Bringing System Introspection to Mobile Devices for System Analysis

Master Thesis
Faculty of Technology
Bielefeld University

by: Andreas Kipp
akipp@techfak.uni-bielefeld.de

supervisors: Dr.-Ing. habil. Sven Wachsmuth
Dipl.-Inform. Frederic Siepmann
Dipl.-Inform. Ingo Lütkebohle

March 30, 2011
Contents

Abstract iii

1 Introduction 1
  1.1 Motivation: Developing Complex Robot Systems .......... 1
  1.2 Bringing the System to Mobile Devices .................. 2
  1.3 Use Cases: What a Developer does with the System ........ 3
  1.4 Outline ............................................. 3

2 Related Work 5
  2.1 Remote Control and Telerobotics ......................... 5

3 Requirements 9

4 System Environment and Tools 13
  4.1 Robot Platforms: BIRON and ToBI ........................ 13
  4.2 Bielefeld Sensor Actuator Abstraction Layer (BonSAI) .... 14
  4.3 Operating System on the Mobile Device .................... 14
  4.4 XMPP, Openfire and the SMACK Library .................... 16

5 System Design 20
  5.1 Communication Interface ................................. 21
  5.2 Probe - The Server Side ................................ 24
  5.3 Client - Observation Window to the Robot ................ 30

6 Implementation 33
  6.1 The Communication Interface ............................... 34
  6.2 The Probe - Implementation of the Server Side ............ 36
  6.3 Implemented Components of the Robot ...................... 41
  6.4 Client .................................................. 42
    6.4.1 Mobile Devices - Playing with Android ............... 43
    6.4.2 Java Client on Desktop Machine ....................... 48

7 Evaluation 54
  7.1 Preconditions for the Evaluation ........................ 54
  7.2 Measuring the Different Latencies ....................... 55
    7.2.1 Goal .............................................. 55
7.2.2 Preparations ........................................ 55
7.2.3 Method ........................................... 56
7.2.4 Results ........................................... 57
7.3 Observing the CPU Load of the Robot System .... 59
  7.3.1 Goal ............................................ 59
  7.3.2 Preparations .................................... 60
  7.3.3 Method ........................................ 60
  7.3.4 Results ....................................... 61

8 Conclusions ........................................... 64

9 Outlook ............................................. 66

Bibliography ............................................ 69

List of Figures .......................................... 71

List of Tables and Listings .......................... 73

Appendix ............................................... 74
Abstract

System Introspection is essential to understand the inner processes controlling a robot which is acting in an autonomous way. To get an idea of the inner states of the controlling components a developer does not want to interrupt the running system. Moreover analyzing a dozen of complex log files once the robot has ended performing a task consumes a lot of time. This thesis presents a design and an implementation of a system for observation that can be attached to a robot system. The system will collect data of the robot and forward it to a client for presentation. XMPP is used as communication framework which provides functionality for exchanging data over a network. The described system was successfully evaluated by attaching the system to the ToBI platform.
1 Introduction

Observing complex systems like robots can be difficult whenever the integrated components are not speaking one language. In the context of this thesis a system is developed that helps to combine different information queues and gives an inside view of a robot system.

1.1 Motivation: Developing Complex Robot Systems

ToBI is a robot platform descended from the BIRON platform (see chapter 4) created and developed at the University of Bielefeld. ToBI can behave autonomously in an unknown environment and perform different tasks. Some examples of these tasks are following and interacting with persons, exploring unknown environments or recognizing objects and manipulate them with a gripper. Every single task needs several components running on the robot. To achieve an autonomous behavior, the different components, running on different hardware systems must be combined.

ToBI performs different tasks at the Robocup@Home\footnote{http://www.webcitation.org/5wrZXRnjK - Robocup@Home} contest. A concrete example for a task is to follow a person guiding the robot around an unknown room. For this task the robot uses different sensors and actuators (e.g. cameras, laser range finder). With algorithms to extract persons from the data of each component, the robot is capable of following the person by using its navigation components to drive towards the person. All the components publish information that needs to be aligned in time. For example, the position of faces must be compared to the positions of legs. By combining the data, the robot can analyze the data and make a decision on further actions, like moving towards the person.

When a developer has to compose the interaction between different components, a general understanding of each component is needed. A component is controlled by a piece of software which is often not created by the developer itself. To understand how a component works, the developer can try to read and analyze the output of each component. There is no standardization on
output, so each component is offering its own kind of information with its on frequency. This frequency can be quite high, and the human perception is often not capable of reading and comprehending as fast as a component (Dyson and Haselgrove, 2001). Also the plotted data is often not in a human readable form. To make the data readable it must be filtered or transformed somehow. When observing the robot the developer often needs more then one component. So two or more windows displaying the components output have to be monitored at once. The ToBI platform runs more then 20 components in parallel, which makes observing without collecting and filtering the data to fewer views nearly impossible.

Besides, the idea behind robots like ToBI is to make the robot behave in an autonomous way. Many robot systems are designed as closed systems and often there is no way to observe the inner processes from the outside. If the robot stops acting as expected, the developer has to move to the robot and check its components. For example a worst case scenario can be an experiment with a human-robot-interaction where the robot suddenly stops interacting. Checking the components of the robot can corrupt all recorded data due to the fact that the robot may recognize not only the subject but also the developer.

My motivation is to make observation of the inner processes possible by creating a system that can be attached to the robot. This system should offer views on the robots components and let the developer choose what he wants to observe. These information should be filtered to the important elements and then be presented on an external device. Due to the limited resources of the robot the system must be designed lightweight and while active it should not influence the normal behavior of the robot.

1.2 Bringing the System to Mobile Devices

Many control and observation systems (see chapter 2) use personal computers to present published data. Sometimes there are situations where the developer is separated from the robot and a personal computer or laptop is not available (e.g. while on business trip). Nowadays mobile devices are capable of using the internet for communication, so the idea is to bring the information to such a device. With improving performance many devices are capable to process a lot of information at once and can be used for the robot observation mentioned.

My idea was to integrate the output of a system collecting the data of components of a robot on such a mobile device. Filtering the data on the system collecting allows to minimize the data to be exchanged. With the reduced data
even complex information can be transferred to and displayed on a mobile de-
vice. The communication also can be handled over the internet, which allows
the system to send the data to devices all over the world. The developer can
be miles away to check the robots state and give advice for users at the local
side. Like the resources on a robot the resources of mobile devices are surely
limited. The system needs to be lightweight and save resources as much as
possible. For example, when observing the robot the mobile phone should not
use up all the battery within 5 minutes.

1.3 Use Cases: What a Developer does with the System

In the appendix of this thesis the use cases for the interaction between a de-
veloper and the system attached to the robot are described. Important use
cases are the request for information of the components on the robot and the
request for components to be logged.

The use case “Request a component” explains the way how a component is
requested from the robot. When the expert is successfully connected to the
observation system, he can request the list of components offered by the system.
Selecting a single component will trigger the component on the probe to start
publishing its information. The information is displayed as different views on
the mobile device.

The use case “Request components to be logged” starts a logging cycle on the
probe. This cycle collects data from selected components, filters it to the needs
of the developer and prints them to a logging file. The developer can select
from available components on a selection view and execute the cycle. While
the system probe is collecting, it creates temporary files for backup. When the
developer stops the cycle or disconnects from the probe, the collected data will
be saved as a unique file on the file system.

1.4 Outline

Components running on a robot can be hard to observe. In this thesis a system
is designed and implemented, that allows to filter information from components
of the robot and collect them into several views on external devices like mobile
phones. Chapter 2 introduces some work that is related to the the system
created in this thesis. Chapter 3 describes the requirements for the system
and chapter 4 explains the systems environment. Chapter 5 covers the design
of the system and the decisions made for the communication interface. The implementation is described in chapter 6. This chapter shows the implementation of the system attached to the software of the robot and two different clients created for the representation of the information. The capability of the implementation is tested and described in chapter 7. Goal of the evaluation was to show the occurring system load on the robot running a task and the latency for exchanging data on the mobile device. Chapter 8 gives a conclusion on how the implementation worked out followed by chapter 9 giving an outlook on future work to be done with the system.
2 Related Work

This chapter gives an overview of relevant topics related to the idea behind the system created in the context of this thesis. Naturally related topics are the remote control of a system and the control of robot systems with telerobotics. Work done in these fields is described and some difference to the system designed are mentioned.

2.1 Remote Control and Telerobotics

Sheridan (1992) gives an introduction to the paradigm of telerobotics. The main idea is to control a system by using a remote system. On the remote side a supervisor controls a system which is connected to the system on the local side with a barrier inbetween. Examples for this barrier can be a local network or even the internet. The remote system provides commands the supervisor can use to manipulate the local side. Selected commands are send to the local system over the barrier. The local system takes these commands and translates them into commands for the execution of different processes (Figure 2.1).

![Figure 2.1: Basic concept of telerobot supervisory control - On the left side the supervisor selects commands to be send and checks information received from the local side. The commands are sent over the barrier to the local system shown on the right. The local system translates the commands for execution on the robot. Information from the robot are send back to the supervisor again over the barrier. [From Ferrell and Sheridan (1967)]](image)

Today a lot of remote control systems can be found in the industry, in scientific work or used for educational purpose at schools. Marin et al. (2005)
created a telelaboratory where two robot arms are remotely programmed with an multimedia user interface (Figure 2.2). The main idea of the laboratory was to evaluate the capabilities of the user interface for remote control. At a local laboratory the setup contains two robot arms monitored by several cameras. A server allows connected users on the remote side to send commands for manipulation on the local side. To observe the manipulation the remote interface offers different views of the cameras and the data collected at the local side. The communication between the local and the remote side was established over sockets using TCP/IP. The remote side used a given Java library handling all the communication overhead. Also the library consists of a skeleton for creating experiments with predefined commands. To evaluate the system test experiments were designed and the latency between sending commands and the execution in the laboratory was measured. The results showed that sending data over the internet can be difficult due to the availability of bandwidth at a given moment. The analyzed data showed that there is no guarantee that a command send from the remote side arrives at the local side in a minimum period of time.

![Image](image1.png)

**Figure 2.2:** Observation software for a telelaboratory - Image b shows the interface which allows to create experiments that manipulate the two arms shown in image a. The system offers different views of the local side by using several cameras. Experiments are designed on the remote side and exchanged via the internet [From Marin et al. (2005)]

There are some points of the telelaboratory that are similar to the system
designed in this thesis. Like the setup of the laboratory the designed system can be used to observe a system by collecting data on the robot side and forward it to a remote side. One difference to the laboratory is the high coupling between remote and local side. The remote system for the laboratory was used to control the system. The system designed in this thesis will primarily be used to observe the system to get an idea of what is processed inside the robot at the current moment. Also the attached system should not directly influence the robot due to the fact that the robot should still behave in an autonomously way.

More similar is the setup designed by Lin et al. (2007). Their system controls a SONY Aibo\(^1\) robot over the internet personal digital assistant (PDA). There where five different servers on the robot side used to control and publish data of the different components. On the client side according modules were offered that were directly connected to their counterpart on the robot. All Components are displayed on a graphical user interface on the PDA. With the mobile device the user was able to control the movement of the robot and to set the angle of the head containing the camera.

![Figure 2.3: Sony AIBO controlled with a HP iPAQ rx1952 over the internet - The control software on the device allows to navigate the robot and monitor the onboard camera [From Lin et al. (2007)]](image)

The capability of the system was tested by navigating the robot through a predefined course. While navigating the round-trip time of the data exchanged between robot and PDA was measured. The experiment was carried out at different day times and showed that the round-trip time was not constant due to sharing the network bandwidth with other systems (Figure 2.4). This system offers several services to control the robot and observe several sensors.

\(^1\)http://www.webcitation.org/5wiCWyI0M - SONY Aibo
Like the telelaboratory there is a high coupling between the remote side and the robot at the local side. The information sent to the PDA were presented inside one screen. On a mobile device the resources and the space to display data are limited. For the approach made in this thesis the system presents each component on a different view and lets the developer choose which component he wants to observe. This also helps to minimize latency on exchanging data. By only sending data of requested components the payload of each message can be minimized.

![Diagram](a) Course for the PDA evaluation  (b) Results of the PDA evaluation

**Figure 2.4:** Measuring the capability of the remote control - Image a shows the course the user has to navigate through. Image b shows the measured average round-trip times and the traveling times at different day times.[From Lin et al. (2007)]

A great difference between the systems mentioned and the system created in this thesis is the part of observation. Other than a pure remote the system designed is attached to a running system and collects data of the components of the robot. With this data the internal processes of the robot can be presented which can help to understand what the robot is currently planning and performing.
3 Requirements

In the context of this thesis the development of a system is described that can be attached to the software of a robot for observation of running components, without influence on the normal behavior. This chapter describes the requirements given for such a system. There are several elements to be distinguished in this thesis:

- **robot** the system to observe and to collect information from
- **probe** the application attached to the robot software for observation
- **client** the application running on an external device communicating with the probe
- **developer** the expert using the application to observe the robot

Who is the Expert?
In the context of this thesis an expert is the person developing the components of the robot. He extends existing components and adds new components to the robot. The expert combines all available components to make the robot behave in an autonomous way. He is not the end user interacting with the robot. The only interaction an expert does is for testing purpose of the components developed.

System Requirements of the Developer
To get an idea of the inner processes of the robot some requirements are given by the developer.

**Requirement 1:**
Only present information requested and filter them to the important parts with a human readable frequency

When observing a component on the robot, the information available can be quiet complex and offers more detail then needed. The data is often presented
with a very high frequency which makes it hard for a human to observe a component of the robot.

**Requirement 2:**
Send and receive information with minimized delay

The delay between the data send and the time of receiving it on the client must be minimized. A high delay can make the data obsolete which makes a decision on the robots state dispensable.

**Requirement 3:**
Change component views at any time

Often the expert wants to check other components on the run, to make sure changes on one component do not influence other components.

**Requirement 4:**
Cooperative work without influencing other observers

Often more then one expert is working with the system. Every expert needs his own view onto the components without interfering other experts. A multi-user environment is therefore beneficial.

**Requirement 5:**
Control logging of filtered information on command and create unique logging files

Many components already create logging files that can be analyzed when the robot has ended its current performance. These files are written once per component and must be aligned by the developer himself. Also the components log their data when the robot starts until it has stopped working. For a post-hoc analysis the probe must collect and filter information from components selected by the developer. The developer must be able to initiate the process and stop it on command. All collected data must be saved into a single file for each developer.

**Requirement 6:**
Extension must be possible

Due to the changes of the components of the robot, extension of the system must be possible. New components must be easily attached to the system.

**Requirement 7:**
Establish a wireless connection to the probe

Due to the need that the robot acts in an autonomous way, the expert does not want to move towards the robot for observation of components. The connection
to the probe needs to be wireless. For example this supports remote observation
while the robot is participating in an experiment. If possible the system must
be available over the internet. This allows the expert to monitor changes even
if he is not at the local side of the robot.

**System Requirements of the Robot**

Though the system is attached to the software of a robot, some requirements
are given by the robot itself.

**Requirement 8:**

*Minimum influence on the robot and inactive when not used*

On the robot a lot of components are executed in parallel. Many of these
components consume resources which are limited by the robots system. The
probe attached should only use a minimum of the resources available. While
the probe is active the normal behavior of the robot should not be influenced
by any means necessary. To achieve an minimum influence, the probe should
only be active whenever a developer is connected to the probe and deactivated
if not used.

**Requirement 9:**

*No hard coded connection between robots components and the system probe*

Due to improving technologies the components on the robot can change. The
probe must capsule the connection reading and controlling components of the
robot to make changes on on both sides possible without affecting the whole
system.

**System Requirements of the Probe**

The probe publishes information to observe the robot. It is attached to the
software of the robot and uses the resources available.

**Requirement 10:**

*Observation of components must allow parallelism and multi-user observation*

Each component inside the probe should be running on its own thread. This
offers parallelism and allows multiple users to observe different information
queues.

**Requirement 11:**

*Well defined communication interface for implementing different communication frameworks*
Due to improving technology the used communication may be replaced later on. To make the communication exchangeable an interface must be created that allows to change the underlying communication framework.

**System Requirements of the Client**

The client displays the information published by the probe within different views.

**Requirement 12:**
**Application running with minimized resource usage to save system resources**

The resources on a mobile device are limited. The application has to be lightweight and should not block the device due to excessive resource usage. For example the battery drain should be reduced by decreasing the needed CPU usage.

**Requirement 13:**
**Well defined communication interface with same structure then the probe**

The application on the client must use the same communication interface the system probe uses. This ensures that both systems use the same base for communication. Like the system probe this interface must be interchangeable.

**Requirement 14:**
**Create different views that can be changed on request and provide a good usability for the developer**

Switching the views must be possible at any time. The usability of these views must satisfy the experts requirements.

**Requirement 15:**
**Support for extension according to changes on the probe**

Due to changes on the probe the views on the application must be extensible.
4 System Environment and Tools

This chapter describes the environment for the system implemented in. It will give an introduction of the robot platform used and the interface used by the platform for connecting hardware components. Also the operating system of the used mobile device and the communication framework is covered. The description of the environment is given at a higher level and provides no details on lower functionalities.

4.1 Robot Platforms: BIRON and ToBI

In the context of this thesis the probe and the client where both developed to work with the ToBI platform (Figure 4.1). ToBI is a project of the CITEC$^1$ Central Lab Facilities at the University of Bielefeld$^2$ descended from the Bielefeld Robot Companion platform (BIRON). ToBI is the platform of the Team of Bielefeld used to participate in the Robocup@Home competition (Wachsmuth et al., 2011). The platform uses several components controlled by two laptops in the backpack of the robot. With this components and the software developed at the Central Lab ToBI can behave in an autonomous way in an unknown environment. For the communication between the software and the hardware ToBI uses the BonSAI interface (see chapter 4.2). Examples for the sensors and actuators integrated on the platform are the 180 degree laser range finder (SICK$^3$), several cameras for visual perception, a Katana-Arm$^4$ for grasping or a stereo microphone for voice recognition placed on top of the robot. At the Robocup@Home contest ToBI performs tasks like following persons, navigating in an unknown environment or grasping objects with its gripper. The operation system on both notebooks is Ubuntu Linux and the software behind the different components is written with the programming languages C++$^5$ and Java$^6$.

\footnotesize{$^1$http://www.cit-ec.de/ToBI - ToBI Team of Bielefeld}  
\footnotesize{$^2$http://www.uni-bielefeld.de/ - University of Bielefeld}  
\footnotesize{$^3$http://www.webcitation.org/5wiFVLrFA - SICK Laser Range Finder}  
\footnotesize{$^4$http://www.webcitation.org/5wiFjsR2R - Katana Arm}  
\footnotesize{$^5$http://www.webcitation.org/5xY8pI0Yj - C++ Reference}  
\footnotesize{$^6$http://www.webcitation.org/5wu0WbS5I - Java Technology Network by Oracle}
4.2 Bielefeld Sensor Actuator Abstraction Layer (BonSAI)

The BonSAI interface provides an abstract view to the sensors and actuators on robot platforms. A Java API allows developers to control actuators or query sensors from a higher level. The API itself controls the communication between higher and lower levels (see Figure 4.2). BonSAI also provides cross-modal sensors that combine several components into one sensor class. One example for a cross-modal sensor is the PersonTracker, which combines laser data with camera images. An example for a cross-modal actuator is the NavigationActuator combining components to control the navigation of the robot (e.g., driving, obstacle avoidance, path planning).

4.3 Operating System on the Mobile Device

For mobile devices the operating system of choice used in this thesis is the android operation system\(^7\). Android is a software platform developed by the

\(^7\)http://www.webcitation.org/5wiFsqc0j- The Android Operation System
Figure 4.2: The different layer of BonSAl - The top layer contains elements for complex behaviors, the middle layer holds functional components for base tasks and the bottom layer controls low level hardware components [From Wachsmuth et al. (2011)]

Open Handset Alliance\(^8\). The architecture behind Android provides an operating system, middleware and several key applications placed in different layers (Figure 4.3). Android applications can be written with the Java programming language through the Android SDK\(^9\). Android comes with support for wireless connections (Bluetooth, WIFI, EDGE, 3G), hardware components (cameras, GPS, accelerometer) and media support. Android relies on a Linux kernel which acts as abstraction layer between hardware and software.

Android offers an open development platform to allow developers the creation of rich and innovative applications. Each Android applications runs within its own life cycle (Figure 4.4) and is called activity. An activity consists of a class containing the logic and an XML file describing the layout of the activity. The life cycle describes the different states of an activity presented by different methods. The onCreate()-method is called once an activity starts and initializes the components. The onStart()-method is triggered when all configuration of the creation state has ended. Once started the activity runs in the foreground of the device within the onResume()-method. In this method the interaction with the user takes place. If the user stops the activity it is not directly destroyed. The state of the activity is changed to onPause() and then onStop(). In this state the operation system places the activity in the background. If the activity is requested again the onRestart()-method is called. If the activity has been terminated by the operation system due to the need of resources the onCreate()-method is called if the activity is requested. If the activities interaction is finally finished the onDestroy()-method is called and the activity ends. One application can contain several activities to provide interaction with the user. The operating system therefore decides which activity is running in foreground and sends unused activities to the background.

\(^8\)http://www.webcitation.org/5wiFwDWLg - Open Handset Alliance

\(^9\)http://www.webcitation.org/5wrOKYh2w - Android Developer Guide
Figure 4.3: The architecture behind android - The top layer contains standard applications. The application framework offers interface for high level components. The libraries contain functionality used by the application framework. The Linux kernel controls the low components and the hardware [From the Android Developer Guide]

4.4 XMPP, Openfire and the SMACK Library

For the communication between the probe and the client a communication framework is needed. In this thesis the communication framework of choice is the Extended Messaging and Presence Protocol (XMPP\textsuperscript{10}). XMPP is an open-standard communication protocol based on the Extensible Markup Language (XML\textsuperscript{11}) published by the XMPP Standards Foundation (XSF\textsuperscript{12}). Saint-Andre (2005) gives an introduction on streaming XML with XMPP.

XMPP uses streams to exchange XML elements between two entities over a network. The first-level child element send over such a stream is called stanza (Saint-Andre, 2004). There are three core stanzas for communication:

- `<message />` used to push information from one entity to another
- `<presence />` send information about network availability of one entity
- `<iq />` for info or query information between entities

\textsuperscript{10}http://www.webcitation.org/5wiG3xxE - Extended Messaging and Presence Protocol
\textsuperscript{11}http://www.webcitation.org/5xNK6MKGA - Extensible Markup Language
\textsuperscript{12}http://www.webcitation.org/5wICGjXV - XMPP Standards Foundation
A client connects and opens a stream to a server and the server opens a stream back to that client. Once connected the server sends the rooster to the client. The rooster contains all other clients known to the client. When the rooster is loaded, the online presence is delivered to all clients of the rooster. Each client returns its own presence. Whenever a message is send to another client, the message is send to the server which then forwards the message to the recipient. If the recipient is not in the domain of the server, the server negotiates a server-to-server connection to the recipients server which delivers the message to its client. Figure 4.5 shows a typical message session containing elements from connecting to a server and exchanging messages.

This work uses the message stanza for data exchange between the probe and the client (see chapter 6.1). The data exchanged is primarily using simple data structures. Most of the data can be packed into an XML structure and
Figure 4.5: Typical instant messaging session - 1) Establish connection to the server "Montague.lit". 2) Request rooster containing known friend accounts. 3) Make own presence available to contacts of the rooster. 4) Receive presence of contacts from rooster. 5) Exchange message with another contact [From Saint-Andre (2005)]

attached to a message stanza. Listing 4.1 shows a typical message stanza.

A message contains the name of the sender, the recipient the message is forwarded to and the body containing the text to be send. XMPP also allows to extend these message with XML elements.

XMPP Library for Java
The probe and the client are both implemented in Java (see chapter 6). To use XMPP within Java the SMACK\footnote{http://www.webcitation.org/5wiKn3w0C - SMACK library by ignite realtime} library is integrated. SMACK offers an open source API with functionality for connecting to a server and exchanging messages between different clients. There is also a port of the library for the android operation system. SMACK allows to extend the message stanza with value pairs added as properties.
<message from="user1@robot" to="user2@robot">
  <body>Hello user 2!</body>
</message>

Listing 4.1: Typical stanza for a message - The message stanza holds the attributes for the sender and the recipient. The text to be send is placed inside a body element.

XMPP Server

Public XMPP servers can be found on the internet. For testing purpose the probe and a local XMPP server have been installed on the same laptop. As XMPP server used for the communication between the probe and the client the Openfire<sup>14</sup> server is used. Openfire is an open source server written in Java with supports for XMPP communication.

<sup>14</sup>http://www.webcitation.org/5wiKd8MIK - XMPP Server by ignite realtime
5 System Design

The following chapter covers the design of the probe, the client and the communication interface used for communication between both entities. The design of each element is described in detail here. To distinguish the components of the robot and the probe, components running on the probe will be referred to as component threads. Components of the robot will be referred to simply as components.

Requirements revisited

A robot acting in an autonomous way with a lot of components running in parallel is limited in resources it can spare. Especially ToBI running over 20 components controlled by two laptops is limited. One of the main requirements for the probe and the client is the minimized use of resources available. To minimize the influence on the behavior of the robot this requirement must be considered permanently. Another important requirement is to create an exchangeable communication interface. Due to changes of the components, the payload of data is not fix and can change very fast. For example the frequency of information retrieval may be higher then the used communication technology can handle. A promising way to deal with this problem is to capsule the communication interface beginning at the design. Due to the fact that often groups of developers integrate components the system needs to support multiple users observing at the same time which requires parallel work to be taken into consideration. Also extension of the system must be possible. When new components are attached to the robot these components must be easily added to the probe and the client.

Overview of all System Elements

Figure 5.1 shows the architecture of the whole system containing the robot, the probe and the client. To capsule the robots components the probe uses the BonSAI interface (see chapter 4.2). The probe and the client use their own
implementation of the designed communication interface. Messages between
the probe and the client are exchanged via a communication server.

Figure 5.1: Overview of the designed architecture - On the left side
is the robot providing its components by using the Bon-
SAI interface. Probe and client are connected via the
communication server.

5.1 Communication Interface

To handle the communication between the probe and the client a communica-
tion interface is designed to be implemented on the different entities. The idea
behind the interface is to capsule all needed communication overhead to this
interface. Also by implementing the interface on both systems the same un-
derstanding of message exchange is guaranteed. The communication interface
controls the information flow between connected entities and the communica-
tion server. The main functionalities that are needed are establishing a
connection to the communication server and the exchange of messages among
connected entities. A handshake mechanism by sending a message and waiting
for a response is used to acknowledge the connection between two entities. To
monitor the connection between the entities a beacon is used. This beacon
must be exchanged to assure that both sides are available.

Entities in this thesis are the probe and the client. At startup both entities
must connect to a connection server by using a connection configuration. The
configuration is distinguished from the source by using a configuration file.
When the client is connected it sends a message to the probe requesting to
establish a communication channel. The probe acknowledges the request and
adds the client to a list of known clients. This list is held by the probe for
a whole runtime. Once the connection between probe and client is created,
the client can request data of component threads controlled by the probe. Re-
quests are forwarded to the components which send data to the client. Figure
5.2 shows how a message is created and transferred from the probe over a
communication server to the client.

21
Figure 5.2: Message exchange between probe and client - On the left side is the component thread publishing new data for connected clients. The data is send to the communication interface that handles all the communication with the communication server. The server itself forwards the message to the client. The communication interface on the client side notifies all observers. Each observer then checks the message and processes it if needed.

Using Publish / Subscribe

The mechanism behind exchanging the data is using the publish/subscribe paradigm (Eugster et al., 2003). One entity subscribes to a publisher. The publisher collects data and forwards it to all subscribers. The probe offers services and publishes their data on request. The client can subscribe to such a service. In contrast to a standard implementation of the publish/subscribe paradigm the publisher also knows the subscriber and vise versa. A component thread is not only acting as a publisher. Each component thread can also receive commands which can change how the data is published. These commands may also change the state of a component thread. An example could be a component that has several parameters for changing the process of filtering the data presented. The client knows the publisher and can directly send commands. Also this can be used to minimize the need of control overhead and decrease the delay between sending and receiving a command. Specific commands are forwarded to a component thread which can directly use the commands for further processing.

To notify entities of incoming messages, the communication interface is using the observer pattern (Freeman et al. (2004), pages 37-78). For the exchange of data between the probe and the client, each message created contains the TAG of the sending element and the TAG of the component that receives the data. A TAG is unique to the system and allows to identify its elements. Whenever a new message is received, the interface notifies its observers. Each element observing this interface checks if the message contains its TAG and then reads
and processes the data. Each observer is inactive as long as no messages are forwarded. This allows the system to save resources normally used for actively monitoring the communication (e.g. polling). Component threads do not need to permanently check if new messages have arrived. Figure 5.3 shows the communication flow by using the observer pattern.

![Communication flow with observer pattern](image)

**Figure 5.3:** Communication flow with observer pattern - The communication interface listens to new messages from the communication server. New messages are forwarded to the interface. Once received, the interface notifies all observers. Each observer checks the message and processes the data if needed.

**Monitoring the Connection**

To assure that the connection between probe and client is permanently alive, a monitoring system is needed. To monitor the connection a beacon system is used. Henry et al. (1998) used a beacon to monitor the state of a spacecraft. The spacecraft sends tones with defined frequencies. From this frequency the state of the spacecraft can be derived. For this thesis a similar approach is used. The idea of a beacon was to save system resources if the connection between the probe and the client got lost. A beacon allows the probe to monitor the connection to each client with minimal effect on the connection itself. When the client connects to the probe, the probe will send a beacon to the client in a given frequency. The client has to answer the beacon by sending a new beacon to the probe. This technique offers several advantages. It allows the probe to manage its component threads. If a client is disconnected due to missing beacons all component threads sending data to the client can be notified about the disconnect. Besides the beacon that is transferred additionally contains a timestamp, which allows to check the round-trip time between probe and client. It can also be used to synchronize the time on both systems. For example the probe can act as time giver and provide its time to all connected clients. Figure 5.4 shows the sequence of a single beacon exchange.
Figure 5.4: Beacon running between probe and client - The probe sends a beacon with a given frequency and waits for an answer. When the client receives the beacon it sends an answer in return.

5.2 Probe - The Server Side

The probe is attached to the software of a robot and acts like a server publishing data of the components of the robot. This data can be live observed by requesting a component thread publishing its information or logged to the file system for post-hoc-analysis. Two main features can be defined as followed:

- **Live Observation** collects data of a component of the robot and forwards the data packed into messages to connected clients
- **Logging Cycle** collects data of selected components and saves them with a timestamp on the file system of the probe

For both features several elements are needed to control the processes between probe and connected clients. The probe implements the communication interface and every element of the probe observes the implementation. The elements on the probe are using several lists to monitor which client is connected to which element. Also some configurations can be applied by using a configuration file placed on the file system the probe. There are four control elements designed for the probe:
**Main Dispatcher**

controls all client connections and manages a list of all connected clients. For example it adds a new client connecting to the probe.

**Component Dispatcher**

handles the communication between the component threads and the client. It starts a component thread when a client requests data, e.g. collecting laser data.

**Logging Dispatcher**

handles the creation of a logging cycle on request by a client. It adds all selected components to a logging cycle and interrupts the cycle on request.

**Component Holder**

connects the components of the robot and makes them accessible to the probe. It controls how component threads and logging cycles can receive new data from components of the robot, e.g. how to read the laser range finder.

Besides each element of the probe holds a unique TAG. This TAG allows to identify the element and allows direct addressing between elements.

**Main Dispatcher**

The main dispatcher controls the connection between the probe and the client. To monitor all clients connected it holds a list of all connected clients. The list is created and available the whole runtime of the probe. When a client sends a connect command the main controller adds the client to this list. An acknowledgment is send in return to the client. To monitor the connection to each client in the list, the main dispatcher is responsible to exchange a beacon with each client. To achieve this, the list of connected clients also holds the count of beacons lost between client and probe. If the count exceeds a maximum, the connection is marked as disconnected. The maximum count for beacons missing can be set inside a configuration file. When a client sends a disconnect command the main dispatcher removes the client from the list of known clients.

**Component Holder**

The component holder is designed to capsule the communication between components of the robot and the probe. The idea behind the component holder is to create a single control instance that offers access to the robot components.
Figure 5.5: Reading and processing client commands - When a new command is received the type of the command is checked. If a client sends the connect command, the Main Controller adds the client to the list of connected clients. Contains the message a disconnect command, the client is removed from the list. If the command is not known, an error is returned to the client.

Figure 5.6: Reading a component with the Component Holder - The Component Holder establishes the connection to the robots components and reads their data on request. If a component is readable the data is forwarded to the requesting entity. If an error occurs while reading the Component Holder forwards the exception object.

Whenever a new component is available or the access modality to a component changes, all changes needed must only be applied to the component holder. The component holder facilitates the component threads and logging cycles to read the data of the components of the robot. If a component is not readable, the component holder controls the exception handling and forwards a given exception to the requesting element. For the implementation of the system done in this thesis, the component holder interacts with the BonSAI interface (see chapter 4.2) and uses a special configuration file to connect the sensors and actuators available to the probe.

Component Dispatcher

The component dispatcher controls the requests of all clients related to specific component threads. The idea behind the component dispatcher is to collect all requests for components in one instance. Only if the dispatcher establishes a connection between client and component thread data will be exchanged.
Also the component dispatcher knows all component threads available and can forward the state of each component thread to the client. This allows the client to monitor the state of components from the robot without directly connecting to a component thread. To receive requests the component dispatcher observes the communication interface for commands (Figure 5.7). Main functionalities are connecting a client with a component thread, disconnecting clients from component threads and sending the list of all component threads available. The component dispatcher knows the list of connected clients. Whenever a client disconnects, the component dispatcher removes the client from all component threads the client was connected to. The component dispatcher adds clients requesting data to the list of the requested component thread. If the thread is inactive the component dispatcher starts the component thread. The design of the probe allows to configure which component threads are available by changing a configuration file.

**Figure 5.7: Processing commands for Component Threads** - The Component Dispatcher checks the incoming commands. If a client requests a Component Thread it is added to the list of clients known to the Component Thread. A disconnect command send to the Component Thread removes the client from the list. The list of all available Component Threads can be requested from the Component Dispatcher.

**Component Threads - Live Observation of Robot Components**

To fulfill the requirement of multi user connection and collecting data in parallel, the elements used for live observation use threading. Threading allows component threads to collect data in parallel and makes multi user observation possible due to handling multiple requests at once. A component thread is reading the data via the component holder and filters it to the needs of the developer (Figure 5.8). In this thesis filtering means to extract parts of the
data and only forward the extracted data. The result is forwarded as message to the client via the communication interface. Each component thread is holding a list of all clients that have currently requested the component thread. If no client is connected, the component thread is inactivated. The component thread needs to listen to the communication interface for direct commands from clients. To achieve this, every component thread is observing the communication interface and checks every message for its TAG. Message containing the TAG are processed by the component thread. Each component thread can acquire a special state. Possible states are SLEEPING, RUNNING or ERROR. Inactive threads in SLEEPING state have no clients connected and do not collect data from the component holder. Active threads in RUNNING state read data and forward it to connected clients. Whenever an error occurs the state is changed to ERROR. Examples can be a robot component not reachable due to system failure. To control the update rate of each component thread the delay between two reading cycles can be configured in the configuration file.

![Diagram](image)

**Figure 5.8:** Component Thread collecting and publishing data - After creating and initializing the list of connected clients is checked. If no client is connected the Component Thread becomes inactive until the Component Dispatcher activates it. If clients are connected the Component Thread reads the components of the robot via the Component Holder and forwards the data to the connected clients.

**Logging Dispatcher**

The logging dispatcher controls logging cycles requested by clients. To monitor all created cycles the logging dispatcher holds a local list containing clients and their logging cycle. The logging dispatcher observes the communication interface for new requests and holds a reference to the list of connected clients.
When a client requests a new logging cycle, the client is added to the local list and a new logging cycle is created. Components selected by the client are added to the list of the logging cycle. When the cycle is successfully created it gets started by the logging dispatcher. If the client sends a stop command for a logging cycle, the logging dispatcher checks the list and interrupts the corresponding cycle. If a client got disconnected while an logging cycle is active, the logging dispatcher also interrupts the corresponding cycle (Figure 5.9).

![Diagram](image)

(a) Execute a logging cycle

![Diagram](image)

(b) Interrupting a logging cycle

**Figure 5.9:** *Logging Dispatcher* starting and stopping *Logging Cycles*
- Diagram a shows the process of creating a new *Logging Cycle* by adding selected components and starting it. Diagram b describes how an active *Logging Cycle* is interrupted via command of the client or when the client is disconnected.

**Logging Cycle - Writing Logging Output on Demand**

Logging cycles are designed to create logging files from the components of the robot. A logging cycle is a thread collecting data of multiple components with a frequency configured in the configuration file. Each logging cycle is created as thread which allows that multiple logging cycles are executed in parallel. The client can request available components to be logged from the probe.

![Diagram](image)

**Figure 5.10:** *A Logging Cycle* creating corresponding files - Each reading cycle opens a new element with the current system timestamp. The data of each selected component is read via the *Component Holder*. This process runs in a loop until it is interrupted by the *Logging Dispatcher*. If the *Logging Cycle* gets interrupted, the created document is written to a file on the probes file system.
For example, a developer can select laser data and the data of a person tracker. The logging cycle creates a single document for the runtime of the whole logging cycle. Every cycle a new logging element is added to the document containing the timestamp of the probe. The data of selected components is read via the component holder and written to the current logging element. During each logging cycle the document is written to a temporary file. This allows to backup data if the connection to the client is lost while a logging cycle is executed. If a logging cycle is stopped by the client or due to connection loss, the temporary file is written to a result file with a unique name for each cycle (Figure 5.10).

5.3 Client - Observation Window to the Robot

The client is designed to present the data published by the probe. The main idea of the client is to present one component at a time using the concept of views. A view displays and structures the data received in a predefined layout. Each view is connected to a component thread of the probe. Every element of the client observes the communication interface to receive data.

Figure 5.11 shows the initiation of the application. The first view presented will allow to set up some parameters for the communication. When the login process is executed the client will try to establish a connection to the probe. If an error occurs, the client displays the error and return to the login view. If the connection is successfully established the client starts the beacon in background and switches to the next view.

![Flowchart](image)

**Figure 5.11:** Starting the Client of observation - At startup the login view is displayed offering configuration options. If a login is executed the client establishes a connection to the probe. If the login is successful the view is switched to the status view and the beacon starts in background. If the connection could not be established an error message is displayed.
To keep the connection to the probe the beacon (see chapter 5.1) is started as a background service (Figure 5.12). This service is started once a successful connection is established and sends the beacon to the probe with a frequency defined in the configuration file. Like the probe the client counts the beacons that could not be received during runtime. The frequency for sending the beacon is delayed to minimize the resources needed. Whenever the count for missing beacons defined in the configuration file is exceeded, the state of the communication will be set to disconnected and the application returns to the login view. A received beacon resets the counter to zero.

Due to the configuration of component threads available, the views displayed on the client are also configured by using a configuration file. To present the data even on a mobile device, the navigation style of choice is a tabbed navigation system (Burrell and Sodan, 2006) presenting each view inside a single tab. Using a tabbed navigation facilitates switching between all offered views (Figure 5.13). The developer can decide which view he wants to see and views not needed are hidden until requested. Also the component dispatcher of the probe only sends a list of component threads accessible by the client. The clients needs to check this list for matching views which gets loaded and displayed inside the tabbed navigation. For each component thread on the probe a corresponding view on the client is designed.

When the client has successfully connected the initial tab view presents information about the connection and the components offered by the probe. The state of the beacon is also displayed. This view will always be created and

![Diagram](image.png)
Figure 5.13: Principle of tabbed navigation - On the top several tabs can be selected changing the content to be displayed.

Figure 5.14: Disconnect last component and request new component - When the view is changed, the client sends a disconnect command to the currently requested component thread. To request the component thread for the new view a connect command is sent.
6 Implementation

This chapter gives an introduction to the implementation of the probe and the client. The first part describes the implementation of the communication interface used on all systems developed in the scope of this thesis. The second part introduces the implementation of the probe. The last part is split into two sub-parts, showing the implementation on the mobile device and on the common computer (e.g. laptop or desktop PC). Figure 6.1 gives an overview of the implemented systems.

![Diagram of system components](image)

**Figure 6.1:** Overview of the system - In this thesis the probe and the client were implemented. The communication between components of the robot with elements of the probe is realized by using the BonSAI interface.

### Programming Language

The programming language of choice is Java\(^1\) programming language. Java allows object oriented programming, supports threading and is platform independent. By integrating threading the designed component threads can run in parallel and platform independence allows to run the client on different systems. The different components of the ToBI platform can be accessed by integrating the BonSAI interface (see chapter 4.2) which is written in Java. Another benefit is a Java implementation of the library used for the XMPP communication.

\(^{1}\)http://www.webcitation.org/5wuOWGp5I - Java Technology Network by Oracle
6.1 The Communication Interface

The communication interface is essential for the transfer of information between the probe and the client. All systems must receive and send data the same way to assure an identically understanding and processing of data exchanged. This means each application must provide the same functionalities for the communication with the corresponding systems. Figure 6.2 shows how the interface is realized and generalized by a concrete implementation. The CommunicationInterface defines several methods to be implemented by a concrete communicator used for the communication with the communication server. The Communicator class is a base class that implements the CommunicationInterface and extends the Observable class. The base class holds essential elements for the communication. The attribute sender keeps the name of the sender of an incoming message. The recipientTag is extracted from a received message and stored to make a reference for the component the message is send to. The receivedProperties map holds all properties extracted from an incoming message. Extending the Observable class allows elements to observe the communicator for incoming messages. On receiving a new message the concrete implementation must trigger a notify-process to allow observers to access the message.

![Diagram of the communication interface with example implementation](image)

**Figure 6.2:** The communication interface with example implementation - The CommunicationInterface defines the needed methods for the communication with the communication server. The Communicator is a base class with several parameters needed. XMPPCommunicator is a concrete implementation of the interface using XMPP functionalities.

The XMPPCommunicator is a concrete implementation of the Communicator class. To use XMPP for the communication, probe and client integrate the SMACK library (see chapter 4.4). The SMACK API offers methods for connecting to an XMPP server, login with given credentials and sending messages.
to other entities. A special `XMPPException` object is thrown whenever an error occurs on using XMPP functionalities.

Details for the communication with the communication server are defined in the XML configuration file (see Listing 6.1). A server can be addressed by a IP address and a port. For the login on the communication server, the method `login()` needs the name of the account and the corresponding password. If the given parameters match the data on the XMPP server, the connection is successfully established. Otherwise an `XMPPException` will be thrown.

The probe and the client are using a predefined standard configuration. The configuration of the client can be changed by setting the parameters on the start view. The configuration of the probe can be set directly in the configuration file.

The data exchanged between the probe and the client is filled into `String`-pairs containing a key and a value. To append several pairs to a message each pair is first added to a map. This map and the recipient are forwarded to the `sendMessage()` method. XMPP is capable of sending `String`-pairs by adding properties to the message stanza. When sending a new message the property map is parsed and all value pairs are added to the message. The communication interface also offers the possibility to attach a body to each message. This body can be used to add some additional textual information to a message (e.g. logging information on how the data was collected). Once all data is appended the message gets forwarded to the recipient.

Additionally the implementation of the probe can forward exception objects caught by the elements of the probe by using the `sendError()` method. The implementation of the `XMPPCommunicator` reads the errorhandler-object and adds the stacktrace of the error handler to a new message. Error messages are forwarded to all clients connected to the element catching the error.
6.2 The Probe - Implementation of the Server Side

Due to the use of the BonSAI interface (see chapter 4.2) the implementation of the probe is called ReSAI (Remote Sensor Actuator Interface). Figure 6.3 shows the main classes implemented for the probe. For a better overview all lists controlled by the different elements are left out. Also helper classes (e.g. TimerConverter) are not shown.

![Diagram of main classes of the probe - The ReSAIServer class creates and initializes all needed dispatchers. Each dispatcher controls several sub components.](image)

**Figure 6.3:** Main classes of the probe - The ReSAIServer class creates and initializes all needed dispatchers. Each dispatcher controls several sub components.

ReSAIServer - Initializing the Probe

The ReSAIServer class initializes at startup all needed classes for the probe. The Clients class extends the Observable class and holds a hash map of all clients connected and their current beacon count. This list is used to monitor all connected clients which connect during the runtime of the probe. For the configuration of the probe the XML configuration file is parsed and stored in the ServerConfig. The ServerConfig holds all details for the communication server, the list of available component threads to be initialized and the
elements available for logging cycles.

After parsing the configuration the **XMPPCommunicator** is created. This establishes the communication with the details of the **ServerConfig**. On successful connect to the communication server, all component threads are created and started by using the list of component threads from the **ServerConfig** class. Each component thread gets a reference to the communicator, a reference to the component holder and is added to the observers of the list of all connected clients. Initialized component threads are stored inside a map for further processing. When all component threads are initialized, the **MainDispatcher**, **ComponentDispatcher** and the **LogDispatcher** are created. By using the different dispatchers the communication flow between the corresponding sub elements and the client can be controlled. Elements can only be requested by calling these dispatcher classes. Each dispatcher gets a reference to the list of connected clients and observes the communicator. For sending messages a reference to the communicator is passed. The **LogDispatcher** and the **ComponentDispatcher** need some additional references. The **LogDispatcher** gets a reference to the component holder and the server configuration. The reference to the **ComponentHolder** is forwarded on request to each created **LoggingCycle**. The **ServerConfig** is used to create a list of available logging components. The **ComponentDispatcher** gets the map of all initialized components for controlling the communication between clients and component threads.

**MainDispatcher**

The **MainDispatcher** class handles all the communication with the clients. To receive commands and send messages the **MainDispatcher** holds a reference to the **Communicator**. The **MainDispatcher** observes the communication interface for new commands send by clients. All possible commands are defined as constants in the **Constants** class and represented as enumerations. The **MainDispatcher** controls the list of all connected clients by using the **Clients** class referenced on creation. Whenever a client sends a connect command, the **MainController** adds the new client to this list or removes the client on receiving a disconnect command. To monitor the connection to a connected clients, the **MainDispatcher** is running a loop process checking the list of connected clients and exchanging a beacon with each client (see chapter 5.1). Inside the loop the **MainDispatcher** checks the beacon count for every connected client. If the beacon count is below a count for maximum missing beacons, a new beacon message is created and forwarded to that client. If the count equals a maximum count the client is removed from the list. Removing a client from the list triggers a notify on all listeners. This principle allows to
stop all active processes of the corresponding client. The frequency of the loop and the count of maximum missing beacons are defined in the XML configuration file. To receive incoming beacons the MainDispatcher also observes the Communicator. When a new beacon is received the beacon count for the corresponding client reseted to zero.

**ComponentDispatcher**

The ComponentDispatcher class controls the communication between component threads and clients and observes the Communicator for new requests from clients. The available commands are stored inside a Constants class. To control the different component threads the ComponentDispatcher holds a list of all available component threads and their current state. On request the list is forwarded to the to clients. When a client requests the data of a component thread, the ComponentDispatcher checks the current state of the component thread. If the state is **RUNNING**, the client is added to the client list of the component thread. If the state is **SLEEPING**, the client is added to the client list and a notify() event is executed to activate the component thread. Whenever a client requests a disconnect from a component thread, the dispatcher removes the client from the list of the corresponding component threads.

**ComponentThread - Live Observation**

For the implementation of a component thread a ComponentThreadInterface and a ComponentThread base class where created. The idea of the base class is to capsule the methods for receiving and sending messages and only offer simpler functionality for the extending classes. For receiving messages the base class observes the Communicator. New messages are handled inside the update() method by reading messages from the Communicator. The elements of the messages are parsed and stored in the receivedProperties map for further processing. Due to the need of executing multiple component threads in parallel the ComponentThread extends the Thread class.

The base class holds several lists. The connectedClients list contains all clients currently connected to this component thread and is used to forward data to those clients. Whenever the list is empty the component thread is gets deactivated. The globalClients list is a reference to the list of all clients connected to the probe. Observation of this list allows to remove clients from the connectedClients list whenever a client disconnects from the probe. The receivedProperties list holds all properties extracted from a new message.
that has been received from a client. All properties to be send to connected
clients are stored in the \texttt{propertiesToSend} list. For creating new messages
the base class offers several functionality. New value pairs can be added to the
\texttt{propertiesToSend} list by using the \texttt{addProperty()} method. By calling the
\texttt{sendMessage()} message the base class iterates over the \texttt{propertiesToSend}
list and adds each value pair to a message. When the message is successfully
created the base class sends the message to each client which is part of the
\texttt{connectedClients} list. After sending the message the \texttt{propertiesToSend}
list gets cleared.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{component_thread.png}
\caption{Base class and interface for a component thread -
The ComponentThread class is abstract and implements the ComponentThreadInterface. The base
class handles all needed communication via a reference
to the Communicator. LaserDataThread is a con-
crete class extending the ComponentThread class. The
methods \texttt{prepare()}, \texttt{processNewProperties()} and
\texttt{processing()} are used by the concrete implementa-
tion.}
\end{figure}

A concrete implementation has to extend the \texttt{ComponentThread} base class
and must implement the methods defined by the interface. An example of an
implementation is shown on Figure 6.4. The \texttt{prepare()} method is called once
the component thread is initialized. It can be used to initialize parameters on
the concrete class (e.g. configuring the laser range finder). The \texttt{processing()} method is called on every cycle of the component thread. To collect data this
method can access the \texttt{ComponentHolder} and read data of components of the
robot. Collected data can be added as new value to the \texttt{propertiesToSend}
map by calling \texttt{addProperty()}. At the end of the \texttt{processing()} method
the base class automatically calls the \texttt{sendMessage()}. At the end of the
\texttt{processing} method a delay is placed which allows to define a frequency for the
process of collecting data. This delay can be changed in the XML configuration
file. The method \texttt{processNewProperties()} is called whenever a new message
is received containing the TAG of the component thread. Passed to this method is the `receivedProperties` map containing all extracted value pairs of the message. The list can be used for further processing of client commands inside the component thread (e.g. setting a navigation goal for the robot).

Each component thread holds an attribute showing its current state (see chapter 5.2). When a component thread is created it is first put into the `SLEEP` state due to no connected clients. Inactive component threads are not actively reading data and can only be activated by the `ComponentDispatcher`. When a new client connects and the component thread is activated by the `ComponentDispatcher` the state is changed to `RUNNING`. While active collected data is forwarded to clients in the `connectedClients` list. Whenever the component thread receives an error from the component holder or an error occurs inside the defined methods, the state of the component thread changes to `ERROR` and the component thread gets deactivated. Component threads in error state are not collecting data and cannot be activated. In the current implementation component threads in `ERROR` state can only be activated by restarting the probe.

**LogDispatcher**

The `LogDispatcher` class controls the creation of logging cycles. It uses the `Constants` class with enumerations for commands to handle clients requests. The `LogDispatcher` holds a reference to the `globalClients` list and the `loggingList` containing clients currently running logging cycles. To send messages a reference to the `Communicator` is stored and for receiving messages the `LogDispatcher` observes the `Communicator`. The list of available logging components is parsed from the `ServerConfig`. On request by a client the `LogDispatcher` offers a list of available logging components. When a client requests a new logging cycle, the `LogDispatcher` parses the incoming message and extracts all requested components to be logged. Once parsed the `LogDispatcher` creates a new `LogThread`, passes all requested logging components to this thread and executes it. When the client requests to stop the logging cycle the `LogDispatcher` checks the `loggingList` for a corresponding `LogThread` and interrupts it. In case of a notify from the list of all clients concerning a clients disconnect the `loggingList` gets parsed and logging cycles for the client get interrupted.
LogThread - the Logging Cycle

A LogThread represents a single logging cycle for one client. A LogThread extends the Thread class to allow the execution of multiple logging cycles in parallel. Components to be logged are added as LogClass elements to a map by using the LogClassInterface. On initialization the LogThread creates a new XML document. Every cycle a new element is created and attached to the XML document containing the current timestamp of the probe. This element is called TimestampElement and used as container for the current data of the logged components. The LogThread iterates over the map of LogClass elements and reads the data of each LogClass by calling the readData() method. The result of the method is added as child to the current TimestampElement.

For backup reason a LogThread saves the current XML document to a temporary file at a given frequency defined in the configuration file. The temporary file is unique for each cycle and is used to backup the logged data in case of an error occurring on runtime. When the LogThread is interrupt by the LogDispatcher, the XML document is saved to a final file on the file system of the probe. Each file gets a unique name containing the creation timestamp and is saved in a separate folder unique for each client.

LogClassInterface - How to Read the Data

The LogClassInterface is created for concrete implementations of logging classes. It is used to read and collect data from the ComponentHolder. The collected data is placed inside a new XML element and can be accessed by the readData() method. The idea behind single logging classes is to support extension of logging cycles. To create new logging classes the LogClassInterface must be implemented and the class itself added to the XML configuration file. Once added the component is automatically offered by the LogDispatcher on request by the client.

6.3 Implemented Components of the Robot

Due to the limited time for implementing the system only seven components of the used robot platform have been implemented as component thread. Implemented sensors are the laser sensor, SLAM sensor, person sensor, speech sensor and a camera sensor for an USB camera. Implemented actuators are controlling the navigation and the speech component of the robot.
The selected sensors give a good impression the data collected on different tasks the robot can perform. Laser and SLAM sensors are used for navigation, person and speech sensor represent information needed for human-machineinteraction. The USB camera was used for testing the capability of sending bigger payloads by using higher frequencies. Laser and SLAM sensors provide their data as arrays of numbers and some additionally information about the robot position. To filter this data the different arrays have been translated to a visual presentation as image. The additional data, like the position of the robot, is added to each images. XMPP does not support binary attachment. In order to still send the image, each image is decoded to a string by using base64\(^2\) and then attached to a new message. On the client side the string is encoded with base64 back to an image. Base64 allows to translate a byte array to a string and vice versa. The same principle has been used for sending images captured by the USB camera. The data of the person sensor and the speech sensor is provided as string containing information about spoken words or angle of leg pairs from persons standing in front of the robot. This data is added as normal String-pair to a message.

Each implemented component thread controlling an actuator can receive specific commands which are forwarded to the robot via the component holder. The component thread for the navigation actuator can receive X/Y coordinates for navigation or direct commands like “manual_stop”. The component thread communicating with the speech actuator parses the sentence to be spoken and calls the corresponding function on the speech actuator by using the component holder.

### 6.4 Client

The design for the client described in chapter 5.3 has been implemented on the android operation system and on a common computer. To display the data of the component threads a tabbed navigation system is implemented in both clients presenting data inside tab views. Although the views presented can be similar implemented, the system in the background for controlling the views is different for the clients. On mobile devices the resources are very limited compared to the common computer. At the time of this thesis a mobile device is not capable of using multi-cores for threading. On the common computer more resources are available which allows to use more resources and display more views at once. For example two views presenting the laser data and the navigation planing can be combined inside one view. The following section gives an introduction of the implementation on both systems.

\(^2\)http://www.webcitation.org/5x3gUQW3U - Base64 (RFC 4648)
6.4.1 Mobile Devices - Playing with Android

In this thesis the operation system of choice is the android operation system (see chapter 4.3). Due to limited resources mentioned and the smaller displays the style of data presentation on a mobile device is not that easy. At the time of this thesis, a mobile device is not capable of running a lot of threads at once with a high frequency. The presentation of data received via the network needs to be implemented in a lightweight way. The use of high update rates or expensive loops must be prevented.

Figure 6.5: Class diagram of the mobile client with important elements - The entry point of the mobile client is the ReSAIDroid class creating all needed elements. Tab activities itself are created and controlled by the TabManager.

The client application on the mobile device is called ReSAIDroid (Remote Sensor Actuator Interface on Android). For the presentation of the data several views where implemented. A view on the android platform is called activity (see chapter 4.3). Each view is used to present a corresponding component thread on the probe. There are three main elements on the android client. The first element is the starting screen that is used to configure several communication parameters and executing a login process. The second element is a manager controlling the tabbed navigation and the third element is a selected tab presenting a view. The mobile client uses a Constants class for commands similar constants of the probe. Figure 6.5 shows the main classes implemented with an example of two tab classes created in the thesis. For
mapping the component threads of the probe to the views of the client a special `TabComponentMap` class was created containing several configuration lists. The `classes` list contains the connection between a view on the client and a component thread or dispatcher on the probe. This list is used to forward an incoming message to the intended class by checking the TAG the received message contains. The name of a tab is connected to the corresponding class via the `tabName` list. The `tabPosition` list is used to define the position of a view inside the tab navigation bar. These lists can be defined in the configuration file placed on the file system of the client. Listing 6.2 shows an example of the view configuration inside the configuration file.

**ReSAIDroid - Starting the Application**

The entry point of the client on the mobile device is the `ReSAIDroid` class presenting the initial view. This view allows to configure parameters needed for the communication by providing corresponding input (Figure 6.6). At the bottom of the view a status field displays information about connection state and errors that occurred. The `ReSAIDroid` class creates an instance of the `XmppCommunicator` which establishes the connection to the probe. When executing the login process, the connection to the communication server is initialized and the client executes the login process with the given parameters. If the connection is established the client sends a connect command to the probe. A timer monitors the communicator for answers of the probe. When the probe acknowledges the connection the beacon is started in background and the current view switches to the `TabManager`. If the probe is not responding the connection attempt is canceled and an error message is displayed in the status field. Once the connection is successfully established, the given parameters from the different inputs are saved to a local file for further connections.
Beacon - Keep Connected to the Probe

The Beacon class on the mobile client is implemented as a thread. This allows to run the beacon in background and in parallel to other activities. The beacon is checked with a lower frequency defined in the config file to reduce the impact on the system load. The beacon listens to the communication interface for incoming beacons sent from the probe. The timestamp of the probe is forwarded to the TabConfig view. The Beacon class monitors the count of missing beacons and resets the counter to zero whenever a new beacon arrives. If the counter for maximum missing beacons is exceeded the Beacon thread stops the current activity and switches to the login view. The count of maximum missing beacons can be configured in the configuration file.

TabManager - Control via Tabbed Navigation

When a client is successfully connected an instance of the TabManager class is created. The TabManager class extends the TabActivity class and acts as a content holder for the different views. Each view implemented as a tab is named with a “Tab” prefix to distinguish a normal view from a tab view. Extending the TabActivity class allows control of the interaction whenever a view is switched. To display the tabs on mobile devices with a lower resolution, the tab navigation can be scrolled horizontally.
TabConfig - The Status View

The TabConfig is the first view created and added to the TabManager. It displays information about the connection to the probe, the data from the last beacon received and the list of the component threads available with their current state. This class is responsible for creating and initializing all tabs to be displayed. For the configuration the TabComponentMap class is referenced. To create tabs the TabConfig requests the list of available components from the probe. When the list is received it is compared to the classes list of the TabComponentMap. Views matching the list received from the probe are initialized and added as tab views to the TabManager.

![Status screen](image-a) ![SLAM component](image-b) ![laser component](image-c)

**Figure 6.7:** Tabbed navigation after login - At the top of each view the available tabs are displayed. Image a shows the status view with information about available components on the probe. Image b shows the data from the SLAM component and image c the data from the laser component.

Component View - A Template for Tab Views

To present a component thread from the probe a corresponding component view is defined on the client. The TabTemplate base class created capsules all the overhead for communication with the probe. The idea of the base class is to offer simple functionality for creating and extending component views. Also it provides an implementation skeleton for a concrete implementation. When the onResume() method is called the base class requests the corresponding component thread. Executing the onPause() method triggers a disconnect from the corresponding component thread. The TabTemplate implements the TabTemplateInterface which forces a concrete component view to implement the processProperties() method for further processing.
1 <LinearLayout
2 xmlns:android="http://schemas.android.com/apk/res/android">
3  <TableLayout
4      xmlns:android="http://schemas.android.com/apk/res/android">
5      <TableRow android:gravity="center">
6          <ImageView android:id="@+id/ivLaser" />
7      </TableRow>
8  </TableLayout>
9 </LinearLayout>

Listing 6.3: Example layout for the laser data view - Each layout uses XML elements with attributes to create the view.

The properties of each incoming message are extracted and forwarded to the `processProperties()` method. Figure 6.8 shows an example of the laser sensor tab extending the `TabTemplate`.

![Image of UML diagram](image)

**Figure 6.8**: Example class extending the `TabTemplate` - The concrete tab view must implement the `processProperties()` method to process data extracted from incoming messages.

Each view must implement its own layout defined in an XML file (Figure 6.3) and extend the `TabTemplate`. A predefined layout can be loaded inside the `onCreate()` method. Also this method can be used to set initial parameters. To send messages to the corresponding component thread on the probe, the base class offers two methods. The `addProperty()` method adds a new value pair to the `propertiesToSend` map stored in the base class. The `sendMessage()` method forwards the message to the corresponding component thread.
Each view adds a listener to the communicator whenever the `onResume()` method is called. To save system resources, this listener is removed from the communicator whenever a view is deactivated by calling the `onPause()` method. The idea behind this principle is to reduce system load by reading messages only on notify be the communicator.

**TabLogDispatcher - Initiating new Logging Cycles**

The `TabLogDispatcher` view communicates with the `LogDispatcher` and requests the list of available logging components. To allow the configuration of components to be logged on the probe available logging component are presented asSelectable list. By executing a logging cycle the client adds properties for each selected component to a message forwarded to the `LogDispatcher`. While the logging process is active, all elements on the view are deactivated except the stop button. Executing the stop process sends a stop command to the `LogDispatcher`. When the current tab view changes the `TabLogDispatcher` does not stop the logging cycle on the probe. This allows to use the live observation even if a logging process is active. Figure 6.9 shows the logging view.

![Logging View](image)

**Figure 6.9:** Starting logging cycles - The `TabLogDispatcher` view controls the creation of logging cycles on the probe. If the active tab is changed while a logging cycle is active, the logging cycle on the probe is not stopped.

### 6.4.2 Java Client on Desktop Machine

For using the client on a common computer a prototype of the client was implemented. Due to more resources the application shows more information combined on one screen. Like the mobile client a tabbed navigation is used to
present the data and for the control of the GUI the Model/View/Controller pattern (see Freeman et al. (2004), pages 529-546) is integrated. The main elements are the login view and the overview panel. The overview contains panels for information about the connection, a list of all available component threads and their states and a panel containing the tabbed navigation. For controlling the interaction all windows are using controller classes handling the communication in background. Figure 6.10 gives an overview of the main classes implemented. For the communication with the probe the communication interface is integrated using a concrete implementation of the XMPP communicator. Each element of the Java client can observe the communication interface and receives incoming messages on notify triggered by the communicator. The ComponentMapper is used for matching component threads of the probe with corresponding views on the client.

![Diagram of main classes of the Java client](image)

**Figure 6.10:** Main classes of the Java client - On successful connect all needed panels are created. The TabHolder creates and controls all panels for presenting the component threads and the logging cycle.

**LoginController and Login Window**

The LoginController and the Login view handle the initialization of the connection to the probe. The Login class creates the GUI with input for the communication parameters. Events of the Login view are observed by the
LoginController which also observes the communication interface for incoming messages. By executing the login process, the LoginController connects to the communication server and performs the login with the provided parameters. When the connection is successfully established, the LoginController sends a connect command to the probe. While waiting for the acknowledgment of the probe a timer checks a flag presenting the state of the connection to the probe. If the connection to the probe is successfully established the flag is set to be connected and the view gets switched to the Overview. When the acknowledgment is not returned within a defined period, an error message is displayed. Figure 6.11 shows the login view.

![Login for the common computer](image1) ![Login executed](image2) ![Message if probe is not reachable](image3)

**Figure 6.11:** Starting the Java client - Image a shows the login window with the parameters for the communication interface. After executing the login process the GUI is inactivated shown in image b. Image c shows the message displayed when the probe is not reachable.

When the connection is successfully established the LoginController creates an instance of the Beacon class. This class uses a timer to send beacons to the probe with a given frequency. The Beacon listens to the communication interface for incoming beacons, just like the probe it counts the missing beacons and stops the client on connection loss. If new beacons are received the beacon counter is reseted to zero.

**OverviewController and the Overview Window**

The main part of the Java client is the overview window which presents information from the probe using different panels. The Overview class creates the TabHolder and the helper panels. Events on the overview window are controlled by the OverviewController. The OverviewController observes the
communication interface and requests the list of available component threads on initialization. When the list is received, the **ComponentMap** is checked and corresponding panel are created. Each matching panel is added to the **TabHolder** and registered to the observers of the communication interface. To manage changing the active tab view, the **TabHolder** is observed by implementing the **ChangeListener** interface. When the tab changes a disconnect command is send to the currently observed component thread. To connect the selected tab to the probe a connect command is send to the corresponding component thread. Figure 6.12 shows the **Overview** view with its panels.

![The overview window with tabbed navigation](image)

**Figure 6.12:** The overview window with tabbed navigation - The upper panel shows all available component threads, the status tab and the tab for logging cycle’s. The lower left panel shows the state of the component threads. Beneath the status panel is the data from the last beacon presented. The lower right panel is used to display error messages received.

**TabHolder, PanelTemplate and the Component Tabs**

For the presentation of the data from the different component threads, a tabbed system is implemented. To use tabs the **TabHolder** extends the Java JTabbedPane. For each panel to be added to the **TabHolder** the **PanelTemplate** base class is created. This template implements the **Observer** interface and ex-
tends the JPanel class. The template is used to define a skeleton for concrete component panels. Each component tab must extend the PanelTemplate and is forced to implement the update() method for observation. Each panel is added to the observers of the communication interface. To present the data each panel can create its own layout.

LogPanel

The LogPanel is implemented for the creation of logging cycles on the probe. This panel requests the list of available logging components from the probe. The received list is presented by check-boxes for each logging component. When the logging cycle is executed, the LogPanel sends the list of selected logging components to the probe and inactivates all elements of the view except the stop button. When the logging cycle is requested to stop, the stop command is send to the probe and the elements of the LogPanel are activated again. Like the mobile client changing the current tab while a logging cycle is active does not result in stopping the logging cycle.

The Helper Panels

To present the state of the probe, the Java client offers several helper panels. Each panel observes the communication interface for corresponding information. Implemented panels are the BeaconPanel, ErrorPanel, StatusPanel and the ComponentPanel.

BeaconPanel

The BeaconPanel presents the data of the last beacon received from the server. It extracts the properties of the received beacon and fills them to corresponding labels.

ComponentPanel

The ComponentPanel uses a timer to request the state of the component threads with a given frequency. When the list is received it is parsed and printed to a label.

ErrorPanel

To show errors received from the probe the ErrorPanel listens to the communication interface for incoming messages containing errors. Once an error message is received it is parsed and displayed.
**StatusPanel**

The idea behind the **StatusPanel** is to present all received messages from the probe. To achieve this, all received messages from the communication interface are printed to a text box. Each message is printed with additional information of the time of arrival, the TAG attached and the name of the sender.
7 Evaluation

In the previous chapters a system was described for system introspection, observing and analyzing log files. The system was successfully attached to the software of the ToBI platform and allows a deeper insight of the internal process of different components at runtime. To proof the capabilities of the system and to check if the requirements concerning the influence on the robot and the quality of the data are fulfilled, two different evaluations were performed.

Requirement two (see chapter 3) was to exchange data with a minimal delay between the time of sending and the time of receiving. The first part of the evaluation deals with the latency occurring between sending data from the probe and receiving it on the client side. The goal was to measure if different frequencies of sending the data will influence the latency or even result in a connection loss. Also the influence of different payloads is checked (e.g. a component presenting its data by an array of 5000 elements).

The second part of the evaluation was performed to check requirement 8: Minimum influence on the robot (see chapter 3). The evaluation measures the additional load on the system when the probe is attached and active. The goal is to check if observing the different components result in influence on the normal performance of the robot (e.g. robot is interrupted while planning to navigate). For this evaluation the robot runs a predefined course with and without the probe attached. Each run measures the load on the CPU of the main laptops controlling the robot.

7.1 Preconditions for the Evaluation

The designed test series were executed on the ToBI platform (see chapter 4.1). The probe is installed on a third laptop which is additionally attached to the robot and connected to the network on the platform. Installed on the third laptop the Openfire-Server (see chapter 4.4) is and used as communication server. The client application is installed on a mobile phone and another separate laptop. Both clients are connected wireless to the router of the robot. To assure that all devices are using the same time, all devices synchronized
their system clocks with a network time protocol\(^1\) daemon running on the same laptop the probe is installed on.

### 7.2 Measuring the Different Latencies

To make a decision for next steps based on the data exchanged between the probe and the client the time between sending and receiving data must be reduced to a minimum. The time delay between sending and receiving is called latency.

#### 7.2.1 Goal

Sending data over a network implies several components in the chain of forwarding data (e.g. sending a message over a communication server to a recipient) Each component consumes time for processing and forwarding the data which makes it obvious that a latency exists for exchanging data between the probe and the client. Due to the fact that the components of the robot offer different information values with different frequencies, its important to reduce the latency. The goal of the first evaluation was to measure this latency regarding different payloads and frequencies when exchanging data. Is the latency higher if a bigger payload is exchanged? Does a higher frequency increase the latency or could it be constant for smaller payloads? The following method will try to give an answer on those questions.

#### 7.2.2 Preparations

For the test an evaluation component thread was created on the probe. This component thread was designed to send messages by configuring several parameters concerning the frequency and the payload. For each test one out of three patterns can be selected which gets automatically added to each message. Table 7.1 shows the patterns predefined. Also the delay between two messages can be set which is used to simulate a frequency for exchanging data. Setting this delay to zero will result in a maximum possible frequency due to sending a message in every step. For the configuration of each test a simple GUI is created and each test can be executed from within this GUI. On the client side a corresponding component view for receiving the messages is created. The component view is capable of storing data by writing data to a logging file on the clients file system.

\(^1\)http://www.webcitation.org/5xCmxgeBb - The Network Time Protocol (NTP)
A test series with a total of 18 tests was performed. For each test the probe forwards a total of 200 messages to the connected client. For the first nine tests the mobile device was connected as client to the probe. For the second nine tests the client connected was the separate laptop. For a single test the evaluation component on the probe was configured with the parameters shown in Table 7.2. Parameters were the amount of messages send to the client, the delay between two messages and the pattern attached to each message. When a single test was executed, the probe first sends a message containing the name of the logging file. The client creates the file according to the given name on its local file system. Once the component thread has send the name of the logging file the messages for the test are forwarded according the parameters configured in the GUI. Whenever a message arrives on the client side, the client stores its current timestamp to a variable. From the received message the time of the probe gets parsed and stored to a variable. Due to the fact that the received data normally is processed somehow, work on the data is simulated by iterating of the properties of each message and counting its characters. At the end of simulating work the stored variables are appended to the logging file. When the component thread reaches the configured amount of message to be send it automatically stops and the test ends.
<table>
<thead>
<tr>
<th>Test No.</th>
<th>Pattern</th>
<th>Delay (ms)</th>
<th># Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Singleline</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>Multiline</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>Complex Data</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>Singleline</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>5</td>
<td>Multiline</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>6</td>
<td>Complex Data</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>7</td>
<td>Singleline</td>
<td>1000</td>
<td>200</td>
</tr>
<tr>
<td>8</td>
<td>Multiline</td>
<td>1000</td>
<td>200</td>
</tr>
<tr>
<td>9</td>
<td>Complex Data</td>
<td>1000</td>
<td>200</td>
</tr>
</tbody>
</table>

Table 7.2: Parameters for the latency test sequences evaluated - The pattern describes the type of the payload attached to a message. The delay is used to simulate a frequency for sending messages and placed between two messages. The number of messages determines how many messages are send in one test. Each parameter can be configured via the GUI of the *evaluation component thread*.

7.2.4 Results

To analyze the collected data the average latency and the standard deviation (STDEV) of the different patterns is calculated (see table 7.3). Figure 7.1 shows the results on the mobile device, plotting the latency of sending the different pattern with a delay of 100ms. The results for the other frequencies and the results for the laptop can be found in the appendix. A first impression on these results is that for sending smaller payloads like the *Singleline* and *Multiline* pattern the latency is quiet low. For the *ComplexData* pattern the latency is with around 250-300 ms quiet higher and may be not suitable for transferring data with a high frequency. In comparison to the laptop the latency seems to be higher on a mobile device.

The *Singleline* data is intended for short message exchange with a higher rate. On the mobile device an average latency of 13.01ms (STDEV 15.42ms) could be measured for a high update rate of 100 ms. On the laptop the average latency measured was 2.67ms (STDEV 5.44ms). Exchanging small payloads can be exchanged with a low latency even with a higher frequency. The *Multiline* simulates bigger payloads containing several information (e.g. lists about component states). With an average latency of 13.74ms (STDEV 11.53ms) on the mobile device and 5.16ms (STDEV 10.17ms) on the laptop even a high frequency of 100ms can be achieved. For both patterns, *Singleline* and *Multiline*, data supports the requirement to achieve a minimum delay (see chapter 3). The *ComplexData* pattern can be compared to exchange bigger payloads (e.g. images or byte arrays). With an average latency of 291.15ms (STDEV
Figure 7.1: Latency of receiving data on the mobile device – The results of sending the different pattern with a delay of 100ms is plotted. For each pattern 200 messages were send to the client.

101.34ms) on the mobile device and 284.99 (STDEV 144.74ms) on the laptop using a frequency of 100ms such data may only be used for observation without expecting actual data. Using lower frequencies like 500ms or 1000ms seems not to reduce the latency.

However several peaks can be observed in the plots of the results. For each test the robot was running the same internal processes active even if the probe is not attached to the robot software. To communicate both controlling laptops use the network of the platform. Observing the network devices on both laptops showed a usage of over 50% network usage used for communication between these laptops. One reason for peaks occurring in the plotted data can be the load on the network bandwidth. Whenever the bandwidth is used up by the exchange of messages consumes more time. Also a comparison of the time of sending between two messages showed that the probe forwards the message with the defined delay. Each message is forwarded to the communication server. Running as own process a processes running in parallel (e.g. receiving data over the network interface) can interrupt the process of dispatching messages and result in a time delay. On the client side the process priority can influence the processing of incoming messages as well. On the mobile device each process underlies the control of the operation system which also controls the network interface. But even with the peaks the overall latency is nearly constant. No message got lost between the probe and the client. Regarding the questions announced in the goal the results indicate that the payload influences the latency. Bigger payloads produce a higher latency. For smaller payloads the delay is nearly constant even for faster message exchange. Also the results of the different clients are similar which indicates that the latency
<table>
<thead>
<tr>
<th>Test No.</th>
<th>Delay</th>
<th>Mobile Device</th>
<th>Laptop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>AVG</td>
<td>STDEV</td>
</tr>
<tr>
<td>Singleline</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>100ms</td>
<td>13.01ms</td>
<td>15.42ms</td>
</tr>
<tr>
<td>4</td>
<td>500ms</td>
<td>8.7ms</td>
<td>9.79ms</td>
</tr>
<tr>
<td>7</td>
<td>1000ms</td>
<td>14.52ms</td>
<td>15.54ms</td>
</tr>
<tr>
<td>Multiline</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>100ms</td>
<td>13.74ms</td>
<td>11.53ms</td>
</tr>
<tr>
<td>5</td>
<td>500ms</td>
<td>19.21ms</td>
<td>12.07ms</td>
</tr>
<tr>
<td>8</td>
<td>1000ms</td>
<td>25.34ms</td>
<td>13.8ms</td>
</tr>
<tr>
<td>ComplexData</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100ms</td>
<td>291.15ms</td>
<td>101.34ms</td>
</tr>
<tr>
<td>6</td>
<td>500ms</td>
<td>281.85ms</td>
<td>41.47ms</td>
</tr>
<tr>
<td>9</td>
<td>1000ms</td>
<td>323.24ms</td>
<td>50.65ms</td>
</tr>
</tbody>
</table>

Table 7.3: Results of the latency evaluation - For each device connected the average (AVG) and the standard deviation (STDEV) of the latency is displayed. The results are divided into the different delays for the pattern attached to a single message (Singleline, Multiline and ComplexData).

depends on the network and not on the processing power of the client.

### 7.3 Observing the CPU Load of the Robot System

To observe a robot and to get an idea of the inner processes, the robot itself must offer an interface to query the desired information. One idea is to make the robot publishing its information at any time. This way often results in loosing needed resources (e.g. additional bandwidth usage even if data is not consumed). The system developed in this thesis uses the first way and queries information of the robot only if a developer needs the information. But even this technique needs to trigger a process on the robot to gather the information and forward it. Due to the fact that this process uses system resources its obvious that the systems performance is influenced. One requirement was to minimize this influence when using the probe (see chapter 3).

#### 7.3.1 Goal

The following evaluation was done to check the system developed in this thesis. Does the implementation consume more resources than available on the robot? Could actively observing the robot result in interruption of the normal behavior compared to the behavior without the probe active? To check the influence the system will be observed with and without the probe active.
For this thesis only a rough estimate can be given on the collected data due to the fact that the robots behavior is never constant. The navigation control of the robot does not create the same path for every run. Also several controlling processes run in parallel which results in different executions for same starting conditions. For the evaluation the assumption is made, that every test run is nearly the same and the collected data of the different runs can be aligned. To make a statement about the results, the average of the runs with and without the probe is analyzed.

7.3.2 Preparations

The different components of the robot are controlled by two laptops placed in the backpack. The laptop controlling the cameras and the speech recognition will be referred to as upper laptop, the laptop controlling the navigation will be referred to as lower laptop. To measure the system load the ToBI platform is running a predefined navigation procedure traveling along a defined course (Figure 7.2). Through the course four points must be reached by the robot. One round through the course takes the robot 45 seconds. The starting position of the robot is marked to the ground and for each test the robot is placed on that position. Monitored are both laptops controlling the robot. Each laptop is synchronized against a NTP daemon running on the third laptop. To measure the CPU load a special tool is written. This tool reads the current CPU load in percent and appends the results to a file on the local file system. To minimize the influence of this tool it only collects the data with a frequency of 250 ms. For each test a total of 180 data points is written to the logging file.

7.3.3 Method

For this evaluation a total of 20 runs is performed. For each test the robot navigates to four predefined points. The first ten runs the probe is not active. On the next ten runs the probe is activated and one client is connected. Before every test starts the robot is placed at the starting position and the system clocks on the main laptops get synchronized. When a single test is executed, the logging tool is started synchronously on both laptops and the navigation procedure starts. After 45 seconds each logging tool automatically stops recording and writes the result file to the file system.
Figure 7.2: Predefined navigation course - The image shows the SLAM map the robot generates by moving around. The robot starts on point a and navigates to the points b, c, d and e.

7.3.4 Results

For the different runs the CPU load is analyzed. For each ten runs the measured data points are aligned and the average for corresponding points is calculated. To compare the results the average CPU load of the runs without the probe active will be referred to as nominal values, while the average values with the probe active will be referred to as the actual value. Figure 7.3 shows the average CPU load on both laptops. A first impression is that there is influence on the system load by using the probe. Calculating the average and the standard deviation of this influence (Table 7.4) supports this impression. Looking at the graph does not show a constantly higher usage. To check this all aligned data points where the actual value is higher then the nominal value are counted. The results reveal that from the 180 aligned data points 146 points (81,11%) on the upper laptop and 161 points (89,44%) on the lower laptop have a higher CPU load. On the lower laptop the average for this difference shows an additional CPU load of 5,86%. On the upper laptop difference shows an additional CPU load of 3,32%. Due to the fact that the evaluation only presents a rough estimation one statement here is that there is influence on the system load. Through the evaluation the laptops never reached a critical level where the CPU load was that high no resources could be spared for the probe. If the system reaches a high usage of over 90% the usage of the probe
<table>
<thead>
<tr>
<th>Laptop</th>
<th>CPU load without Probe</th>
<th>CPU load with Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AVG</td>
<td>STDEV</td>
</tr>
<tr>
<td>lower</td>
<td>26.62%</td>
<td>11.1%</td>
</tr>
<tr>
<td>upper</td>
<td>46.75%</td>
<td>7.79%</td>
</tr>
</tbody>
</table>

Table 7.4: CPU load with and without the probe activated - Shown is the average (AVG) and the standard deviation (STDEV) of the CPU load on the different laptops.

must be reconsidered due to the measured additional resource usage. If the resources can be spared, the evaluation showed that running an active probe attached does not influence the behavior of the robot while performing.
Figure 7.3: Average CPU load with and without the probe activated.

- On the upper (b) and the lower (a) laptop the average CPU load is higher with the probe active. The peak at the beginning of each test shown in image b is created by the initialization of the navigation process which consumes a lot of processing power.
8 Conclusions

In the context of this thesis two systems were designed and implemented. The first system is a probe attached to the software of a running robot system. This probe collects data of the components of the robot and publishes this information. The second system is the client receiving the data on request. The described design and the implementation allow to extend the system by using predefined elements. New components or changes on the existing elements can be applied on both systems.

By designing and implementing a communication interface both systems can exchange and understand data the same way. Also this interface makes the underlying communication framework replaceable. The use of XMPP as underlying communication framework offers all needed elements to support the designed communication interface. The only weakness here is the missing support for exchanging complex data structures (e.g. binary attachments). For the implementation several tests showed that even this problem could be solved in an acceptable way (e.g. compress data to Base64). However for complex data (e.g. video steams) the created system should only establishes a way for the communication and leave the communication itself to a better technique.

The probe offers two different ways to look at the inner processes of the robot. With the live observation elements designed and implemented, the different components of the robot can published their data on request. The developer can directly monitor the output and make a decision if applied changes work as expected. Due to the possibility to receive commands from the client, each component can also change its state or how its data is published. The created logging cycles allow the developer to choose from the the component of the robot and to collect only data of interest. Different outputs of different components can be combined into one output file for further processing. Due to the use of an threaded architecture multiple users can use the live observation or the logging cycles in parallel.

Live observation and the logging cycles are presented inside corresponding views on the client side. Using a tabbed navigation allows the developer to choose from the different view at any time. Each view can also be used to send a command to the corresponding component on the probe. This allows
the developer to control elements on the probe (e.g. change the detail of data to be published).

Several components of the robot platform used for this thesis were implemented into the probe. The corresponding views on the client gave a first impression on the inner processes of the robot. The capabilities of the probe and the client where tested and the results proved that the design can be used for observation and to collect information of the robot. Only for exchanging bigger payloads between probe and clients the underlying communication framework should be reconsidered or the framework should only be used to set up a communication for such data provided over other communication tunnels.

In the field of robotic a way to observe the inner process of a robot system is quite essential. With the system created in this thesis system introspection of a robot system is made possible. By observing the inner processes of the robot the developer can directly see effects of applied changes and make decisions based on the data presented.
9 Outlook

Given this new tool its now possible to think about further extensions which could no be applied to the limited amount of time. The currently implemented components on the probe cover only a short range of the components of the robot. Also some of the components implemented can offer more information then already presented. With the system designed all the other components can be presented as well and also the style of the presentation can be enhanced. System introspection allows to understand and enhance the components of the robot. Every additionally added component view can help the developer to monitor the systems behavior which can result in faster improvements of the robot system.

Some of the components of the robot can not be presented with the current communication framework (e.g. audio transmission). To achieve this an additional communication tunnel should be implemented or the underlying communication framework should be changed. There are many ideas to improve the data to be published and how to present these information to the developer.

There are several use cases for the system created. One that is often mentioned while the system was created is the observational usage when running experiments like human-machine-interaction. At the time of this thesis the inner process of the platform used could not be monitored without moving towards the robot and checking the controlling laptops. Whenever this happens the collected data may be worthless due to the fact that the appearance of the developer can influence the interaction to be observed. Due to internal processes that need time to calculate the next step the robot sometimes seems to stop its current task. When moving towards the robot it can just move on at exact that moment. The system of this thesis allows to get an idea of the inner processes and allows to minimize the need of moving towards the robot. Also the system can be extended to give concrete information on the running experiment which allows to collect additional data (e.g. presenting the current stage of the experiment showing the time needed for each stage). The system can also enhance Human-Robot-Interaction by providing additional communication queues. One idea is to extend the probe to be used by the robot as an extended sensor. A client connected to the probe can be seen as external sensor or actuator. The input of the client can be used for classifying objects.
on images provided by the robot's cameras. Another sensor could offer the
different sensors of a mobile device (e.g. accelerometer, gyrometer). All these
sensors can be used as additional input for the robot. While the robot navigates
through an unknown environment it can use the probe to request information
from all connected clients (e.g. how to navigate through the environment).

Besides the different ideas which have risen in the progress of this thesis, the
designed system already helps the developer in understanding the behavior of
the components. Moreover is it a good basis for further observation of the
inner processes of a robot system.
Bibliography


### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic concept of telerobot supervisory control</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Observation software for a telelaboratory</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Sony AIBO controlled with a HP iPAQ rx195 over the internet</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Measuring the capability of the remote control</td>
<td>8</td>
</tr>
<tr>
<td>4.1</td>
<td>ToBI platform used at the Robocup@Home competition</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>The different layer of BonSAI</td>
<td>15</td>
</tr>
<tr>
<td>4.3</td>
<td>The architecture behind android</td>
<td>16</td>
</tr>
<tr>
<td>4.4</td>
<td>Life-cycle of an android activity</td>
<td>17</td>
</tr>
<tr>
<td>4.5</td>
<td>Typical instant messaging session</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of the designed architecture</td>
<td>21</td>
</tr>
<tr>
<td>5.2</td>
<td>Message exchange between probe and client</td>
<td>22</td>
</tr>
<tr>
<td>5.3</td>
<td>Communication flow with observer pattern</td>
<td>23</td>
</tr>
<tr>
<td>5.4</td>
<td>Beacon running between probe and client</td>
<td>24</td>
</tr>
<tr>
<td>5.5</td>
<td>Reading and processing client commands</td>
<td>26</td>
</tr>
<tr>
<td>5.6</td>
<td>Reading a component with the Component Holder</td>
<td>26</td>
</tr>
<tr>
<td>5.7</td>
<td>Processing commands for Component Threads</td>
<td>27</td>
</tr>
<tr>
<td>5.8</td>
<td>Component Thread collecting and publishing data</td>
<td>28</td>
</tr>
<tr>
<td>5.9</td>
<td>Logging Dispatcher starting and stopping Logging Cycles</td>
<td>29</td>
</tr>
<tr>
<td>5.10</td>
<td>A Logging Cycle creating corresponding files</td>
<td>29</td>
</tr>
<tr>
<td>5.11</td>
<td>Starting the Client for observation</td>
<td>30</td>
</tr>
<tr>
<td>5.12</td>
<td>Checking and sending beacon</td>
<td>31</td>
</tr>
<tr>
<td>5.13</td>
<td>Principle of tabbed navigation</td>
<td>32</td>
</tr>
<tr>
<td>5.14</td>
<td>Disconnect last component and request new component</td>
<td>32</td>
</tr>
<tr>
<td>6.1</td>
<td>Overview of the system</td>
<td>33</td>
</tr>
<tr>
<td>6.2</td>
<td>The communication interface with example implementation</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Main classes of the probe</td>
<td>36</td>
</tr>
<tr>
<td>6.4</td>
<td>Base class and interface for a component thread</td>
<td>39</td>
</tr>
<tr>
<td>6.5</td>
<td>Class diagram of the mobile client with important elements</td>
<td>43</td>
</tr>
<tr>
<td>6.6</td>
<td>Login view and connecting to the probe</td>
<td>45</td>
</tr>
<tr>
<td>6.7</td>
<td>Tabbed navigation after login</td>
<td>46</td>
</tr>
<tr>
<td>6.8</td>
<td>Example class extending the TabTemplate</td>
<td>47</td>
</tr>
<tr>
<td>6.9</td>
<td>Starting logging cycles</td>
<td>48</td>
</tr>
</tbody>
</table>
## List of Tables and Listings

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Typical stanza for a message - The message stanza holds the attributes for the sender and the recipient. The text to be send is placed inside a body element.</td>
<td>19</td>
</tr>
<tr>
<td>6.1</td>
<td>Example of the XMPP configuration in the XML configuration file on the probe - <strong>ServerIP</strong> and <strong>ServerPort</strong> determine the address of the communication server. <strong>ServerAccount</strong> and <strong>Password</strong> are used to define the account used for the probe.</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>Example of the view configuration on the client - Each component has a unique name, a corresponding element on the probe and a local class name.</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Example layout for the laser data view - Each layout uses XML elements with attributes to create the view.</td>
<td>47</td>
</tr>
</tbody>
</table>
Appendix

Use Cases

<table>
<thead>
<tr>
<th>Use Case 1: Connecting to the probe</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Primary Actor</strong></td>
</tr>
<tr>
<td><strong>Secondary Actor</strong></td>
</tr>
</tbody>
</table>
| **Basic Flow** | 1. Client initiates connection process.  
2. Client establishes connection to probe.  
3. Probe adds client to the list of connected clients.  
4. Probe sends acknowledge message to client.  
5. Client switches view to status screen. |
| **Extensions** | 2a. Connection error occurs, client shows error message  
2b. Timeout occurred, client shows error message  
5a. Error on switching view, client shows error message |
### Use Case 2: Request list of available components.

<table>
<thead>
<tr>
<th>Description</th>
<th>The client requests the list of available components from the probe.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Actor</td>
<td>Client</td>
</tr>
<tr>
<td>Secondary Actor</td>
<td>Probe</td>
</tr>
<tr>
<td><strong>Basic Flow</strong></td>
<td>1. Client sends a request for the list of available components.</td>
</tr>
<tr>
<td></td>
<td>2. Probe fills message with list of available components.</td>
</tr>
<tr>
<td></td>
<td>3. Probe forwards the message to the client.</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
<td>1a. Connection error occurs, client shows error message</td>
</tr>
<tr>
<td></td>
<td>1b. Timeout occurred, client shows error message</td>
</tr>
<tr>
<td></td>
<td>2a. List not create, probe sends error to client</td>
</tr>
<tr>
<td></td>
<td>2b. Message could not be created, probe sends error to client</td>
</tr>
</tbody>
</table>

### Use Case 3: Request a component.

<table>
<thead>
<tr>
<th>Description</th>
<th>The client requests a specific component. The probe adds the client to the list of the requested component. The component begins to send data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Actor</td>
<td>Client</td>
</tr>
<tr>
<td>Secondary Actor</td>
<td>Probe, Component Thread</td>
</tr>
<tr>
<td><strong>Basic Flow</strong></td>
<td>1. Client sends a request a component.</td>
</tr>
<tr>
<td></td>
<td>2. Probe adds client to the list of the component thread and activates this thread.</td>
</tr>
<tr>
<td></td>
<td>3. Component thread forwards data to all connected clients.</td>
</tr>
<tr>
<td><strong>Extensions</strong></td>
<td>1a. Connection error occurs, client shows error message</td>
</tr>
<tr>
<td></td>
<td>2a. Component thread unknown, probe sends error to client</td>
</tr>
<tr>
<td></td>
<td>2b. Client already requests component, probe sends error to client</td>
</tr>
<tr>
<td></td>
<td>3a. Client connection lost, component thread removes client from list</td>
</tr>
</tbody>
</table>
### Use Case 4: Disconnect from component.

**Description**
The client sends a disconnect command for a component. The probe removes the client form the list of the corresponding component.

**Primary Actor**
Client

**Secondary Actor**
Probe, Component Thread

**Basic Flow**
1. Client sends a disconnect command for a component to the probe.
2. Probe remove client from corresponding component thread.
3. Component thread stops to send data to the client.

**Extensions**
1a. Connection error occurs, client shows error message
2a. Component thread unknown, probe sends error to client
2b. Client not in list of the component, probe sends error to client

### Use Case 5: Request component state.

**Description**
The client requests a list of the components states from the probe.

**Primary Actor**
Client

**Secondary Actor**
Probe, Component Threads

**Basic Flow**
1. Client sends a request a list of the component states.
2. Probe collects the state of all available components.
3. Probe add the collected states to a message and forwards it to the client.

**Extensions**
1a. Connection error occurs, client shows error message
2a. Error occurs while collecting states, probe sends error to client
### Use Case 6: Request components to be logged.

<table>
<thead>
<tr>
<th>Description</th>
<th>The client requests components to be logged on the probe.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Actor</td>
<td>Client</td>
</tr>
<tr>
<td>Secondary Actor</td>
<td>Probe</td>
</tr>
</tbody>
</table>
| Basic Flow | 1. Client sends a request to log components.  
2. Probe checks if requested components are available.  
3. Probe creates new logging cycle with requested components. |
| Extensions | 1a. Connection error occurs, client shows error message  
2a. Components not available for logging, probe sends error to client.  
3a. Error on creating logging cycle, probe sends error to client. |

### Use Case 7: Request logging to be stopped.

<table>
<thead>
<tr>
<th>Description</th>
<th>The client requests to stop logging components on the probe.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Actor</td>
<td>Client</td>
</tr>
<tr>
<td>Secondary Actor</td>
<td>Probe</td>
</tr>
</tbody>
</table>
| Basic Flow | 1. Client sends a request to stop logging components.  
2. Probe checks if components are logged by client.  
3. Probe stops the process of logging components. |
| Extensions | 1a. Connection error occurs, client shows error message  
2a. No components currently logged by client, probe sends error to client. |

### Use Case 8: Disconnect from probe.

<table>
<thead>
<tr>
<th>Description</th>
<th>The client disconnects from the probe.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Actor</td>
<td>Client</td>
</tr>
<tr>
<td>Secondary Actor</td>
<td>Probe</td>
</tr>
</tbody>
</table>
| Basic Flow | 1. Client sends a disconnect command to the probe.  
2. Probe removes the client from the list of connected clients. |
| Extensions | 1a. Connection error occurs, client shows error message |
Use Case 9: Sending beacon to keep connection.

<table>
<thead>
<tr>
<th>Description</th>
<th>The client sends a beacon to the probe. The probe answers the beacon with its own beacon.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Actor</td>
<td>Client</td>
</tr>
<tr>
<td>Secondary Actor</td>
<td>Probe</td>
</tr>
</tbody>
</table>
| Basic Flow | 1. Client sends a beacon to the probe.  
  2. Probe resets beacon counter for the client to zero.  
  3. Probe sends beacon back to client.  
  4. Client resets beacon counter to zero. |
| Extensions | 1a. Connection error occurs, client shows error message  
  2a. Client not in list of connected clients, probe sends error to client. |

Evaluation Diagrams

Figure 9.1: Latency of receiving data on the mobile device - The results of sending the different pattern with a delay of 500ms is plotted. For each pattern 200 messages were send to the client.
**Figure 9.2:** Latency of receiving data on the mobile device - The results of sending the different pattern with a delay of 1000ms is plotted. For each pattern 200 messages were send to the client.

**Figure 9.3:** Latency of receiving data on the laptop - The results of sending the different pattern with a delay of 100ms is plotted. For each pattern 200 messages were send to the client.
Figure 9.4: Latency of receiving data on the laptop - The results of sending the different pattern with a delay of 500ms is plotted. For each pattern 200 messages were send to the client.

Figure 9.5: Latency of receiving data on the laptop - The results of sending the different pattern with a delay of 1000ms is plotted. For each pattern 200 messages were send to the client.
Affidavit

I hereby declare that this master thesis has been written only by the under-
signed and without any assistance from third parties. Furthermore, I confirm
that no sources have been used in the preparation of this thesis other than
those indicated in the thesis itself.

Bielefeld, March 2011

Andreas Kipp