

4.6 Logging and Playback

The creation of logfiles is an essential debugging tool for the development of algorithms that evaluate sensor data. It allows to run the algorithm repeatedly with the same data to guarantee valid evaluation results. It is also necessary for developers working with the robot, as the hardware resources are limited.

Logfiles are recorded when running the software server mode. In this profile the `Message-LoggerModule` writes previously defined incoming messages to a file. In playback mode the logfile is read and played by the `PlaybackModule`. The previously recorded messages are deserialized and send to the system core. From the system's point of view there is no difference between playback mode and operation with the actual hardware robot. During the playback of logfiles the GUI has the same layout as in server mode with an additional control window to control time and speed of the playback (see Figure 5). In this mode the hardware is replaced by virtual devices that can be controlled via the GUI.

When messages are changed logfiles may not be compatible anymore because data might be missing when trying to deserialize the message. In order to deal with this problem, a version system is used that stores a version number during the serialization of messages. The deserialization constructor of the message reads the version number and differentiates between the versions of the messages.

4.7 Debugging and Profiling

Another debugging tool is offered by the `ProfilerModule`. The corresponding GUI-tab displays relevant system information (Figure 6). The CPU usage of the whole application as well as of every loaded module is presented. Further, the number of messages in the inbound and outbound queue, the number of dropped messages and the module's status or, if available, the state of the state machine is displayed. This module offers a convenient way to find blocking threads or detect messages losses.

A special component of the architecture, the `Tracer`, records debug notifications in an HTML-logfile. Four types of notifications are distinguished: errors, warnings, infos, and system infos (the latter displays very detailed information). The system tab in the GUI allows to filter these notifications to record only a certain type (e.g. errors and warnings) in the HTML-logfile.

INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin
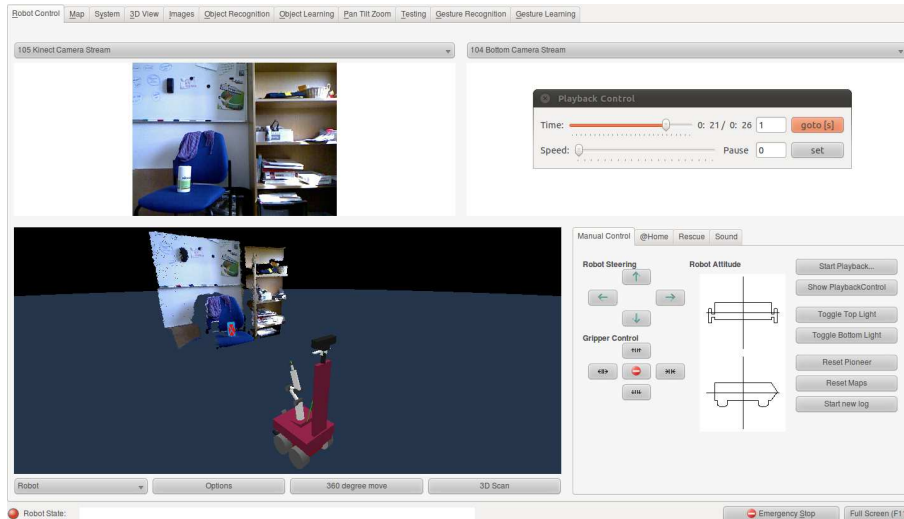
www.informatik2011.de

Figure 5: Graphical user interface during logfile playback. The robot's scene graph is shown in the 3D view together with a point cloud from a RGBD-sensor. A grippable object was detected and annotated in the 3D view (blue box with red point cloud). In the same window the image data from the sensor is visible and control elements to control the playback's position and speed.

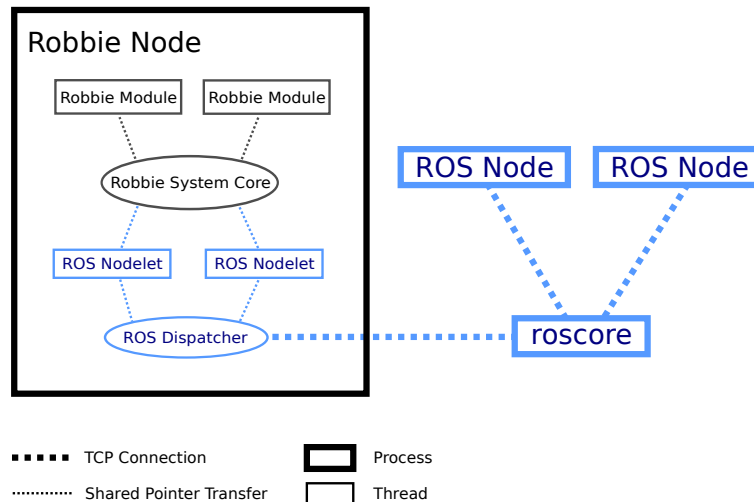Figure 6: Graphical user interface for software and system overview.

Figure 7: Schematic description of the communication between Robbie (black) and ROS (blue). The small dotted lines indicate shared pointer transfer for the communication between modules in the Robbie architecture. The large dotted lines are TCP connections that require data copy. The threads for modules and nodes in Robbie are encapsulated in a single process while roscore and other nodes run in separate processes.

5   Extensions

This Section describes the interface to ROS integrated in the Robbie architecture. Similar to Robbie, the software modules in ROS[5] (called *nodes* in the ROS nomenclature) communicate by sending messages. Since ROS is supported by a large community, that provides many solutions for general robotic applications, we developed an interface connecting Robbie to ROS. The main idea is to keep on the one hand the robust and extensive implementations of Robbie and on the other hand to enhance it with new features which can be easily integrated from ROS.

Figure 7 gives a schematic overview of the communication between Robbie and ROS. From ROS' point of view, the Robbie architecture is represented by a node. The changes in Robbie include an additional ROS dispatcher (besides the Robbie system core) that manages the messages transfered by ROS nodelets within Robbie. Nodelets in Robbie act as a linkage between the two systems. They can subscribe and publish topics (ROS message types, see [QCG+09]) via shared pointer transfer within Robbie and also handle the common Robbie messages. Beyond the functionalities provided by Robbie, nodelets are able to use the common ROS functionalities, which is depicted as ROS nodes communicating with the roscore in Figure 7.

The integration of ROS enables us to use the rviz visualization tool[6]. A screenshot of our

---

[5]Documentation of open-source Robot Operating System ROS `http://www.ros.org/`
[6]Documentation of ROS package rviz `http://www.ros.org/wiki/rviz`

INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin

www.informatik2011.de

robot with various sensor data in the 3D visualization environment is depicted in Figure 8. The rviz-interface was realized by a ROS nodelet in Robbie that receives scene graph, map, point clouds and other messages from Robbie modules. Within the nodelet this data is transformed into ROS message formats (e.g. tf[7] for scene graph) and made available to roscore. The rviz tool needs to run at the same time and subscribe to these messages. It allows us for example to monitor the robot's transformation tree, visualize laser data and the points of interest in the current map in a 3D scene. This visualization tool is suitable as a fast debugging tool since sensor data can be visualized easily. However, it can not replace the complete Robbie user front-end because it does not provide a full control software and overview of all sensors and system states at one glance.
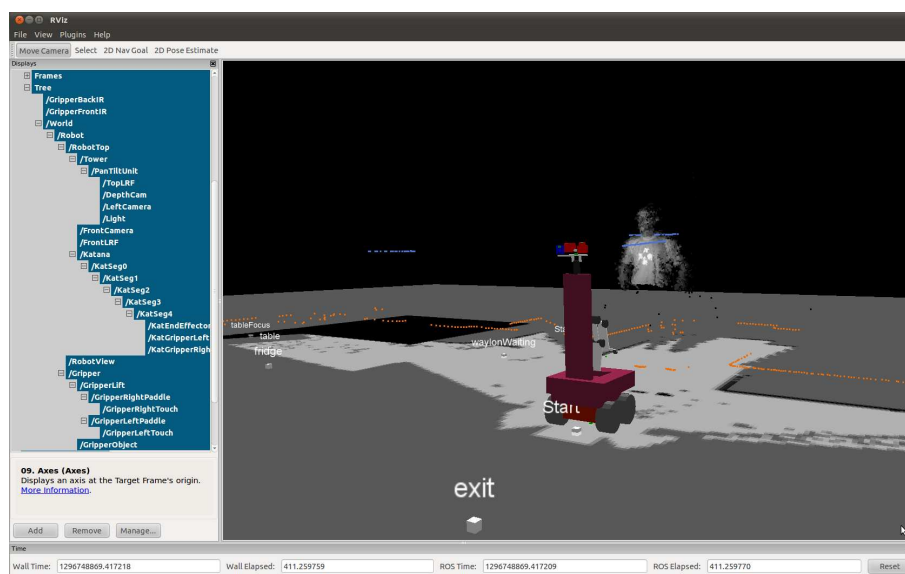

Figure 8: The rviz visualization tool showing various visualizations of sensor data. In this view points of interest, e.g. start and exit, an occupancy map, bottom laser (orange dots), top laser (blue dots), Time-of-Flight data and tf-data for the robot's scene graph are visualized.

6 Discussion and Future Work

In this section the limits and possibilities of the Robbie architecture will be discussed. We compare our system to ROS as it has become a commonly used and widespread robot architecture.

Both architectures provide similar concepts: modules in the Robbie architecture are related to the concept of nodes in ROS. Modules in Robbie run in separate threads as do ROS nodes if they are integrated in a nodelet manager as nodelets. Originally (i.e. without

---

[7]Documentation of ROS package tf `http://www.ros.org/wiki/tf`

the integration in a nodelet manager) ROS nodes would run in separate processes. As processes do not share memory, this may result in a large amount of data that needs to be copied for the communication between nodes. In the Robbie framework all modules are encapsulated in one process, but run in different threads. Therefore, the communication between them is accomplished via shared pointer transfer. This behavior is also depicted in Figure 7. Of course running the whole architecture in one process has also a downside: if one of the modules does not work properly the whole process is affected (e.g. if a device is not initialized correctly on startup the whole system needs to be restarted). Consequently, a crash in one of the modules leads to a crash of the whole system.

However, the numerous advantages are far more significant. The compact implementation of the software is the basis of our custom made GUI that allows to control and monitor the whole system. Contrary, ROS nodes often come with a separate user interface or command line output. Since fast setup, configuration and debugging are essential tasks when participating at competitions like the RoboCup the Robbie architecture is better capable of suiting our needs. On the other hand, the tools provided by ROS facilitate developing and prototyping of new components.

The usage of messages as interfaces to modules provides an abstraction of functionalities and reduces the complexity for developers. In ROS messages are related to certain topics that define the type of a message. Robbie messages are loosely related to topics, but topics can be defined by developers when designing a new message if they are necessary. Apart from the communication by messages ROS provides so called *services*. Services define a certain communication behavior between nodes: one request needs to be followed by a response. In Robbie this behavior can be modeled by a module subscribing and sending messages that act as request-response services, but remote procedure calls per se are not available.

Our architecture needs to be easy to learn and maintainable as it is used for educational purposes. Hence, in contrast to ROS, it is only programmable in C++, which may constrain advanced programmers, but helps to keep readability. Launch files in ROS define which nodes and configuration values should be started. Profiles in the Robbie configuration are the counterpart and can be easily used to combine modules and configurations for a certain task.

In future we plan on using more ROS components to communicate with hardware instead of developing own hardware interfaces. We believe that it is better to rely on hardware components developed and maintained by a large community. This will allow us to focus on the development of new algorithms for image processing, human-robot interaction and task planning.

Further, we intend to extract suitable components from the Robbie architecture and model them as ROS nodes like it was done for the talking head introduced in [TGNL+11]. If such components prove themselves useful we plan on publishing them and thus contribute to the ROS community. Hence, both systems may benefit from each other if they are combined in a reasonable way: ROS to easily integrate new devices and features and share software with a large community, Robbie to run a compact, easy to learn and maintainable system with extensive implementations and features with visualization and control interface.

INFORMATIK 2011 - Informatik schafft Communities
41. Jahrestagung der Gesellschaft für Informatik , 4.-7.10.2011, Berlin

www.informatik2011.de

7   Conclusion

We explained the fundamentals of our message-based robot architecture and its further enhancements. Robbie offers a modular structure and is easy to maintain, extend, and adapt to different robot platforms. The extension of Robbie with the widely spread robot platform ROS was introduced. ROS is a generic architecture, but may contain much overhead and be very complex. However, it is a widely spread platform that offers lots of open-source implementations by a large community. We showed that the integration of ROS into Robbie is accomplished while keeping an easily understandable and maintainable platform for educational purposes. This allows us to use prefabricated components from ROS (e.g. the control of devices), and on the other hand focus on our research (e.g. on sensor processing and visualization).

References

[BKM+05]   A. Brooks, T. Kaupp, A. Makarenko, S. Williams und A. Oreback.   Towards component-based robotics. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 163–168, 2005.

[Bru01]   Herman Bruyninckx. Open Robot Control Software: the OROCOS project. In *ICRA*, pages 2523–2528. IEEE, 2001.

[CCIN08]   Daniele Calisi, Andrea Censi, Luca Iocchi und Daniele Nardi.   OpenRDK: A modular framework for robotic software development.   In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2008)*, pages 1872–1877. IEEE, September 2008.

[LHP+11]   Dagmar Lang, Marcel Häselich, Martin Prinzen, Simone Bauschke, Alexander Gemmel, Julian Giesen, Ruwen Hahn, Laura Haraké, Paul Reimche, Guido Sonnen, Matthias von Steimker, Susanne Thierfelder und Dietrich Paulus. RoboCupRescue 2011 - Robot League Team resko@UniKoblenz (Germany). Technical report, Universität Koblenz-Landau, www.uni-koblenz.de, 2011.

[QCG+09]   Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler und Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[Sie02]   Johannes Siedersleben.   Quasar: Die sd&m Standardarchitektur.   Technical report, sd&m Research, 2002.

[TGNL+11] Susanne Thierfelder, David Gossow, Carmen Navarro Luzón, Sebastian Nowack, Nico Merten, Susanne Friedmann, Lydia Rebecca Weiland, Daniel Mies, Julian Giesen, Marcel Häselich, Dagmar Lang und Dietrich Paulus.   RoboCup 2011 - homer@UniKoblenz (Germany).   Technical report, Universität Koblenz-Landau, www.uni-koblenz.de, 2011.

[USEK02]   H. Utz, S. Sablatnog, S. Enderle und G. Kraetzschmar. Miro - middleware for mobile robot applications. *Robotics and Automation, IEEE Transactions on*, 18(4):493–497, Dezember 2002.