

ISSN 0280-5316
ISRN LUTFD2/TFRT--5702--SE

Distributed Real Time Robot Vision in Java

Samuel Kasper

Department of Automatic Control
Lund Institute of Technology
March 2003

Department of Automatic Control Lund Institute of Technology Box 118 SE-221 00 Lund Sweden	<i>Document name</i> MASTER THESIS	
	<i>Date of issue</i> March 2003	
	<i>Document Number</i> ISRN LUTFD2/TFRT--5702--SE	
<i>Author(s)</i> Samuel Kasper	<i>Supervisor</i> Rolf Johansson, Tomas Olsson, Klas Nilsson at LTH. Manfred Morari ETH, Zürich.	
	<i>Sponsoring organization</i>	
<i>Title and subtitle</i> Distributed Real Time Robot Vision in Java (Distribuerat realtidsprogrammerat robotseende i Java)		
<i>Abstract</i> <p>The goal of this thesis was to design and implement a distributed real time vision system in Java and to investigate its performance with the aid of some robot experiments.</p> <p>Controlling a process requires reference- and feedback signals which can be sensed by means of digital cameras. The image data coming from the cameras needs to be collected, processed and transmitted through the network without long time delays.</p> <p>The high-level programming language Java was used for all implementations in order to achieve portability and object orientation. The UDP protocol provides the basis for the communication tool since it allows fast and connectionless data transmission. Computer vision was developed with the aid of Java Advanced Imaging as well as Java Native Interfaces.</p> <p>Based on the linearised and simplified model of the ball-and-plate process, continuous time controllers were designed and simulated using Simulink. A discretised controller was implemented in Java.</p> <p>Various timing tests showed the excellent performance of the developed communication tool. The Java-based image processing turned out to work fast enough with respect to the sampling interval of the digital cameras. The aim of the experiment was to make a ball roll along a predefined trajectory by tilting a plate appropriately. The ball-and-plate process was successfully controlled by a PD controller using an industrial robot as actuator and digital cameras as sensors for the feedback and reference signal.</p>		
<i>Keywords</i>		
<i>Classification system and/or index terms (if any)</i>		
<i>Supplementary bibliographical information</i>		
<i>ISSN and key title</i> 0280-5316		<i>ISBN</i>
<i>Language</i> English	<i>Number of pages</i> 106	<i>Recipient's notes</i>
<i>Security classification</i>		

The report may be ordered from the Department of Automatic Control or borrowed through:
University Library 2, Box 3, SE-221 00 Lund, Sweden
Fax +46 46 222 44 22 E-mail ub2@ub2.se

Distributed Real Time Robot Vision in Java

Samuel Kasper, ETH Zürich

Master Thesis

**Department of Automatic Control
Lund Institute of Technology
Sweden**

Winter 2002/2003

Supervisors

Prof. Dr. Rolf Johansson, rolf.johansson@control.lth.se

Prof. Dr. Manfred Morari, morari@aut.ee.ethz.ch

Tutors

Dr. Klas Nilsson, klas.nilsson@cs.lth.se

Tomas Olsson, tomas.olsson@control.lth.se

Student

Samuel Kasper, skasper@ee.ethz.ch

Acknowledgments

This documentation was written at the *Department of Automatic Control* [1] of the *Lund Institute of Technology* [2] in Sweden during the winter semester 2002/2003.

The author would like to thank his tutors Klas Nilsson and Tomas Olsson for their dedicated and competent help in various aspects of this work. Klas Nilsson is an expert concerning computer science whereas Tomas Olsson is a specialist in the field of computer vision and control. Furthermore, I would like to thank Prof. Dr. Johansson and Prof. Dr. Morari for their assistive inputs.

I would also like to thank all the staff of the Department of Automatic Control who made me feel comfortable from the beginning.

Lund, 8th of March 2003

Samuel Kasper

Contents

1	Introduction	1
1.1	Problem Formulation	2
1.2	Outline of the Report	2
2	Communication in Distributed Real Time Systems	3
2.1	Analysis of the TCP-Based Architecture	3
2.1.1	Client-Server Communication	3
2.1.2	The Nameserver	6
2.1.3	Weak Points	7
2.2	Demands for a Distributed Real Time System	9
2.3	System Design	11
2.3.1	Network Layers	11
2.3.2	UDP Protocol	11
2.3.3	UDP-Based Design	12
2.4	Software Development	12
2.4.1	Java-Based Networking Classes	14
2.4.2	Java-Based Data Structures	17
2.5	Tested Time Delays	19
2.5.1	UDP versus TCP Using Linux PCs	19
2.5.2	UDP versus TCP Using Sun Workstations	21
2.5.3	Long-Time Measurements	21
3	Computer Vision	25
3.1	Problem Definition	25
3.2	Approaches	25
3.2.1	Thresholding	26
3.2.2	Edge Detection	28
3.2.3	Other Approaches	29
3.3	Java Advanced Imaging	29
3.3.1	JAI API	29
3.3.2	Implementation	30
3.4	Time Testing	31
3.4.1	Average Image-Processing Time	31

3.4.2	Graphical Evaluation	32
4	Interfaces	36
4.1	Camera Interface	36
4.1.1	Overview	36
4.1.2	Modification of the C-Implementation	36
4.1.3	Java Native Interfaces	37
4.1.4	Combination of the Subsystems	38
4.1.5	Timing Tests	39
4.2	Matlab Java Interface	39
5	Multi-Camera Robot Control	41
5.1	Ball-and-Plate Process	41
5.1.1	Overview	41
5.1.2	Block Diagrams	41
5.2	Model of the Plant	42
5.2.1	Ball-and-Beam System Model	42
5.2.2	Ball-and-Plate System Models	43
5.3	Design of the Ball-Position Controller	44
5.3.1	Time Response of the Plant	45
5.3.2	Continuous-Time Controller	45
5.3.3	Java Implementation of the Controller	51
5.4	The Robot System	51
5.4.1	System Overview	51
5.4.2	Motion Control	52
5.4.3	Robot Communication	52
5.5	Robot Experiments	53
5.5.1	Experiment Based on Visual Feedback	53
5.5.2	Experiment Based on Two Digital Cameras	54
5.5.3	Experiment Based on Two Robots	56
5.5.4	Problems and Possible Solutions	56
5.6	Results	57
6	Conclusions	60
6.1	Summary	60
6.2	Outlook	61
	Bibliography	63
A	Used Software Tools	65
B	Used Hardware	66
C	Analysis of Serialised Streams	67

D	Short User Manual	70
D.1	General Information	70
D.2	Running the Experiments	71
E	Source Code Extract	72
E.1	LabComm	72
E.2	Computer Vision	81
E.3	Interfaces	84
E.4	Controller	88
E.5	Matlab	89

List of Figures

2.1	Protocol encapsulation	4
2.2	A client-server connection	5
2.3	Communication and feedback control of a robot	6
2.4	Step 1: Server registers to the nameserver	7
2.5	Step 2: Client makes a lookup request	8
2.6	Step 3: Client connects to the server	8
2.7	Approaches for a distributed system	10
2.8	The layers of a network	11
2.9	Flow of the visual data through the network	13
2.10	The node's state-diagram	13
2.11	Making TCP connections	15
2.12	Sending/receiving datagram packets via UDP	15
2.13	The implemented data structure	19
2.14	Network time-delay for 2-way UDP transmissions (2 kbytes)	22
2.15	Histogram referring to diagram 2.14	23
2.16	Network time-delay for 2-way UDP transmissions (2 kbytes)	23
2.17	Network time-delay for 2-way UDP transmissions (2 kbytes)	24
2.18	Histogram referring to diagram 2.17	24
3.1	Grey-scaled camera image before thresholding operation	26
3.2	Grey-scaled camera image after thresholding operation	27
3.3	Ball- and fixed point position in frame number i	27
3.4	Ball- and fixed point position in frame number $i+1$: Tracking rectangle centered around the previous ball-center	28
3.5	Grey-scaled camera image after Sobel edge detection	29
3.6	Processing time for threshold operation and image analysis on a Sun machine	32
3.7	Histogram referring to diagram 3.6	33
3.8	Processing time for threshold operation and image analysis on a Linux machine	33
3.9	Histogram referring to diagram 3.8	34
3.10	Processing time for threshold operation and image analysis on a Linux machine using incremental garbage collection	35
3.11	Histogram referring to diagram 3.10	35

4.1	Sony DFW-V300 digital camera	37
4.2	Java Native Interface (JNI)	38
4.3	Combination of the implemented subsystems	39
5.1	Cascaded closed loop control system	42
5.2	The vision system: image as the input and ball position as the output	42
5.3	The ball-and-beam system	43
5.4	The plate and its descriptive coordinates	44
5.5	The linearised and simplified plant	45
5.6	Smoothened step	46
5.7	Step response of the linearised and simplified system . . .	46
5.8	The linearised process with a PD controller	47
5.9	Input and output for the PD regulated plant, cut-off fre- quency $\omega_n = 4$ rad/s and damping ratio $\delta = 0.7$	48
5.10	Input and output for the PD regulated plant, cut-off fre- quency $\omega_n = 1.15$ rad/s and damping ratio $\delta = 1.1$	48
5.11	The linearised process with a PID controller	49
5.12	Input and output for the PID regulated plant, cut-off fre- quency $\omega_n = 4$ rad/s, damping ratio $\delta = 0.9$ and $\alpha = 1$. . .	50
5.13	Input and output for the PID regulated plant, cut-off fre- quency $\omega_n = 1$ rad/s, damping ratio $\delta = 0.9$ and $\alpha = 1$. . .	50
5.14	The industrial robot that serves as an actuator	52
5.15	Robot motion control with two cascaded PID controllers . .	53
5.16	The plate and digital camera attached to the robot	54
5.17	Robot experiment based on visual feedback	55
5.18	Robot experiment based on two digital cameras	55
5.19	Reference- and feedback signal, experiment with one cam- era	58
5.20	Reference- and feedback signal, experiment with two cam- eras	58
5.21	Control signals for joint 4 and 6 referred to Figure 5.19 . .	59
5.22	Comparison of control- and position signal (extraction of Figure 5.21)	59

List of Tables

2.1	Two-way matrix transmission via TCP and UDP using Linux PCs (serialisation)	20
2.2	Two-way matrix transmission via TCP and UDP using Linux PCs (loop)	20
2.3	Two-way matrix transmission via TCP and UDP using Sun Ultra10 (serialisation)	21
2.4	Two-way matrix transmission via TCP and UDP using Sun Ultra10 (loop)	21
3.1	Average processing times for Java image analysis	32
5.1	Experimental parameters for the PD controller	57

Chapter 1

Introduction

This project is about distributed real time robot vision in Java. Nowadays, distributed systems consisting of several communicating nodes are very common because they make it possible to run processes on the machine that is most suitable. Furthermore, a potentially time consuming operation like computer vision can be executed on a host that is prevented from doing other calculations. The communication software MatComm (see Section 2.1) has been developed at the Department of Automatic Control of LTH and was taken as initial point for the design of a new communication tool for distributed real time systems.

Controlling a process with the aid of a robot requires an underlying real time system. As a consequence, the time delays of the network as well as the processing time of the participating processes are crucial and have to be short. Using a distributed system helps to separate the different tasks (e.g., image processing and control) locally, but on the other hand, the nodes must be able to quickly exchange data between each other. Previous work concerning real time systems and time delays has been done at the department [3],[4].

Modern control makes often use of visual sensors since they are fast, precise and relatively simple to install. Digital cameras provide sampling frequencies starting at 30 Hz and they are very convenient to acquire reference- and feedback signals in a robot environment. Examples of earlier investigations dealing with visual servoing are shown in [5] and [6]. The image data is large and its analysis requires sophisticated algorithms in order to be done in a reasonable time.

To demonstrate real time robot vision, a plate is attached to a industrial robot and it is slanted in such a way that a ball located on this plate follows a predefined line. The experiment is based on a previous master thesis [7].

1.1 Problem Formulation

This thesis deals with three independent problems as well as with the combination of them. First of all, a fast and portable communication software is required in order to enable efficient data transmission in a distributed real time system. Additionally, digital cameras serving as sensors in control applications have to be integrated in a Java environment. Thus, camera interfaces and suitable Java-based image processing methods are demanded. For experimental validation of the concept, we chose real time control for the ball and plate process. Finally, that process should be actuated by an industrial robot using digital cameras as visual sensors.

1.2 Outline of the Report

Chapter 2 presents the development of the communication software for a distributed real time system.

Chapter 3 describes the computer vision. The Java Advanced Imaging API is chosen as development environment.

Chapter 4 explains how the different parts of the real time system are put together with the aid of interfaces.

Chapter 5 is about multi-camera robot control. The ball-and-plate process is taken as a demonstrator.

Chapter 6 concludes the report and shows possible extensions for further development.

Appendix A contains all the software tools that were used for this project.

Appendix B enumerates the hardware that was used for the experiments and for the development.

Appendix C explains how serialised output streams are represented in Java.

Appendix D demonstrates how the implemented software packages have to be utilised.

Appendix E is a collection of Java, C and Matlab code written at various stages of the project.

Chapter 2

Communication in Distributed Real Time Systems

2.1 Analysis of the TCP-Based Architecture

In order to exchange information between different hosts, a communication software is required. Controlling a robot usually needs more than one computer since the different tasks can be very time consuming. For instance, the controller could run on one machine whereas the image processing for the feedback signal is done on another host. As a consequence, the processed data has to be sent from the vision host to the control host.

So far, the software package MatComm was used at the Department of Automatic Control for interprocess communication. MatComm was developed at the afore mentioned department and is written in C. The main idea of this software is to send and receive Matlab matrices over TCP sockets. MatComm exists in different versions supporting several main platforms such as Solaris, Irix and Win32.

2.1.1 Client-Server Communication

MatComm is based on TCP/IP. IP, the Internet Protocol, has a number of advantages over its competitors; AppleTalk and IPX, most of which stem from its history. Due to its development time being during the Cold War, the protocol ended up with many features that interested the military. It had to be robust and therefore, IP was designed to allow multiple routes between two points with the ability to route packets of data around damaged routers. The protocol also had to be open and platform-independent.

Since there are multiple routes between two points and since the shortest path between two points may change over time as a function of network traffic, the packets that make up a particular data stream may not all take the same route. Furthermore, they may not arrive in the order they were sent, if they even arrive at all. To improve on the basic scheme, TCP, the Transmission Control Protocol, was added. TCP gives each end of a connection the ability to acknowledge receipt of IP packets and request retransmission of lost packets. Moreover, TCP allows the packets to be put back together on the receiving end in the same order they were sent on the sending end. Figure 2.1 shows the protocol encapsulation.

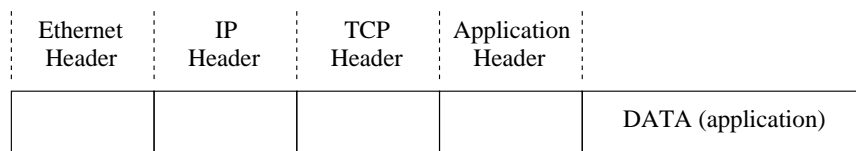


Figure 2.1: Protocol encapsulation

TCP also adds to the IP Protocol the concept of port numbers. By using port numbers, TCP identifies which service a particular transmission is intended for on the destination machine. Ports are numbered ranging from 0 to 65000, which allow transmissions to be sent directly to a particular piece of software that is listening to the specified host. Port numbers between 1 and 1023 are reserved for well-known services.

Data is transmitted across the network in packets of finite size called datagrams. Each datagram consists of a header and a payload. The header contains the address and port the packet is going to, the address and port the packet came from and various other information used to ensure reliable transmission. The payload contains the data itself. However, since datagrams have finite length, it is often necessary to split the data across multiple packets and reassemble it at the destination.

Sockets are an invention of Berkeley UNIX. They shield the programmer from low-level details of the network, like media types, packet sizes, packet retransmission, network addresses and more. Sockets are available for Unix, Windows, Macintosh and Java. A socket can perform seven basic operations:

- Connect to a remote machine
- Send data
- Receive data

- Close a connection
- Bind to a port
- Listen for incoming data
- Accept connections from remote machines on the bound port

Whereas both clients and servers use the first four operations, the last three are only needed by servers that have to wait for clients to connect to them. As a consequence, a client can only connect to a server if the server is listening when the client makes the request. Figure 2.2 illustrates a client-server connection. Note that the chosen port numbers are just examples.

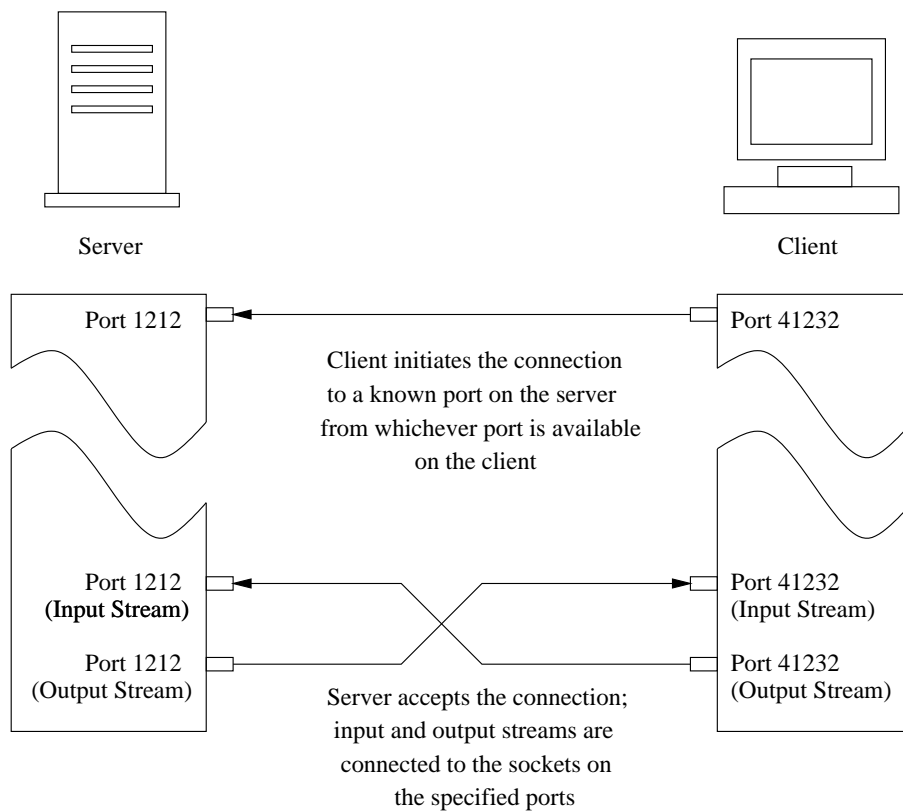


Figure 2.2: A client-server connection

In the following paragraph, it is explained how the described mechanisms are utilised in a robot experiment using visual feedback. The

controller for the robot runs on one machine while the image processing is done on another computer. This computer is also connected to a camera that acts as visual sensor. The two machines use MatComm in order to communicate with each other. The client-server model has to be identified such that the control computer acts as a server, listening for a client (the vision computer) that wants to connect (see Figure 2.3). The server must run and listen before the client tries to connect, and as soon as the connection is established, the client can send data (Matlab matrices) to the server. This data consists of nothing other than the feedback signal used for the controlling.

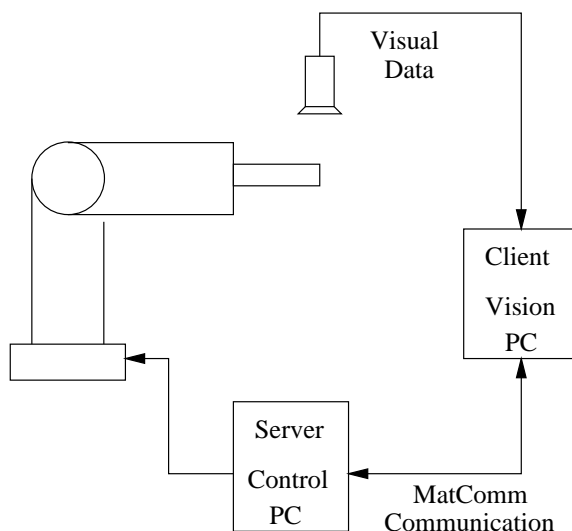


Figure 2.3: Communication and feedback control of a robot

On the application level, MatComm provides functions such as looking for available servers, starting a server, opening a connection to a server and closing an existing connection. In addition, Matlab matrices can be sent and received. This set of methods allows full control over a client-server setup.

2.1.2 The Nameserver

A server is listening for client requests on one specific port. For example, the HTTP service, which is used by the Web, generally runs on port 80. Clients on the other hand are only able to connect to a server if they know its address and the port number where it is listening. A nameserver prevents clients from dealing with port numbers and IP addresses. Every time a new server is started on any host, it is registered

to the nameserver. The nameserver stores and updates a database consisting of entries that belongs to a specific server. An entry contains at least the name of the service and the IP address of the host where the service is running on and the associated port number.

Before a client can connect to a server, it checks what services are available or if a specific service is running by making a lookup request to the nameserver. The client then gets the important parameters such as the port number and IP address associated with the desired service. The only port number and IP address the client has to know is the one of the nameserver. Otherwise, it will not be able to connect to the nameserver. Therefore, the nameserver is usually running on a default host listening on a default port. Figures 2.4 to 2.6 illustrate how a client connects to a server with the aid of a nameserver.

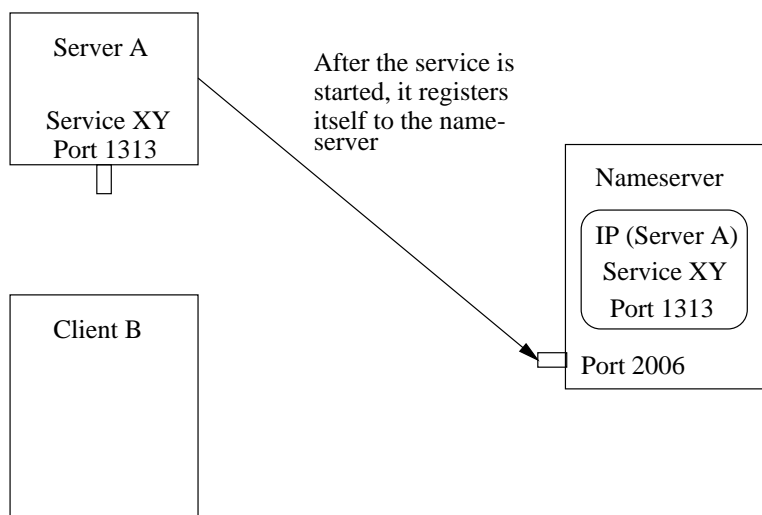


Figure 2.4: Step 1: Server registers to the nameserver

2.1.3 Weak Points

In this subsection, we discuss some of the weaknesses of MatComm, which exist due to it is being written in C and based on TCP/IP. First of all, the programming language C is not platform-independent and consequently different versions of MatComm are required for the different platforms. An update of MatComm can be very time consuming because all versions need to be modified. C does not provide expanded network libraries and thus, the resulting source code is somewhat complicated and extensive. There is also a lack of a good documentation tool, thus

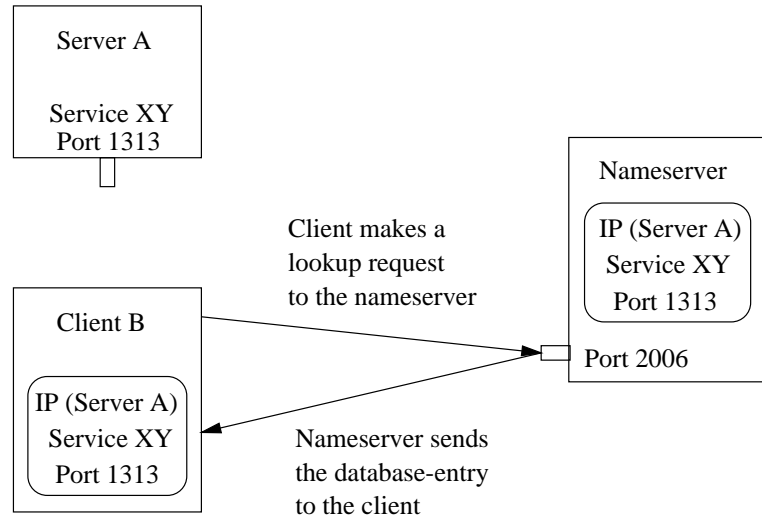


Figure 2.5: Step 2: Client makes a lookup request

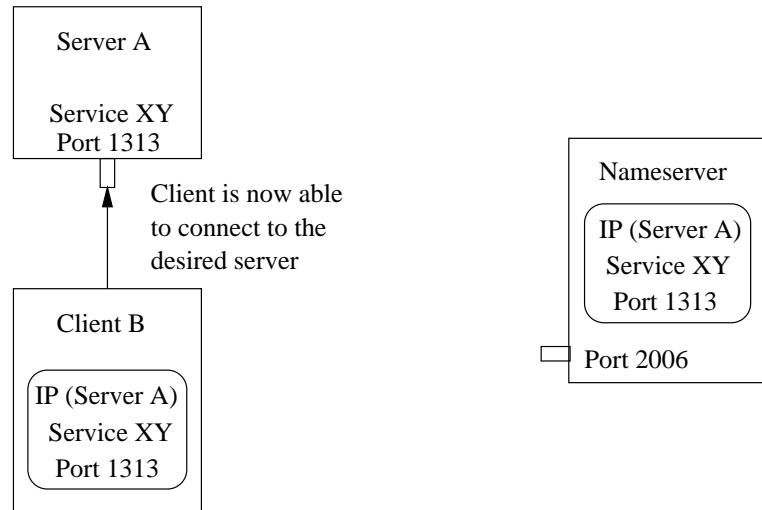


Figure 2.6: Step 3: Client connects to the server

understanding the code becomes time consuming.

As described above, TCP has a high reliability although it also carries a fair amount of overhead. This comes at a price: speed. Section 2.5 shows that the TCP based communication is slower compared to UDP. Let us consider the robot experiment with visual feedback. Using digital cameras, the sampling rate is high, hence the order of the data is not particularly important and the loss of individual packets will not completely corrupt the stream. However, speed is a crucial parameter. Accordingly, it would be more appropriate to use a faster protocol and

to abandon the high reliability.

TCP is a connection-based protocol meaning that a client is only able to communicate with a server if a connection was established between them. As long as this connection is open, no other client can connect to the same port on the server since that port is already used. Considering only a simple client-server setup, there are no problems. However, in a distributed system, several clients want to communicate with a server. If all clients send their data constantly to the same service (like in the robot experiment), the connection-based approach is no longer convenient.

Finally, MatComm's nameserver is also implemented in C and thus, it is not portable and runs always on the same machine. Clients and servers know the IP address and the port number of this machine and so they can connect to the nameserver. If for a certain reason the nameserver's host is not running, the connectivity is blocked. In conclusion it would be better if an instance of the nameserver could run on every host.

2.2 Demands for a Distributed Real Time System

The term *real time* is used to describe any information-processing activity or system that has to respond to externally generated input stimuli within a finite and specified delay. In a real time control system, the computer is interfaced directly to the process that has to receive a signal within limited time. Some important characteristics for an ordinary real time system are listed below:

- It must be extremely reliable
- Time requirements must be met
- The system is often large and complex
- It has to interface with non-standard I/O devices

The following enumeration is valid with reference to the robot experiment mentioned earlier in this chapter. Every system has its own specialties and particular demands although before designing a system, analysis must be done to find out exactly what is required.

First of all, the image acquisition as well as the processing and the communication need to be fast. Analogue cameras in combination with frame grabbers are too slow and thus, digital cameras should be used. The image analysis is a rather expensive task and must be performed

on an appropriate computer. Every camera should be connected to a separate machine and the time delays for the transmission of the feedback signal over the network must not be larger than an upper limit. Otherwise, the controlling is impossible. Consequently, communication has to be type-based, meaning fast but less reliable feedback data and slower but highly reliable transmission of commands or files. There is not a single protocol that could fulfill all these demands. Thus, at least two different protocols are required.

Regarding the topology, two different approaches have to be considered. Whereas one approach assumes all participating nodes as equal, the other distinguishes between clients and a central server (see Figure 2.7).

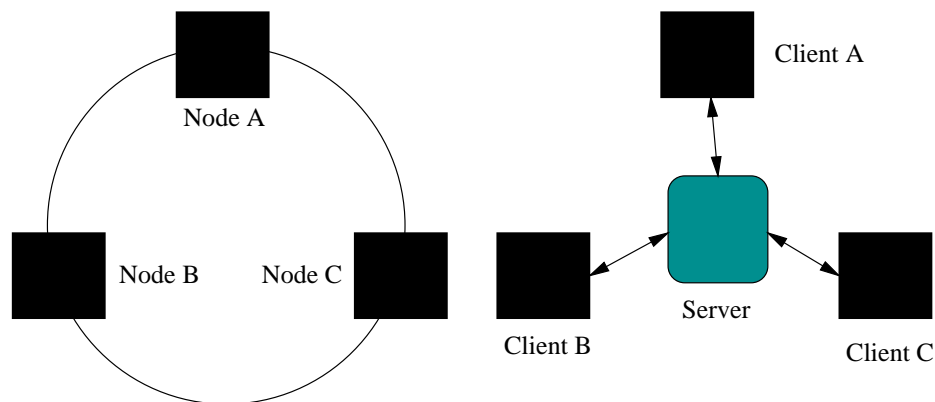


Figure 2.7: Approaches for a distributed system

If communication is only between a single machine and several other machines, the client-server model seems to be the more convenient. A client-server application typically stores large quantities of data on an expensive, high-powered server, while most of the program logic and the user interface is handled by client software running on relatively cheap personal computers. However, this client-server model does not fit very well into the experimental setup of our system consisting of a robot and cameras for feedback information. The single nodes (computer vision and controlling) are client and server at the same time and data must be sent between all participating nodes. Consequently, a peer-to-peer approach to communication is more appropriate.

2.3 System Design

2.3.1 Network Layers

There are different layers of communication on a network. Each layer represents a different level of abstraction between the physical hardware and the information being transmitted. A layer only talks to the layers immediately above and below it. Figure 2.8 illustrates a simplified four-layer model, although the real details are much more elaborate. However, since software development is mostly done in the application and transport layer, the complexity of the physical layer is hidden. Once data is transmitted across the physical layer, it bubbles up through the layers on the receiving end. It appears, to the application layer, that it is talking directly to the application layer on the other system; the network creates a logical path between the two application layers.

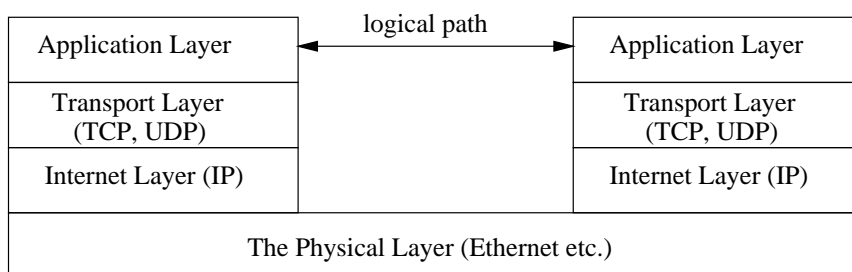


Figure 2.8: The layers of a network

2.3.2 UDP Protocol

UDP (User Datagram Protocol) was chosen as new protocol for fast data communication. It is utilised to send data that does not necessarily need to be very reliable. The UDP packet is encapsulated in an IP packet and can consist of up to 65507 bytes of data. UDP does not need to initiate or end a transmission because it is connectionless, only packets are sent. UDP features the following advantages:

- Very high speed (up to three times faster than TCP)
- Excellent way to broadcast position information
- Excellent performance under noisy network conditions

- Very useful for sending information that will be updated at short intervals

Using UDP, ports do not have to be bound in order to establish a connection. If a node wants to send data to another node, it simply needs to know the IP address of the remote host as well as the port on which the desired service is running. It then packs the data into a UDP packet and sends this packet over the physical network layer. As a consequence, a port can receive data packages from different senders at the same time. That is exactly what we need in our distributed system. The only limitation is the packet size since the data has to fit into one UDP packet. However, 65507 bytes should be enough for feedback information.

2.3.3 UDP-Based Design

We are now ready to put all parts together. Concerning the transport layer, UDP is chosen for fast feedback-data transmission whereas TCP can be used for the transmission of commands or files. All nodes are able to communicate with each other; no server is required. A nameserver prevents the nodes from dealing with IP addresses and port numbers.

At least two machines are needed for the image acquisition and processing while the controller runs on another computer. Figure 2.9 shows how the visual data flows to the controller where it is interpreted as feedback information. The UDP packages are sent to the same port. Since every UDP package carries the information from where it has been sent, the single packages can be associated with the appropriate sender.

The nodes have to be in a defined state. Every node registers to the nameserver before sending or receiving any data (compare Figure 2.10). Consequently, communication works between all participants. For instance, the nodes in charge of the image processing are able to exchange data if required. In addition, the nodes can send and receive data simultaneously. This generic design opens up many possible applications.

In summary, the use of the User Datagram Protocol (in addition to TCP) makes the design very flexible and powerful. The demands of a real time distributed system can be fulfilled. From a network point of view, every node is equal, which makes the topology simple and easily understandable.

2.4 Software Development

Usually, diverse platforms are used in a laboratory (Windows, Unix, Linux, ...). Therefore, the communication software should be portable

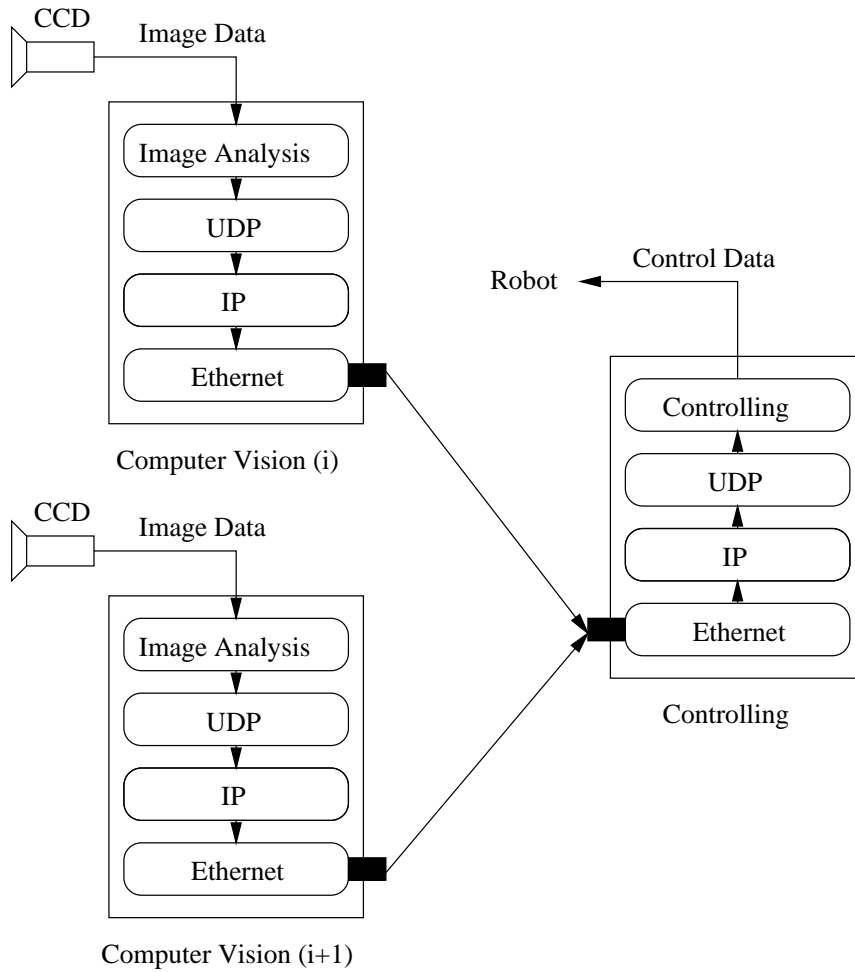


Figure 2.9: Flow of the visual data through the network

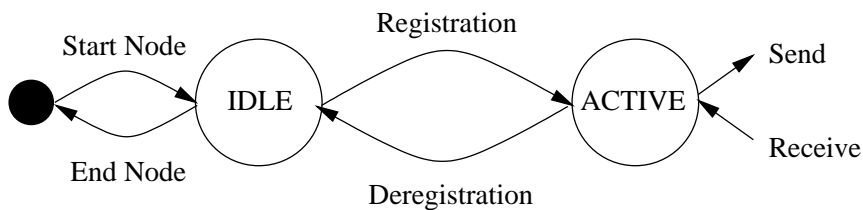


Figure 2.10: The node's state-diagram

in order to avoid several different versions. The programming language Java [8] enables programmers to write platform-independent code. As long as there exists a virtual machine for a specific architecture, the Java classes can be used without modification. Furthermore, Java provides a wide range of class libraries. In particular, *Java.net* offers a good

basis to deal with UDP and TCP protocols. Finally, in order to make source code more comprehensible, a clear documentation is required. Java is equipped with a tool that allows the automatic production of a HTML-based documentation. For all these reasons, Java was chosen as programming language whereas [9] and [10] were used as very helpful references.

The implemented package is called *LabComm*. A more general name than *MatComm* was taken since *LabComm* does not only support communication between Matlab but also between other tools used in a laboratory. The package includes network classes (see Section 2.4.1) as well as classes for the defined data formats (see Section 2.4.2). The documentation in combination with the provided demonstrations implies a quick learning curve.

LabComm is based on *MatComm*. The original *MatComm* was developed by Anders Blomdell at the Department of Automatic Control of LTH Lund. It is written in C and uses TCP as underlying network protocol. On the other hand, Mathias Haage who works at the Department of Computer Science of LTH implemented a Java version of *MatComm*. It is also using the TCP protocol. Both versions were taken as starting point for the implementation of *LabComm*.

LabComm provides the features of *MatComm*, but it is more powerful since it also includes the User Datagram Protocol what makes it more convenient for distributed systems.

Finally, the Java concept of serialisation was applied which makes it possible to write not only built-in but also custom classes to a stream and to write from the stream respectively. As a consequence, *LabComm* allows to send and receive not only Matlab matrices but also structs.

2.4.1 Java-Based Networking Classes

As mentioned above, the UDP and the TCP protocol are used for the transport layer, both of these protocols are supported by Java. A good introduction to Java network programming is given in [11].

The *ServerSocket*- and *Socket* class are related to building normal TCP connections. *ServerSocket* represents the socket on a server that waits and listens for requests of service from a client. *Socket* represents the endpoints for communication between a server and a client. When a server receives a request for service, it creates a *Socket* for communication with the client and continues to listen for other requests on the *ServerSocket*. The client also creates a *Socket* for communication with the server (refer to Figure 2.11). The *accept()* method blocks the caller until a connection has been established. Once the connection is established, *getInputStream()* and *getOutputStream()* may be used in communication between the sockets.

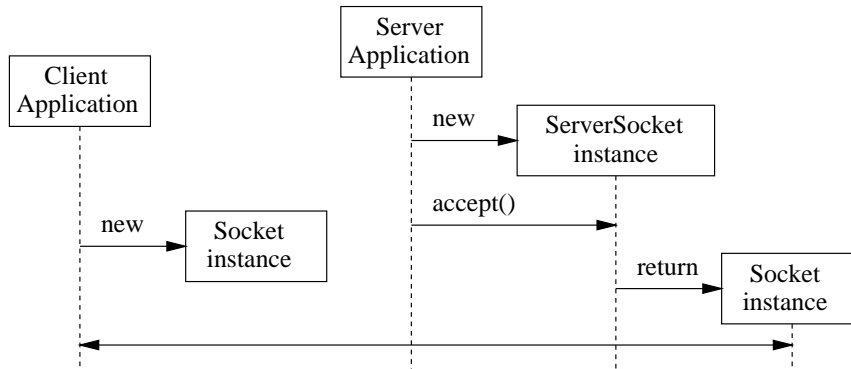


Figure 2.11: Making TCP connections

On the other hand, the *DatagramPacket*- and *DatagramSocket* class are related to sending and receiving datagram packets via UDP. *DatagramPacket* represents a datagram packet that is used for connection-less delivery and normally include destination address and port information. *DatagramSocket* is a socket used for sending and receiving datagram packets over a network via UDP. A *DatagramPacket* is sent from a *DatagramSocket* by calling the *send()* method of *DatagramSocket* with *DatagramPacket* as the argument. The *receive()* method is used for receiving a *DatagramPacket*. The sequence is shown in Figure 2.12.

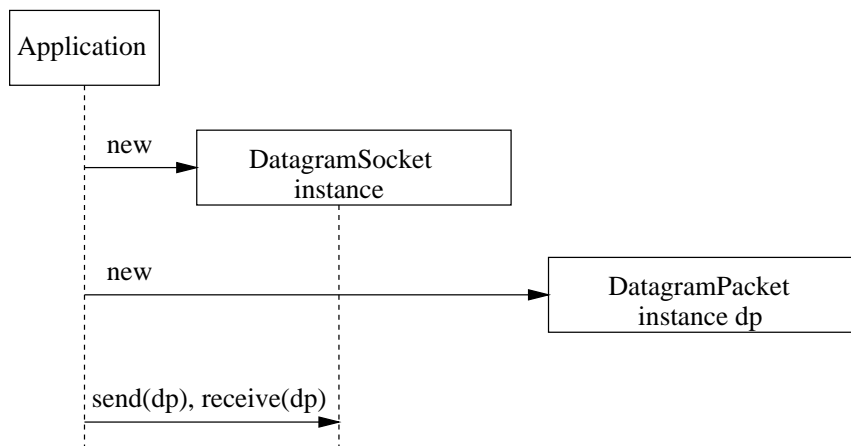


Figure 2.12: Sending/receiving datagram packets via UDP

In order to be able to send and receive data at the same time, multiple threads are needed. That is why the implementation of a UDP node consists of the classes *UDPNode*, *UDPSendThread* and *UDPRe-*

ceiveThread. The threads handle the sending and receiving, respectively of the data while the *UDPNode* class provides functions to start a UDP node, to register it, to connect to another host and to terminate the node. Threads are a powerful instrument to make the program easily controllable. It is also simple to add more threads and thus enhance the functionality.

Regarding the TCP implementation, there is the client class called *TCPClient*, the server class named *TCPServer* and a thread for the server, the class *TCPServerThread*. The server is waiting for a client that wants to connect. Thus, a thread is required. On the other hand, the client does not need an additional thread since it is not listening for incoming requests.

All nodes have to communicate with the nameserver. In case of TCP, only the servers register to the nameserver. On the other hand, all UDP nodes are registered. A special client called *StorkClient* is responsible for the communication with the nameserver. Every entry of the nameserver's database contains the following elements:

- A protocol type (e.g. UDP-MatComm)
- An IP address (e.g. 130.253.83.22)
- A port number (e.g. 4853)
- A service name (e.g. realIT)

All services can be identified by these four values. For interaction between the *StorkClient* and the *StorkNameServer*, the *StorkPacket* is used. This packet consists of the following elements:

- A version number (for internal check)
- The kind of the request (e.g., lookup, reply, register)
- A list containing entries of the nameserver
- The IP address of the nameserver
- The port number listening for client requests

The nameserver has only three tasks to perform: registration of a new node, deleting an existing registration if a node is no longer active and forwarding its list with the node entries. It uses a thread that is listening for incoming (UDP)-packets. As soon as a packet is received, it is transformed to a *StorkPacket*, the version is checked and the task is chosen according to the kind of request. Because the nameserver was implemented in Java, it is much simpler than the C-version and it is no longer bound to a default host. For more details, refer to Appendix E.1.

2.4.2 Java-Based Data Structures

As described above, the original MatComm provides only Matlab matrices for communication. These matrices are two-dimensional arrays of type float, double and character. When any of these matrices are sent, they are written element by element to the output stream and conversely received, element by element from the input stream. Built-in types (e.g., double, float, integer) can be handled by streams without problems. However, the bigger the matrices are, the more time it takes to write them to the stream element by element and the read them from a stream, in the same way. The cost increases with factor n^2 assuming that n is the dimension of a square matrix. In addition, it is desirable to have additional data structures than just matrices. Often, structs are an appropriate representation, especially in a control context. The following is an example of how a struct can look like:

```
public class struct {
    public String name;
    public Object value;
}
```

The value of the struct could be an array, a matrix or another struct. If sophisticated data such as structs has to be sent and received, a new approach must be chosen: Java object serialisation. Object serialisation extends the core Java Input/Output classes with support for objects. It supports the encoding of objects, and the objects reachable from them, into a stream of bytes. In addition, it supports the complementary reconstruction of the object graph from the stream. Serialisation can be used for communication via sockets. The default encoding of objects protects private and transient data, and supports the evolution of the classes. A class may implement its own external encoding and is then solely responsible for the external format.

An object is serialisable only if its class implements the *Serializable* interface. Within the Java programming language, an interface is a device that unrelated objects use to interact with each other. Interfaces are used to define a protocol of behaviour that can be implemented by any class anywhere in the class hierarchy.

The *Serializable* interface is an empty interface. It does not contain any method declaration. Its purpose is simply to identify classes whose objects are serialisable. The serialisation of instances of a class implementing the *Serializable* interface are handled by the *defaultWriteObject* method of the output stream. This method automatically writes out everything required to reconstruct an instance of the class. For many classes, this default behaviour is good enough. However, default serial-

isation can be slow, and a class might want more explicit control over the serialisation.

Serialisation can be customised for classes by providing two methods for it: *writeObject* and *readObject*. The *writeObject* method controls what information is saved and is typically used to append additional information to the stream. On the other hand, the *readObject* method either reads the information written by the corresponding *writeObject* method (in the same order in which it was written) or can be used to update the state of the object after it has been restored. The specification of Java's object serialisation is featured in [12].

For complete, explicit control of the serialisation process, a class must implement the *Externalisable* interface which is a subinterface of the *Serializable* interface. For externalisable objects, only the identity of the object's class is automatically saved by the stream. The class is responsible for writing and reading its contents, and must coordinate with its superclasses to do so. We decided to select the *Externalisable* interface in order to have a fully controllable design. An additional investigation shows how objects are represented after they were written to a stream using the serialisation concept (see Appendix C).

In the following, the chosen structure is introduced. Figure 2.13 illustrates an overview.

As can be seen, only the abstract classes implement the *Serializable* interface. The reason is that no read- or write method can be defined for the abstract classes. The *Serializable* interface prevents us from implementing mandatory read- and write methods.

So far, *LabObject* is the superclass for matrices and structs. If required, other types like arrays can easily be appended. A matrix of type double, float, character, integer or string has to be packed into a *MatrixPacket* before being sent. Such a packet contains the subsequent elements:

- A version number (e.g. 0x4d415430)
- The type of the matrix (e.g. TYPE_DOUBLE)
- The number of rows of the matrix
- The number of columns of the matrix
- The two-dimensional matrix itself

Finally, the class *LabStruct* provides the fields and methods for a more general data structure than just Matlab matrices. Since it implements the externalisable interface, it can be written to a stream as well as read from a stream. This class is also the superclass for further even more specific structs that may be used in combination with certain software packages. For more details, refer to Appendix E.1.

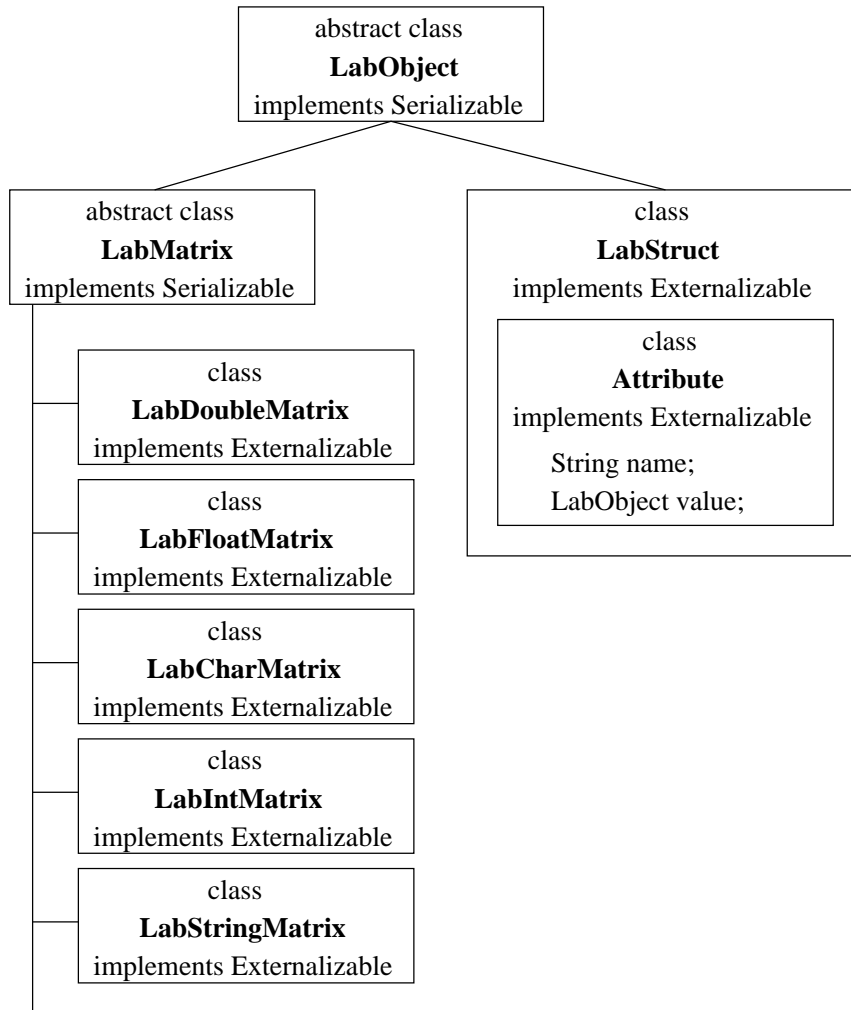


Figure 2.13: The implemented data structure

2.5 Tested Time Delays

The implemented classes were tested by means of demonstration programs. In particular, the occurring time delays when data is sent over a network need to be measured and discussed.

2.5.1 UDP versus TCP Using Linux PCs

The demo-program runs on two different machines connected via Ethernet. A test-matrix with variable size is sent from one node to the other and the receiving node sends it back to the sender. This has to be done since the system time of different machines is not exactly synchronised.

The difference between the time the matrix is sent and the time it is received describes the searched transmission delay. Finally, the average delay over a hundred transmissions is taken. Note that using the TCP protocol, the sequent matrix cannot be sent earlier than 100 milliseconds afterwards, otherwise, the receiver will not be able to handle the incoming data. The UDP protocol does not have this restriction for small data packages. Table 2.1 lists the measured values. Every measurement was repeated three times and is now represented by its average value. The matrix is written to the stream using the *Externalisable* interface. The experiment was performed for a TCP based client-server setup as well as for UDP nodes.

Matrix Size	TCP Setup	UDP Setup
64 bytes	3 ms	3 ms
1.4 kbytes	5 ms	4 ms
8 kbytes	6 ms	5 ms
16 kbytes	7 ms	7 ms

Table 2.1: Two-way matrix transmission via TCP and UDP using Linux PCs (serialisation)

The same experiment as described above was performed for matrices written element by element to the stream. Table 2.2 shows the results.

Matrix Size	TCP Setup	UDP Setup
64 bytes	2 ms	1 ms
1.4 kbytes	35 ms	2 ms
8 kbytes	4 ms	4 ms
16 kbytes	6 ms	8 ms

Table 2.2: Two-way matrix transmission via TCP and UDP using Linux PCs (loop)

If the matrices are written element by element to the stream, the transmission turns out to be slightly faster for small data packages. However, there is no big difference if the amount of data increases. Comparing UDP and TCP, UDP works faster (as expected) and allows higher receiving frequencies for small data packages. Furthermore, TCP-based transmission sometimes leads to unexpected and un-logical results. Why, for instance, should it be faster to transmit 8 kbytes than 1.4 kbytes of data (compare Table 2.2)? The only unexpected result regarding UDP was that packages larger than 54.4 kbytes could not be received although the maximal package size for UDP packets is 65,5

kbytes. In a real experiment, the transmitted data will be smaller than 54.4 kbytes and so the problem does not arise.

2.5.2 UDP versus TCP Using Sun Workstations

Let us consider the same two experiments as described above. However, relatively slow Sun workstations (Ultra10) are used instead of the very fast Linux machines. The results show many big differences between the diverse measurements. Later on, however, only fast machines will be used for the experiments. Tables 2.3 and 2.4 list the results.

Matrix Size	TCP Setup	UDP Setup
64 bytes	9 ms	8 ms
1.4 kbytes	100 ms	13 ms
8 kbytes	210 ms	27 ms
16 kbytes	140 ms	42 ms

Table 2.3: Two-way matrix transmission via TCP and UDP using Sun Ultra10 (serialisation)

Matrix Size	TCP Setup	UDP Setup
64 bytes	4 ms	3 ms
1.4 kbytes	190 ms	9 ms
8 kbytes	210 ms	30 ms
16 kbytes	220 ms	51 ms

Table 2.4: Two-way matrix transmission via TCP and UDP using Sun Ultra10 (loop)

As can be seen, UDP-based transmissions are always faster than the TCP-based. In addition, the measured delays seem to be well scaled for UDP while TCP leads to unexpected results again. Comparing the two tables, it can be concluded that the approach using serialisation is faster for package sizes starting at about 1 kbyte.

2.5.3 Long-Time Measurements

In order to get a representative average value as well as other statistic data, the measurements need to be done over longer periods. Only the fast Linux machines are taken into account. Packages of a typical size (2 kbytes) are transmitted via UDP. All the sending- and receiving times

are written to a file and analysed with Matlab. The results are shown in Figures 2.14 to 2.18.

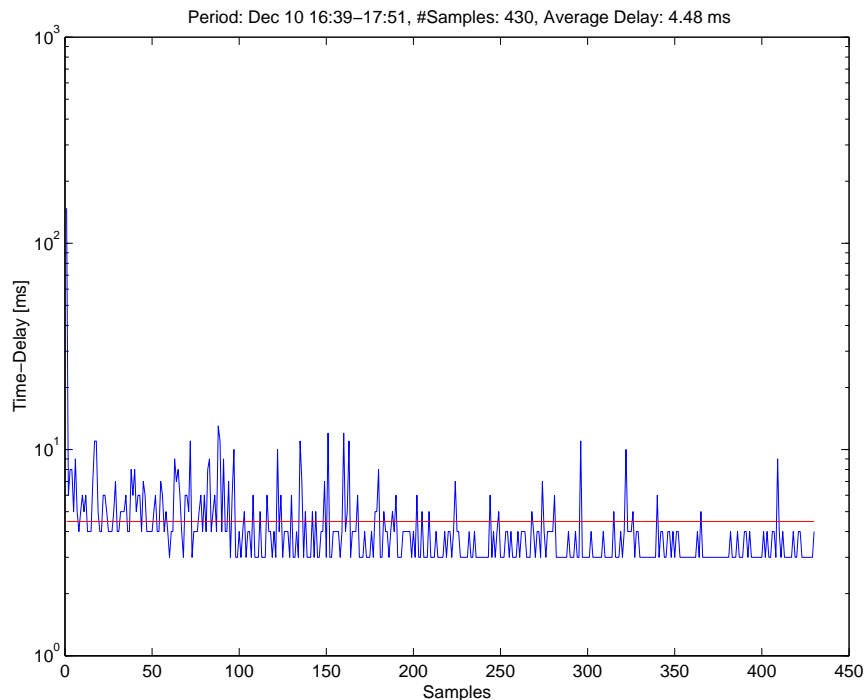


Figure 2.14: Network time-delay for 2-way UDP transmissions (2 kbytes)

The average time delay was below 6 ms what shows that the communication runs fast enough to transmit crucial data. In Figure 2.16, transmissions seem to be rather slow at around samples 1000 to 1500. That is because of the backup processes running in the background. Consequently, if performing the real experiments, heavy network traffic must not be admitted. Otherwise, communication turns out to be too slow.

Apart from some exceptional peaks, the course is very constant and jumps between two values (compare Figures 2.17 and 2.18). However, it takes some time until the speed has reached the maximum. Therefore, the more samples are considered, the lower is the average time delay.

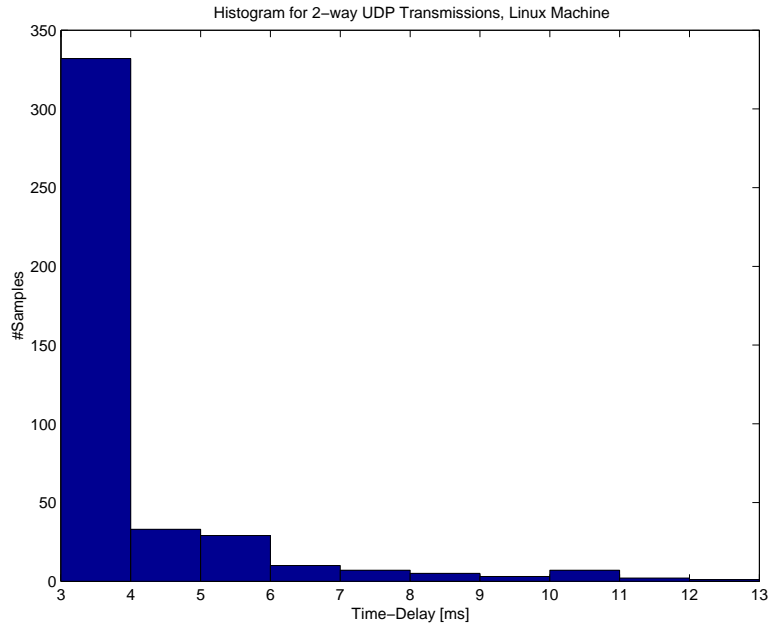


Figure 2.15: Histogram referring to diagram 2.14

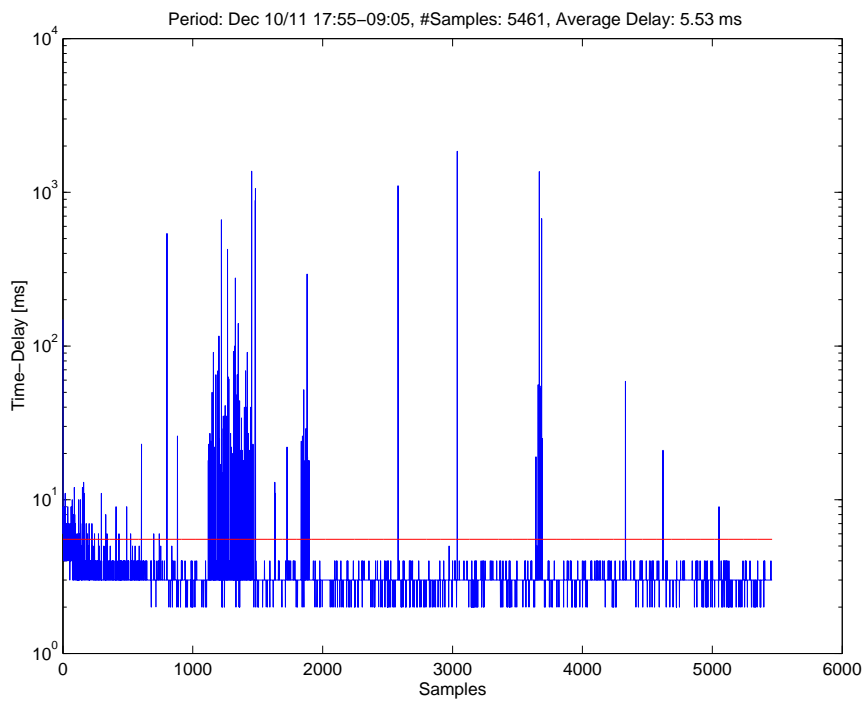


Figure 2.16: Network time-delay for 2-way UDP transmissions (2 kbytes)

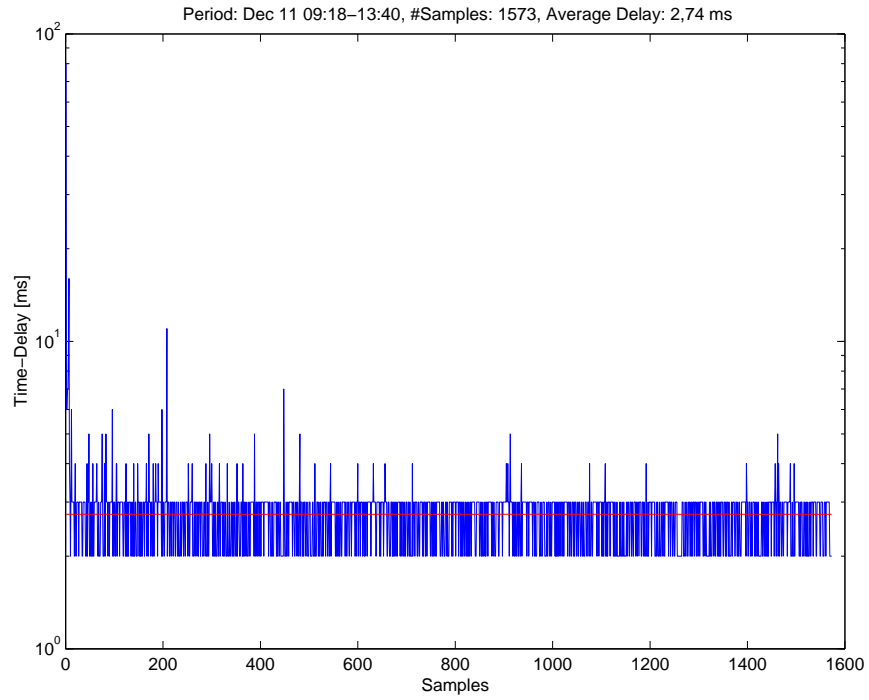


Figure 2.17: Network time-delay for 2-way UDP transmissions (2 kbytes)

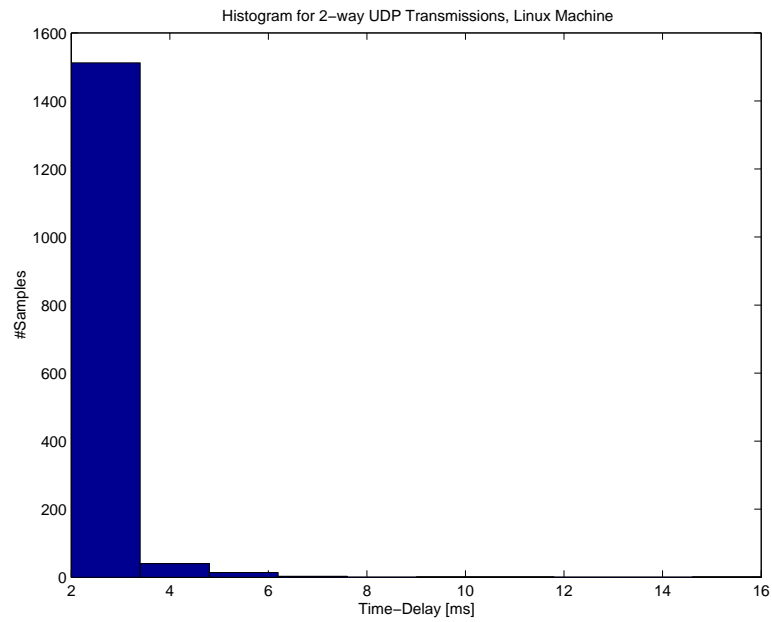


Figure 2.18: Histogram referring to diagram 2.17

Chapter 3

Computer Vision

3.1 Problem Definition

Computer vision deals with a wide range of applications and therefore, it is essential to define the kind of problem first. This having been done, suitable approaches for the image processing have to be chosen.

The aim of the ball-and-plate experiment is to make the ball roll along a predefined trajectory by tilting the plate appropriately. The position of the ball is determined by means of digital cameras and the plate is actuated by an industrial robot. Consequently, our vision problem deals with finding the ball in an image taken by one of the digital cameras.

To make this task solvable without very sophisticated vision algorithms, the ball must be easily distinguishable from the background (the plate). Moreover, visible shadows should be avoided. The image processing works faster if intensities are taken into account instead of colours because only one band is needed for grey-scaled images. Background and ball differ most if their intensities are 0 (black) and 255 (white) respectively. A white ball on a black plate also eliminates the shadows. Thus, this setup was chosen.

3.2 Approaches

Segmentation is an important and difficult step in many vision applications. Before tasks such as inspection or recognition can be carried out, one wants to know which are the different physical entities (e.g. objects) in the image. One would like to find out which pixels belong together. A delineation of regions should be made corresponding to object outlines or the boundaries of certain object surface patches.

3.2.1 Thresholding

Images where the different segments (e.g. objects) have clearly distinct intensities with respect to one another are the simplest to segment. In such cases, thresholding is appropriate. Thresholding amounts to somehow determining a critical intensity level and assigning pixels with intensities on either side of this level to *objects* and *background* respectively.

An image of the ball-and-plate experiment is more or less bimodal, i.e. has two well-separated peaks. In this case, the threshold can be chosen as the intensity corresponding to the minimum between the peaks in the histogram. Figure 3.1 shows a grey-scaled image taken by one of the digital cameras whereas Figure 3.2 illustrates the same image after thresholding:

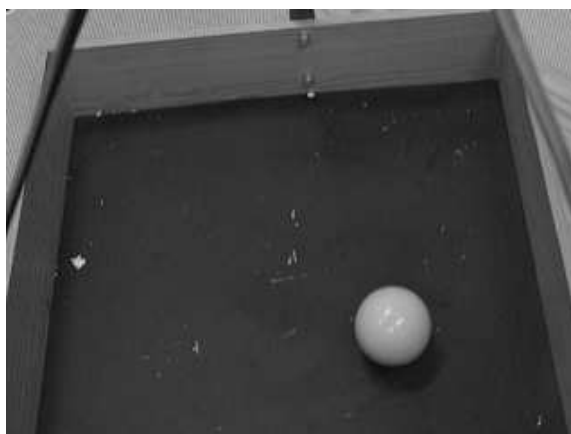


Figure 3.1: Grey-scaled camera image before thresholding operation

As can be seen, some pixels around the ball are on the wrong side of the threshold. When such pixels occur isolated or in small groups, these errors can be corrected by a combination of erosion and dilation (refer to [13]). However, since there are few such errors in comparison with the size of the ball, these methods were not implemented.

To make the image processing faster and to cut objects that are on the same side of the threshold than the ball, a region of interest has to be defined. A simple rectangle can be used to track the ball within the image. Since the position of the ball will not change significantly between subsequent images, this rectangle can be centered around the previous center of the ball. Moreover, at least one fixed point within the image must be taken into account in order to calculate the relative position of the ball's center. Figures 3.3 and 3.4 illustrate the tracking process. The

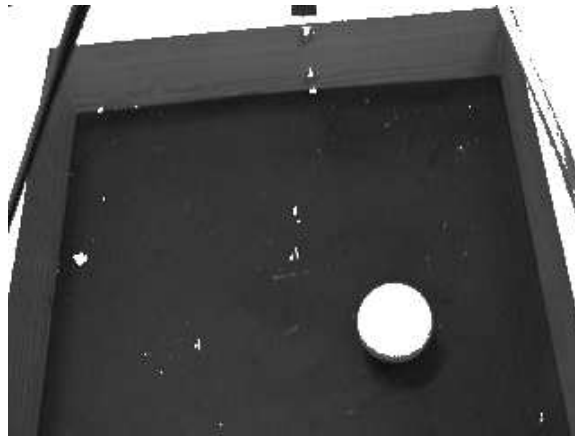


Figure 3.2: Grey-scaled camera image after thresholding operation

ball's coordinates in reference to the picture are determined by a transformation of the coordinate system. A similar transformation is used to calculate the coordinates of the fixed point with respect to the picture. Taking into account all the pixels which belong to the ball, the following equation yields its center of mass:

$$v_{cm} = \frac{1}{N} \sum_{i=1}^N \vec{v}_i$$

Finally, the ball-center in reference to the fixed point(s) is given by a vector addition.

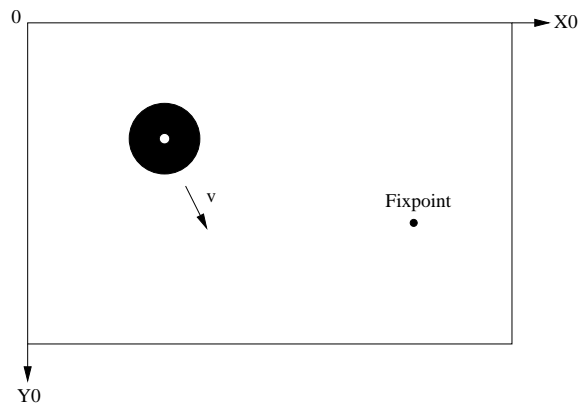


Figure 3.3: Ball- and fixed point position in frame number i

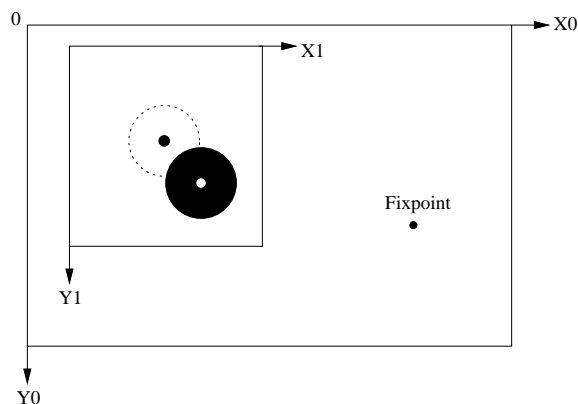


Figure 3.4: Ball- and fixed point position in frame number $i+1$: Tracking rectangle centered around the previous ball-center

3.2.2 Edge Detection

Ideally, a grey-level edge is a border between two regions, each of which has approximately uniform grey levels. Edges in images often result from occluding contours (outlines) of objects. In this case, the two regions are projections of two different surfaces. Taking a cross section of the image along a line at right angles to an edge, the resulting grey-level profile will only approximate a step discontinuity. There is a wide range of edge detection algorithms. Depending on the input picture, a specific method is chosen. However, in this thesis, we focused on thresholding rather than edge detection. Therefore, some of the algorithms are only briefly explained (for more information refer to [13] or a similar source):

- Gradient operators: With the aid of Sobel convolution masks, left and right edges can be detected. The gradient magnitude combines the outputs of the two filters. Figure 3.5 shows the result of a Sobel mask operation.
- Zero crossings: Edges are considered to lie on the inflection points of the intensity function. Such points can be found at the zero crossings of the Laplacian. The Laplacian operator can be approximated by a convolution mask. Additionally, it is usually combined with a Gaussian filter.
- Hough transform: This transformation uses the symmetry of shapes to extract them in lower dimensional spaces. For instance, circles are represented by the two coordinates of the center and the radius.

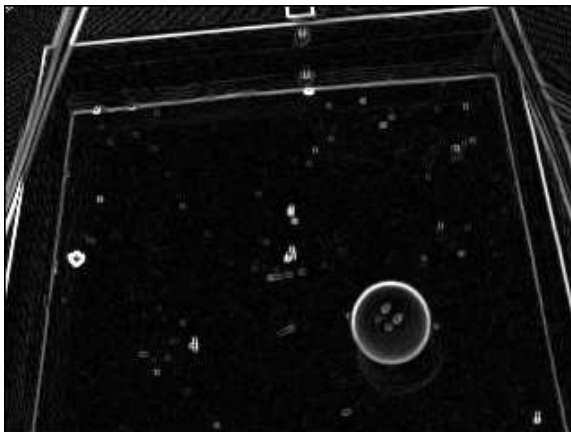


Figure 3.5: Grey-scaled camera image after Sobel edge detection

Again, a tracking rectangle as described above is needed in order to reduce the image to the region of interest. To find the center of the ball, the equation for the center of mass is no longer suitable. A circle is defined by three parameters that can be found by using a *least-squares error* method.

3.2.3 Other Approaches

A so called *line search* has been developed in an earlier project (see [7]). Based on the knowledge about the position of the ball on previous images, a line is drawn in direction where the ball is assumed to be on the current image. An exceptional change of intensity along this line indicates the edge of the ball.

There might be even more possibilities to solve our vision problem. However, the threshold method leads to satisfying results and is at the same time very fast and easy to implement. Thus, the computer vision has been done with the aid of thresholding. Further work could easily incorporate a more sophisticated approach.

3.3 Java Advanced Imaging

3.3.1 JAI API

The Java Advanced Imaging (JAI) API extends the Java platform for developing image processing applications and applets in Java. It is a set of classes providing imaging functionality beyond that of Java 2D and the Java foundation classes, though it is designed for compatibility with those APIs. The JAI API implements a set of core image processing ca-

pabilities including image tiling, regions of interest, deferred execution and a set of core image processing operators, including many common point, area, and frequency domain operators.

It turned out that simple Java based image processing methods are fast enough for real time applications (see Section 3.4). However, using cameras with higher sampling rates than 30 Hz or more sophisticated algorithms, the speed could be too low. To solve this problem, an optional C-based library (mediaLib) for Win32, Solaris and Linux provides performance enhancements for many operators. In addition there is a lower level SPARC library (VIS) and a MMX library allowing additional hardware acceleration for many operators. If Java Advanced Imaging code does not find these libraries, pure Java code is used.

The API can be extended by using the *SampleDescriptor* template. The programmer has the opportunity to add own methods or to modify existing operators. For this thesis, only predefined operators were used.

3.3.2 Implementation

The implemented package is called *RobotVision*. It provides methods to read images from a file and to write images. An image can also be built with the aid of its data buffer. In addition, different image processing functions allow to manipulate the images (convolving, thresholding, gradient). The processed pictures can also be displayed. The source code is available in Appendix E.2.

The main purpose of this package is to provide suitable methods for the ball-and-plate experiment. The grey-scaled images coming from the digital cameras need to be read and segmented using a threshold operation or edge detection principles. Afterwards, the center of mass of the ball has to be found in relation to fixed points.

The RobotVision package is also meant to be used as a template for further development. It demonstrates the functionality of the JAI API in combination with the Java core classes.

Finally, the principle of creating predefined operators with the JAI API is explained. The following code illustrates a cutout of the essential method:

```
public RenderedImage process(RenderedImage im) {
    ParameterBlock paramBlock = new ParameterBlock();
    paramBlock.addSource(im);
    paramBlock.add(parameter1);
    paramBlock.add(parameter2);
    paramBlock.add(parameter3);
    return JAI.create("operator", paramBlock);
}
```

RenderedImage is a common interface for objects which contain or can produce image data in the form of rasters. It is expressed as a collection of pixels. A pixel is defined as a 1-by-1 square; its origin is the top-left corner of the square (0,0), and its energy center is located at the center of the square (0.5,0.5). A *ParameterBlock* encapsulates all the information about sources and parameters (objects) required by a all classes that process images. For the thresholding operation, the added parameters are the upper and lower level as well as the mapping value. For gradient- and convolving operators, kernels (matrices) have to be defined for the corresponding methods. These kernels are then added to the *ParameterBlock*. The *JAI* class allows to create new images or collections by applying operators. The following operators were used: *threshold*, *gradientmagnitude* and *convolve*.

To sum up, the JAI API is a useful tool for a wide range of imaging operations. It can be extended by programmers in a simple way and provides C-based libraries for performance optimisation.

3.4 Time Testing

The sampling rate is a critical issue in real time systems. The Sony digital cameras (Appendix B) are able to take up to thirty pictures a second. However, they have to be slowed down if the subsequent image processing takes more time than the sampling interval of 33,33 ms ($= \frac{1}{30}$). Thus, the implemented code was tested on different machines.

3.4.1 Average Image-Processing Time

The experiment was performed on Sun and Linux machines. First, a colour picture of size 320*240 pixels is read in. The input image must be grey-scaled since the thresholding method works best for this case. Afterwards, the image analysis is executed and we focus on the used time for the thresholding or edge detection and the following calculation of the center of mass. A rectangle of size 100*100 pixels is used as searching region. Five different input images are processed and this procedure is repeated a hundred times. Finally, the average value of all the measurements is taken. Table 3.1 lists the results. Note that the loading takes an additional 1 to 4 ms.

Even the relatively slow Sun machines managed to perform the image processing in about half of the time-limit. Consequently, the digital cameras can sample with the maximum sampling rate (30 Hz). There seems to be a lot of freedom for enhanced computer vision algorithms.

	Sun	Linux
Convolve (Edge Detection)	16 ms	6.6 ms
Gradient (Sobel)	18 ms	12.8 ms
Threshold	15 ms	4.5 ms

Table 3.1: Average processing times for Java image analysis

3.4.2 Graphical Evaluation

To gain an impression how the average values for the processing time are accomplished, the measured values were written to a file and visualised using Matlab. Figures 3.6 to 3.9 illustrate the resulting graphs and histograms.

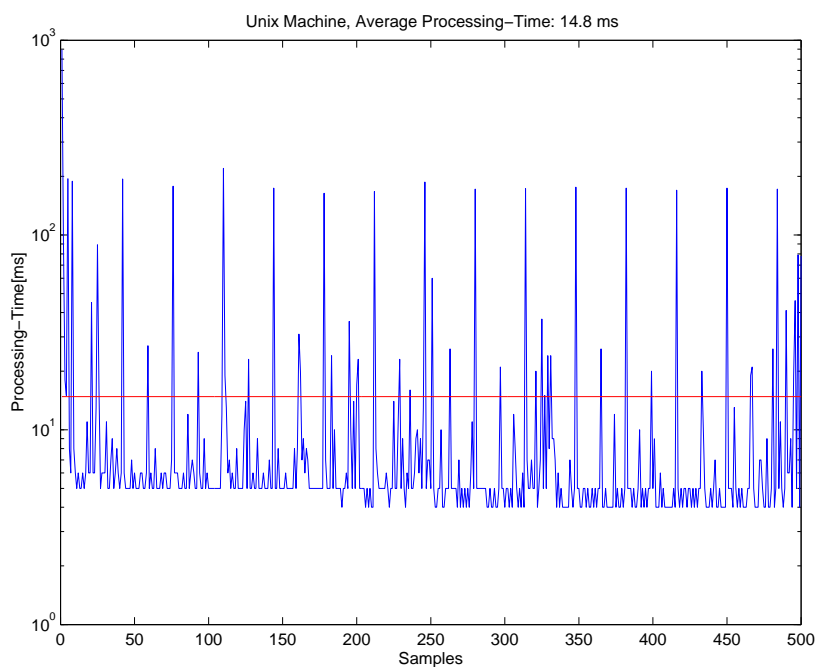


Figure 3.6: Processing time for threshold operation and image analysis on a Sun machine

Diagrams 3.6 and 3.8 show peaks that occur periodically. Using the `-verbose:gc` flag of the Java HotSpot virtual machine (VM), printouts are generated every time the Java garbage collector is called. In this way, it could be shown that the garbage collector is the reason for these peaks since it disturbs the execution of the application threads. As a consequence, objects should not be allocated too often and littering must

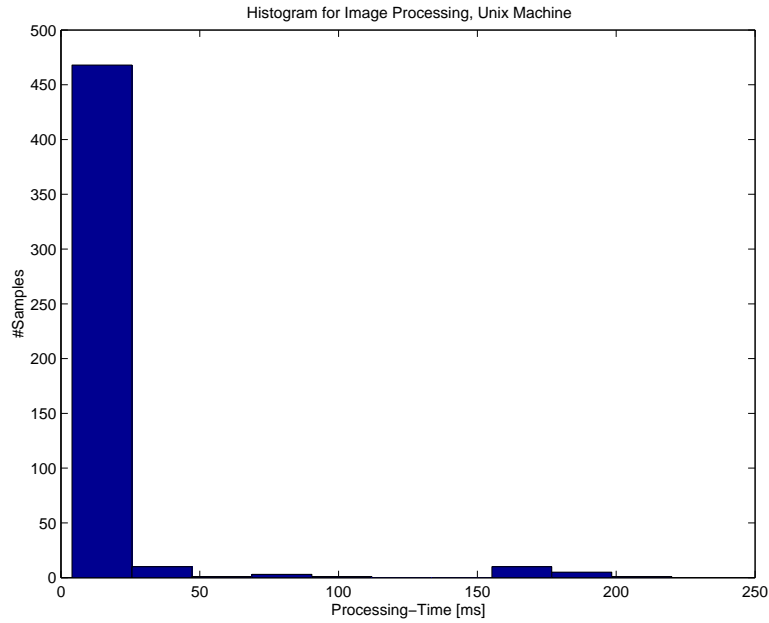


Figure 3.7: Histogram referring to diagram 3.6

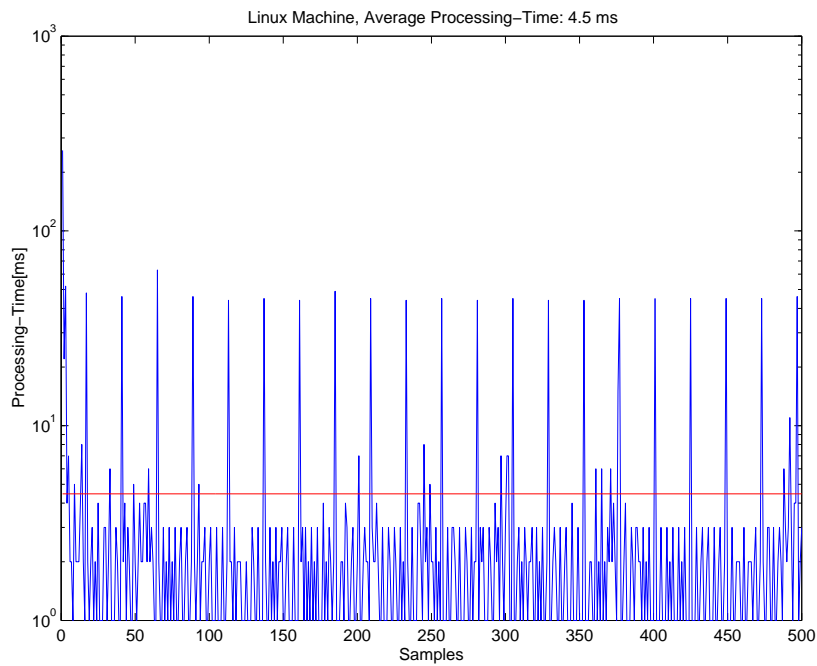


Figure 3.8: Processing time for threshold operation and image analysis on a Linux machine

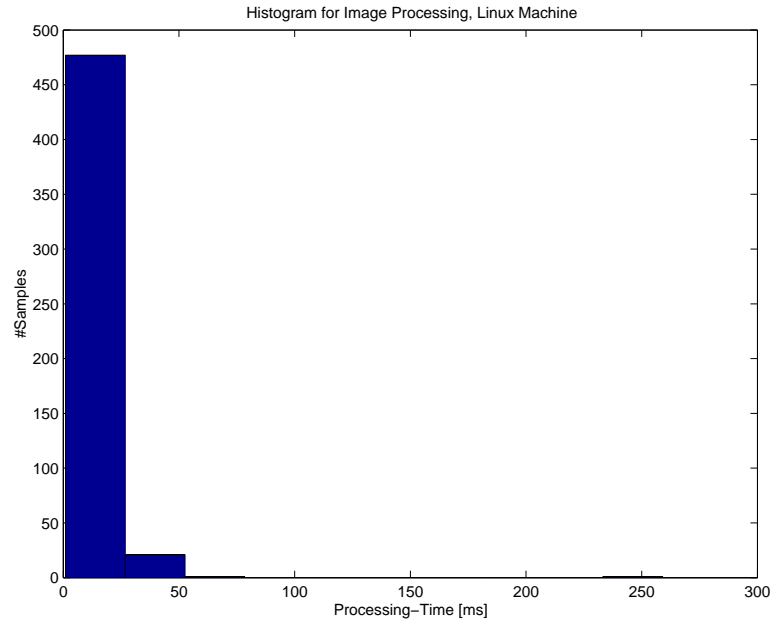


Figure 3.9: Histogram referring to diagram 3.8

be avoided.

The peaks are problematic for a control scenario. However, there are some tricks in order to make Java more suitable for real time applications and thus to yield better results:

- **Java HotSpot VM versions:** The default VM is tuned for the performance profile of client applications, on the other hand, the server VM is tuned for long-running server applications. It has a longer startup time but maximizes the operation speed. The image processing will be running on a server connected to a digital camera and thus, using the server VM improves the performance. The `-server` flag has to be set for the VM.
- **Incremental garbage collection:** A full garbage collection may potentially pause the execution for quite long times. Using the command option `-Xincgc`, the major garbage collection is instead done incrementally and so the worst-case pauses are decreased. The mentioned peaks can be avoided with the aid of incremental garbage collection.
- **Explicit garbage collection:** Using the `System.gc()` method, it is possible to explicitly invoke a major garbage collection. However, Sun recommends not using this technique.

Figures 3.10 and 3.11 show the effect of using incremental garbage collection. Note that the periodical peaks have disappeared and that the average processing-time is higher. However, the longer this experiment is performed, the closer the average value is to the previous one.

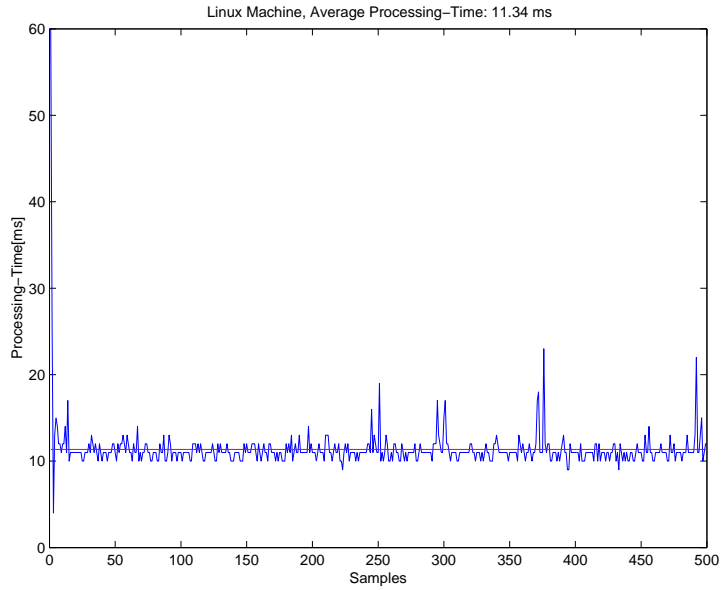


Figure 3.10: Processing time for threshold operation and image analysis on a Linux machine using incremental garbage collection

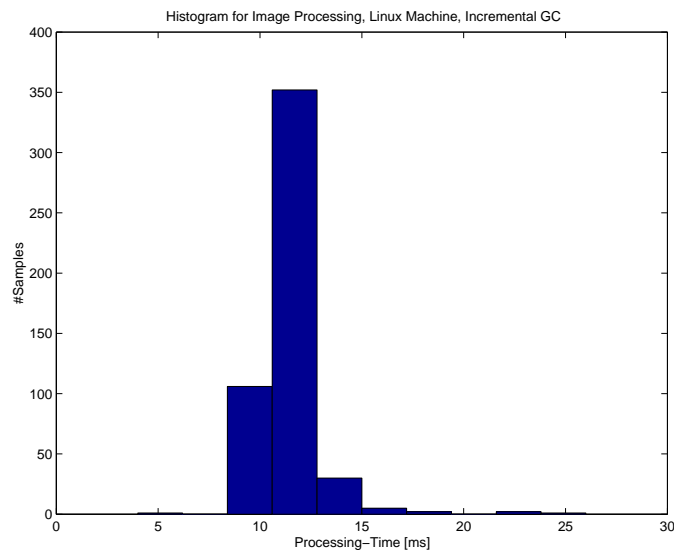


Figure 3.11: Histogram referring to diagram 3.10

Chapter 4

Interfaces

4.1 Camera Interface

4.1.1 Overview

Digital cameras are required as sensors for the feedback signals or as sensors for an input signal (user-defined trajectory). For this thesis, Sony DFW-V300 cameras [17] with a digital interface were used (Figure 4.1). They can sample with up to 30 Hz and the acquired image consists of 320*240 pixels. Additionally, the IEEE 1394 serial bus is needed for the camera output. The cameras are connected to a PC via a FireWire card [18] which features simplified cabling, hot swapping and transfer speeds up to 400 megabits per second. Unibrain [19] offers a complete development toolkit that allows developer to write professional applications for high quality 1394 digital cameras. With the aid of the Fire-i API based on Microsoft's 1394 stack, a C-application in order to initialise and start the cameras was implemented at the department. This code was used as starting point for further development. The Fire-i API is only available for Microsoft Windows and thus, the image acquisition and processing must run on a Windows machine.

4.1.2 Modification of the C-Implementation

Since the above mentioned Fire-i API is available in C only, we had to find a way in order to access the camera's image buffer from Java. The standard Java class library is not adequate for applications that are working on a hardware level and thus, the code cannot easily be translated. However, after some modifications, the methods are accessible through appropriate Java interfaces.

Firstly, the code was analysed and all the unnecessary parts like windows, statistics and image processing were removed. An additional method called *getImageDataBuffer()* used to access the camera's image



Figure 4.1: Sony DFW-V300 digital camera

buffer was implemented. Native methods are easier to handle through interfaces if they do not have C-specific types as parameters. Therefore, the modified C-implementation works with references to global variables rather than with method-parameters. Apart from the mentioned method, two others are required in order to start and terminate the camera. They are based on the Fire-i API and called *StartupCameraServer()* and *StopCameraServer()*. Appendix E.3 shows the methods of interest.

4.1.3 Java Native Interfaces

The Java Native Interface (JNI) [14, 15] is the native programming interface for Java that is part of the JDK. By writing programs using the JNI, it is ensured that the code is completely portable across all platforms. Native methods and the JNI are used in the following situations:

- The standard Java class library may not support the platform-dependent features needed by an application.
- There may exist already a library or application written in another programming language and it should be accessible to Java applications.
- A small portion of time-critical code needs to be implemented in a lower-level programming language, such as assembly. The Java application calls those functions.

JNI serves as the glue between Java and native applications. Diagram 4.2 shows how the JNI ties the C side of an application to the Java side.

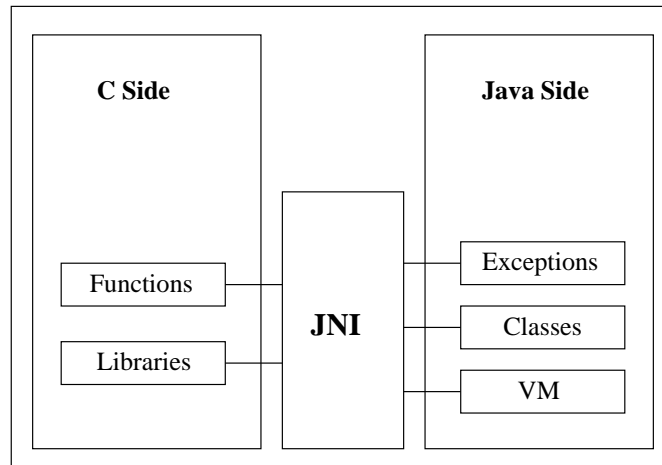


Figure 4.2: Java Native Interface (JNI)

The API for the digital camera is available in C only. In order to initialise the camera as well as to access its memory, a Java Native Interface is required. The Java class containing the main method must declare and call all the native methods of interest. To become the formal signature for the native methods, a header file must be generated using the *javah -jni* command (see Appendix E.3). Finally, the native language code is compiled into a library. On Solaris and Linux, a shared library is created, while on Windows it is called a dynamic link library (DLL). The Java application loads the appropriate library and is enabled to call the previously declared native methods.

4.1.4 Combination of the Subsystems

Since it is now possible to access the image buffer from the digital camera, we can combine it with the previously implemented packages LabComm and RobotVision (see Chapters 2 and 3). The RobotVision package provides methods to build an image from a buffer and to process it whereas LabComm offers methods that allow sending the data over a network via TCP or UDP. Hence, the implementation of the package *VisionServer* was rather straightforward. Its main method calls functions to initialise and start the camera, to access the image data, to perform the image processing and to send the calculated coordinates of the center of mass to another network node (controller). Figure 4.3 illustrates how the subsystems work together.

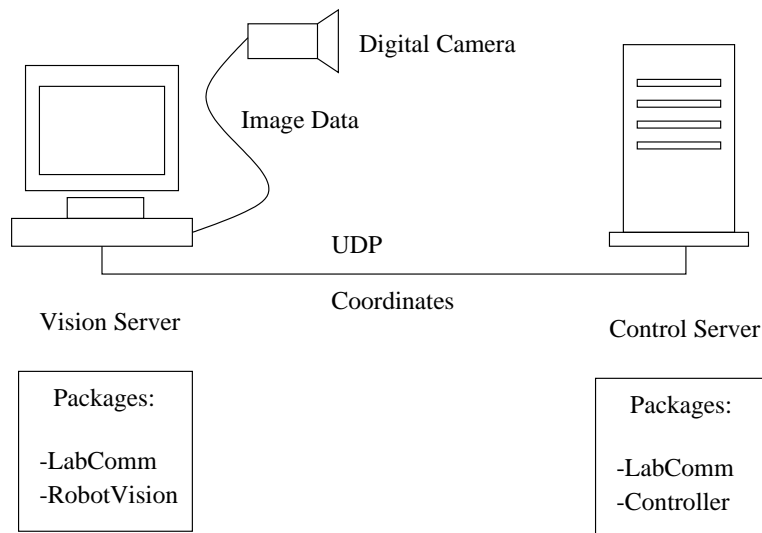


Figure 4.3: Combination of the implemented subsystems

4.1.5 Timing Tests

All the subsystems have already been tested regarding the occurring time delays. However, it was interesting to investigate the total delay in order to find out if the implementation is suitable for a real time system. As explained above, Windows machines had to be used as vision servers.

It turned out that the sampling rate has to be decreased to 15 Hz if the slower machine (450 MHz) is chosen. However, the faster Windows machine (1800 MHz) managed to process all within 33 ms, i.e. the maximum sampling rate (30 Hz) of the digital cameras can be applied. UDP allows to send and receive small data packages with a frequency of up to 40 Hz. Thus if the network is not overloaded, the transmission speed should be adequate to obtain an input- or feedback signal for the controller synchronised with up to 30 Hz.

4.2 Matlab Java Interface

Every installation of Matlab includes a Java Virtual Machine (VM). Therefore, the Java interpreter can be used via Matlab commands. It is possible to build and run programs that create and access Java objects (for detailed information about the interface refer to [16]). The Matlab Java interface enables to:

- Access Java API class packages that support essential activities such as I/O and networking

- Access third-party Java classes
- Construct Java objects in Matlab
- Call Java object methods, using either Java or Matlab syntax
- Pass data between MATLAB variables and Java objects

In order to use the LabComm communication software (see Section 2.4) in combination with Matlab, an interface like the one described above is required. All third-party classes have to be added to the Matlab *classpath.txt* file. Packages can be imported to Matlab in the same way as to Java. Finally, Java objects can be constructed and their public methods and fields are accessible from the Matlab workspace. To sum up, the Java main method for UDP or TCP nodes needs to be implemented in Matlab. E.5 illustrates the resulting script.

As can be seen, Java and Matlab combine well. Having a Java class, it is rather simple to include it into the Matlab environment.

Chapter 5

Multi-Camera Robot Control

5.1 Ball-and-Plate Process

5.1.1 Overview

In order to demonstrate and test the implementations, a suitable experiment had to be chosen. We decided to perform the ball-and-plate experiment which is based on the one dimensional ball-and-beam process. The experiment can be executed with the aid of a robot and reference- as well as feedback signals are relatively simple to acquire with digital cameras. The object is to balance a ball on a flat plate by tilting the plate to move the ball. The plate is attached to the gripper of an industrial robot which tilts it using two joints (two degrees of freedom). The ball does neither take off nor slip and it rolls in the appropriate direction under the force of gravity.

5.1.2 Block Diagrams

A cascaded control system will be used to control the ball-and-plate apparatus. On the one hand, the ball position on the plate has to be controlled and on the other hand, a robot motion controller is required for the angle control of the plate. Finally, the vision system tracks the current position of interest and processes an appropriate signal for the controller. Figure 5.1 illustrates the cascaded closed loop system.

The vision system (Diagram 5.2) includes a digital camera (refer to Section 4.1.1) and the image processing (refer to Chapter 3). Note that both the feedback- and the reference signal can be generated in the same way using the vision system.

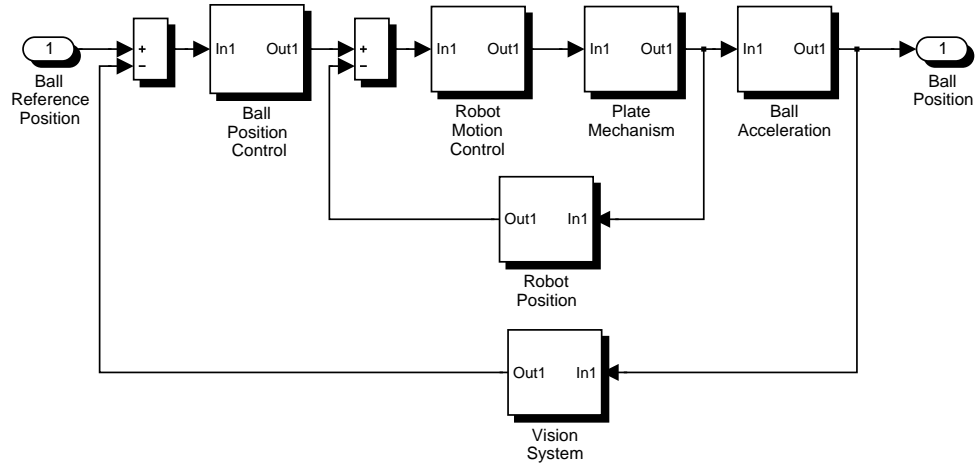


Figure 5.1: Cascaded closed loop control system

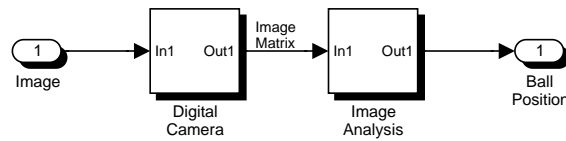


Figure 5.2: The vision system: image as the input and ball position as the output

5.2 Model of the Plant

In order to obtain some fundamental information about the behaviour of the ball-and-plate system, the simpler case of a ball-and-beam system is introduced first. While the ball has two degrees of freedom on a plate, it has only one on a beam.

5.2.1 Ball-and-Beam System Model

The dynamics for the ball position can be determined by classical mechanics. Assume that the ball position is x , the beam angle is ϕ and that the beam lies in the center of rotation ϕ . Coordinates and forces are defined according to Figure 5.3.

The force equation gives

$$\begin{aligned} m\ddot{y} &= -mg + N \cos \phi + F \sin \phi \\ m\ddot{z} &= -N \sin \phi + F \cos \phi. \end{aligned}$$

Multiplying with $\sin \phi$ and $\cos \phi$ and summing up results in

$$m(\ddot{y} \sin \phi + \ddot{z} \cos \phi) = -mg \sin \phi + F. \quad (5.1)$$

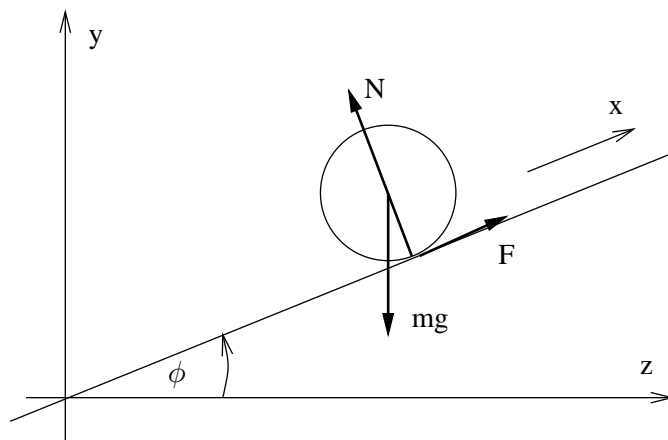


Figure 5.3: The ball-and-beam system

The condition for a rolling ball without friction is $Fr = J\dot{\omega} = -J\ddot{x}/r$. For a solid ball, we have $J = 2mr^2/5$, hence $F = -2m\ddot{x}/5$. Furthermore, $z = x \cos \phi$ and $y = x \sin \phi$. Deriving this twice, multiplying with $\cos \phi$ and $\sin \phi$, summing and we get

$$\ddot{y} \sin \phi + \ddot{z} \cos \phi = \ddot{x} - x\dot{\phi}^2.$$

If we insert this expression and the one for F in (5.1), we get

$$m(\ddot{x} - x\dot{\phi}^2) = -mg \sin \phi - \frac{2}{5}m\ddot{x}.$$

If we finally assume that ϕ and $\dot{\phi}$ are small, we obtain

$$G(s) = -\frac{5g}{7s^2} \approx -\frac{7}{s^2}.$$

As can be seen, for sufficiently small radii, angles, angular velocities and angular accelerations, the plant is in principle a double-integrator.

5.2.2 Ball-and-Plate System Models

The models of the ball and plate system which are deeply discussed in [7] are not derived in this thesis. Two models are distinguished, a full model and a reduced one. The equations of motion for the two degrees of freedom are coupled and not mathematically equivalent. The reason for this lies in the set-up of the experiment depicted in Figure 5.4. While the first rotational axis (the x-axis) is motionless, the second one (the v-axis) is rotated around the first one. However, if these equations are linearised, they turn out to be decoupled and equal to the equations of motion for the ball-and-beam system.

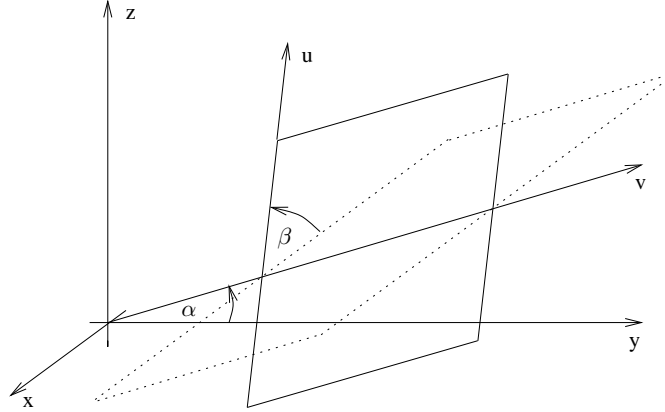


Figure 5.4: The plate and its descriptive coordinates

The linearised equations of motions for the full model of the ball-and-plate system are

$$\begin{aligned}\ddot{u} &= -\frac{5g}{7}\beta + r\ddot{\beta} \\ \ddot{v} &= -\frac{5g}{7}\alpha + r\ddot{\alpha}\end{aligned}$$

where r is the radius of the ball. Since the ball is required to remain constantly in contact with the plate, centrifugal forces normal to the plate are of no interest. Thus, the distance from the origin to the ball's center of mass can safely be approximated by the distance from the origin to the point of contact of the ball and the plate. As a further simplification, the coupling between the three components of the angular velocity of the ball is disregarded. This is equivalent to neglecting the spin normal to the plate. The linearised equations of motion for the reduced model of the ball-and-plate system are

$$\ddot{u} = -\frac{5g}{7}\beta \approx -\frac{1}{7}\beta \quad (5.2)$$

$$\ddot{v} = -\frac{5g}{7}\alpha \approx -\frac{1}{7}\alpha. \quad (5.3)$$

Equations (5.2) and (5.3) showing that the process is basically a double-integrator for both dimensions are used for the development of the controller.

5.3 Design of the Ball-Position Controller

Because of the duality between the two differential equations describing the linearised ball-and-plate system, the motion in each dimension can

be controlled separately. Finally, two controllers are applied to the ball-and-plate system.

5.3.1 Time Response of the Plant

Before the controller can be designed and simulated, we need to understand the behaviour of the linearised plant. Since the system is non-linear, the time response is taken into account. The simulation tool *Simulink* is very helpful for this purpose. Figure 5.5 shows a Simulink representation of the linearised and simplified plant. Since the inclination angle α is differentiated, we cannot excite the system with an ordinary step function. A smoothed step (see Figure 5.6) is taken instead where T is the rise time. Diagram 5.7 illustrates the step response of the linearised plant.

$$u(t) = \begin{cases} 0 & (t \leq 0) \\ \frac{1}{2}(1 - \cos \frac{\pi}{T}t) & (0 \leq t \leq T) \\ 1 & (t \geq T) \end{cases}$$

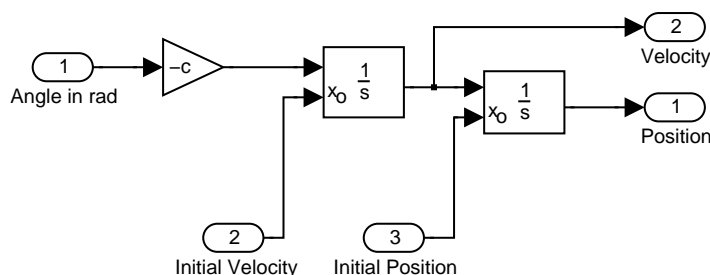


Figure 5.5: The linearised and simplified plant

5.3.2 Continuous-Time Controller

The controller will be implemented in Java and thus, it must be digital. We can either derive a digital controller from a continuous one by emulation (e.g. Euler's method) or directly develop a digital design based on the discretised model of the plant. Emulation design yields reasonable results at sample rates on the order 20 times the bandwidth of the plant. The process was defined in Equations (5.2) and (5.3) and has a bandwidth of about 3 rad/s. The sample rate is limited to 30 Hz (188.5 rad/s). As a consequence, the controller can be derived from a continuous design.

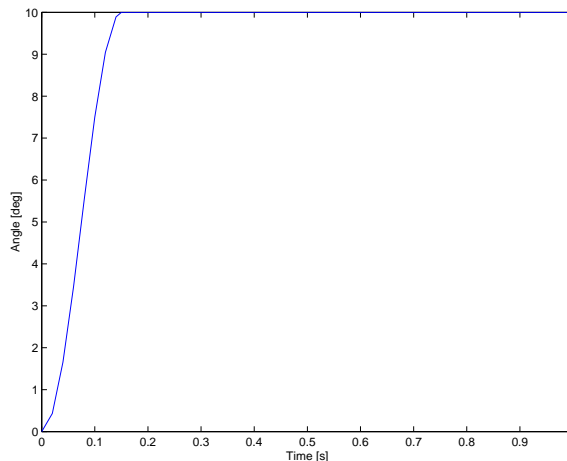


Figure 5.6: Smoothed step

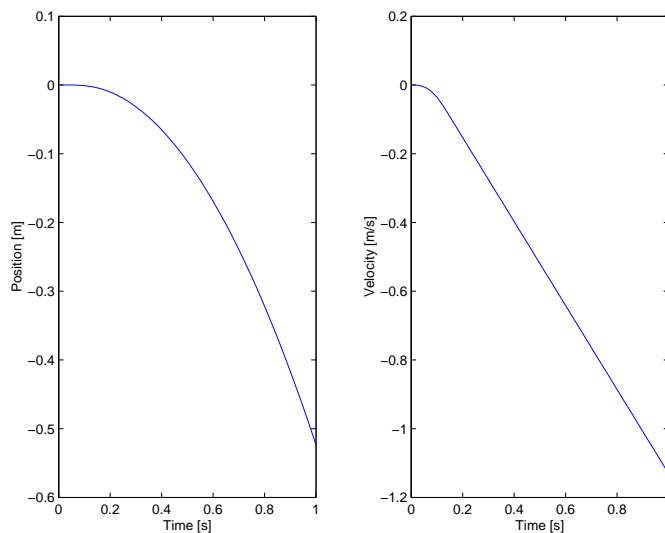


Figure 5.7: Step response of the linearised and simplified system

PD Controller

As a first step, a standard PD controller is considered. To make the controller practically useful, some modifications are needed. We do not want to apply derivation to high frequency measurement noise, therefore the following modification is used:

$$sT_D \approx \frac{sT_D}{1 + sT_D/N}$$

N is the maximum derivative gain, often between 10 and 20. Furthermore, since the set-point does often change in steps, the D-part becomes

very large. Derivative weighting helps to avoid this phenomenon.

$$D(s) = T_D s (\gamma Y_{sp}(s) - Y(s))$$

In our design, γ is set to zero like it is usually done in process control.

Figure 5.8 shows the process together with a PD controller. The two poles of the closed-loop system can be chosen arbitrarily by selecting the proper values of K and T_D .

$$G(s) = \frac{-cK(sT_D + 1)}{s^2 - sT_DcK - cK} \quad (5.4)$$

If we specify the characteristic polynomial as

$$p(s) = s^2 + 2\delta\omega_n s + \omega_n^2$$

and if we compare it to the denominator of Equation (5.4), the parameters of the controller are found:

$$K = -\frac{\omega_n^2}{c}$$

$$T_D = \frac{2\delta}{\omega_n}$$

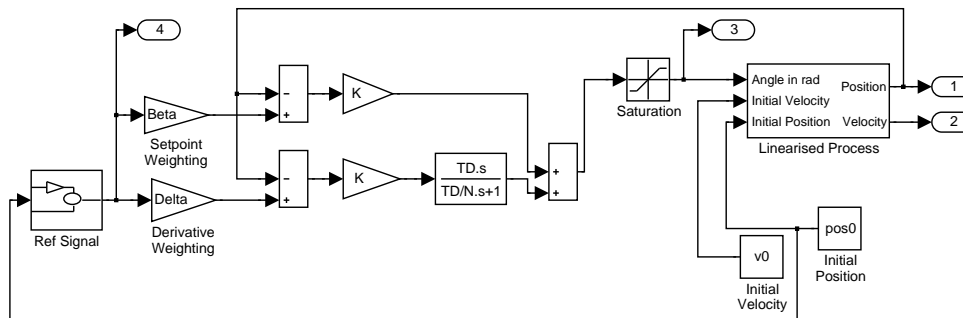


Figure 5.8: The linearised process with a PD controller

Figures 5.9 and 5.10 show the input- and output signals of the regulated plant for different damping ratios and cut-off frequencies.

It turned out that the real process can be controlled using a PD controller. The parameters were set according to the slow controller design (compare Figure 5.10 and Section 5.6).

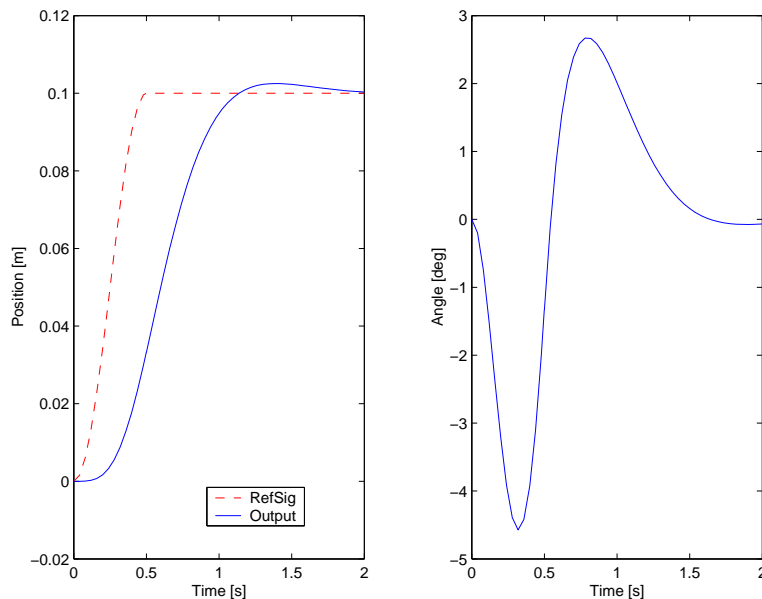


Figure 5.9: Input and output for the PD regulated plant, cut-off frequency $\omega_n = 4$ rad/s and damping ratio $\delta = 0.7$

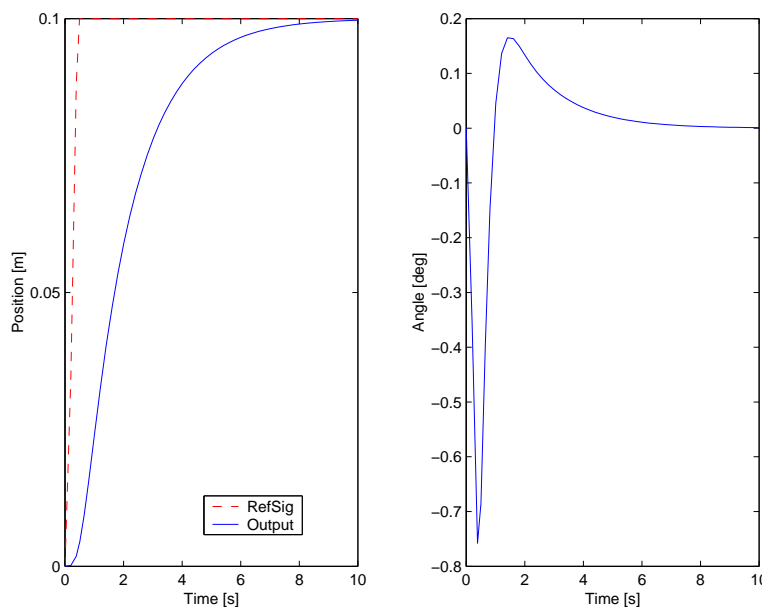


Figure 5.10: Input and output for the PD regulated plant, cut-off frequency $\omega_n = 1.15$ rad/s and damping ratio $\delta = 1.1$

PID Controller

In order to eliminate the stationary error, an integral part is required. In this section, a PID controller is designed and simulated. As described

above, the derivative gain is limited and derivative weighting is used. Additionally, anti-windup prevents the integral part from integrating up to very large values in case of control signal saturation. Anti-windup is implemented by using a back-calculation: when the control signal saturates, the integral is recomputed so that its new value gives a control signal at the saturation limit. The re-computation is done dynamically through a LP-filter with time constant T_r .

Figure 5.11 shows the process together with a PID controller. The three poles of the closed-loop system can be chosen arbitrarily by selecting the proper values of K , T_D and T_I .

$$G(s) = \frac{-cK(s^2T_D + s + \frac{1}{T_I})}{s^3 - s^2T_DcK - scK - \frac{cK}{T_I}} \quad (5.5)$$

If we specify the characteristic polynomial as

$$p(s) = (s + \alpha\omega_n)(s^2 + 2\delta\omega_n s + \omega_n^2)$$

and if we compare it to the denominator of Equation (5.5), the parameters of the controller are found:

$$K = -\frac{\omega_n^2(1 + 2\alpha\gamma)}{c}$$

$$T_D = \frac{\alpha + 2\gamma}{\omega_n(1 + 2\alpha\gamma)}$$

$$T_I = \frac{1 + 2\alpha\gamma}{\alpha\omega_n}$$

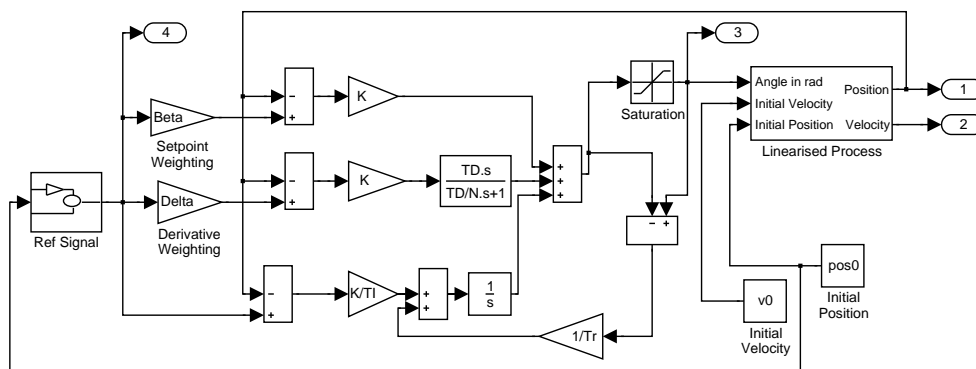


Figure 5.11: The linearised process with a PID controller

Figures 5.12 and 5.13 show the input- and output signals of the regulated plant for different cut-off frequencies.

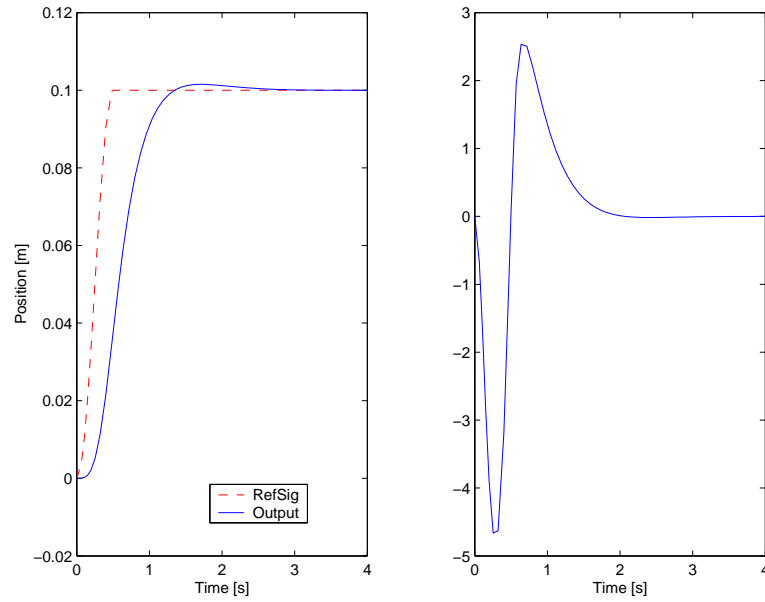


Figure 5.12: Input and output for the PID regulated plant, cut-off frequency $\omega_n = 4$ rad/s, damping ratio $\delta = 0.9$ and $\alpha = 1$

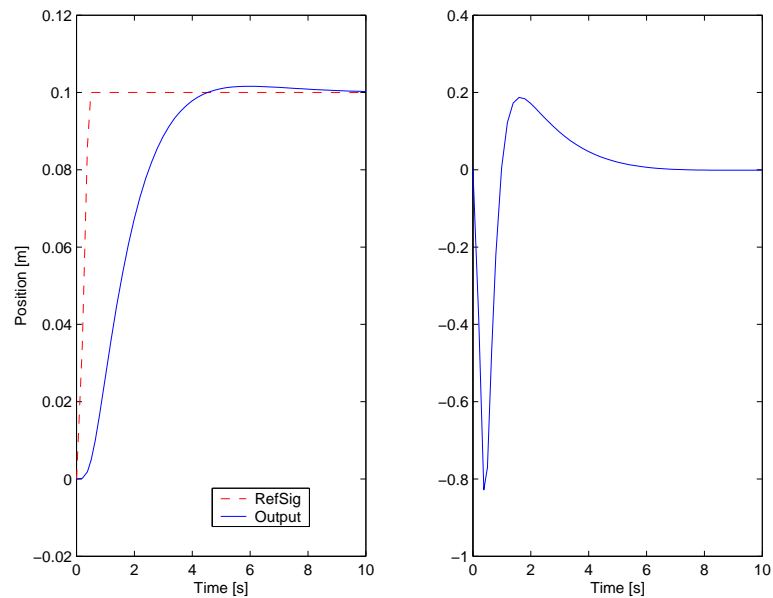


Figure 5.13: Input and output for the PID regulated plant, cut-off frequency $\omega_n = 1$ rad/s, damping ratio $\delta = 0.9$ and $\alpha = 1$

It turned out that the real process is very difficult to control using a PID controller. Since the process is a double-integrator, it is very sen-

sitive to the I-part of the controller. We could not stabilise the process with a PID controller.

5.3.3 Java Implementation of the Controller

Since the communication tool as well as the computer vision were implemented in Java, we decided to develop the controller in Java as well. The continuous representation of a PID controller has to be discretised for this purpose. In the following, the chosen discretisation is shown.

- P-part:

$$u_p(k) = K(y_{sp}(k) - y(k))$$

- I-part: Forward difference is used

$$\frac{I(t_{k+1}) - I(t_k)}{h} = \frac{K}{T_I} e(t_k)$$

- D-part: Backward difference is used

$$D(t_k) = \frac{T_D}{T_D + Nh} D(t_{k-1}) - \frac{KT_D N}{T_D + Nh} (y(t_k) - y(t_{k-1}))$$

Assuming that the saturation values for the control signals are u_{max} and u_{min} , the subsequent equations describe the PID controller including anti-windup.

$$\begin{aligned} u &= P + I + D \\ y &= \text{sat}(u, u_{max}, u_{min}) \\ I &= I + \frac{Kh}{T_I} e + \frac{h}{T_r} (u - y) \end{aligned}$$

The Java based controller receives feedback- and reference data, visualises these data and calculates the control signal which is finally sent to the actuator (the robot). The parameters can be changed at runtime via a user interface. Appendix E.4 shows the source code for the controller.

5.4 The Robot System

5.4.1 System Overview

For experimental purposes, the plate for the rolling ball was screwed to the gripper of an industrial robot with six degrees of freedom like the one in Figure 5.14.

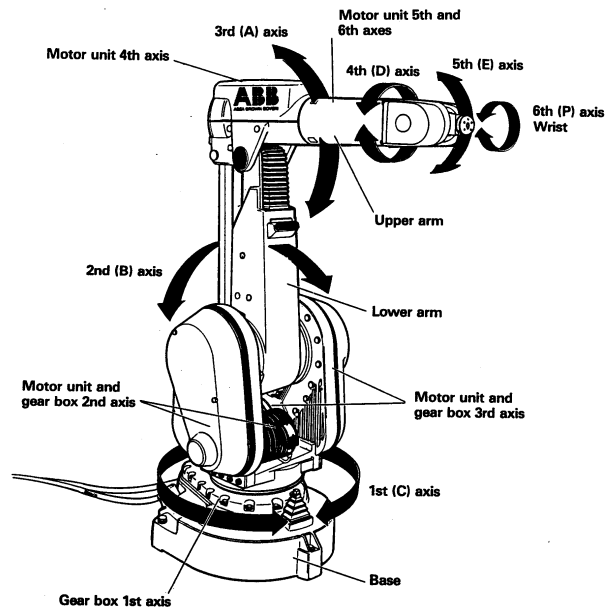


Figure 5.14: The industrial robot that serves as an actuator

Only two joints are used since the ball is rolling in two dimensions only. The maximal rotational speed is $300^\circ/s$ and the sampling frequency is up to 4 kHz. The robot is much faster than the dynamics of the ball-and-plate system and thus, the parameters of the PID controllers for the robot motion could be tuned independently of the control design for the trajectory tracking of the ball. The existing robot software [23] makes it possible to link user-specific controllers to the system.

5.4.2 Motion Control

The robot's motion control consists of two cascaded PID controllers (see Figure 5.15). Position control for each joint was realised with an inner velocity control. The reference signal sent to the robot must contain the absolute rotational angle for the joints and optionally the estimated angle velocity of the joints. The robot has only resolvers for measuring the positions and no tachometers for measuring the velocities, thus the angle velocity is the position differentiated.

5.4.3 Robot Communication

In order to send control data to the robot, MatComm (refer to Section 2.1) is required. The simplest way to use MatComm is in combination with Matlab. Consequently, the control signals have to be sent to Matlab

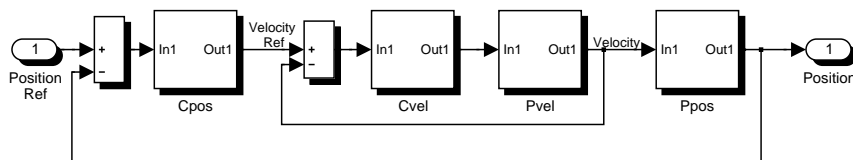


Figure 5.15: Robot motion control with two cascaded PID controllers

first where trajectories are built and these trajectories are finally sent to the robot server via MatComm. In Section 4.2, it is described how Java classes can be called from Matlab. Based on such a Matlab-Java interface, a script was developed that handles both the UDP communication between Java nodes and the TCP communication between Matlab and the robot server (see Appendix E.5). MatComm provides a function in order to send control angles to the robot. However, this function turned out to be too slow for a data-transmission speed required in combination with 30 Hz sampling frequency. Trajectories were generated and sent instead. They consist of a time stamp as well as position- and velocity references for each joint. Within the sampling interval, the position values are interpolated between a start- and end value. The end position of trajectory n is the start position for trajectory $n + 1$. The robot is able to follow trajectories that were sent with a frequency of 30 Hz. That means that the sampling rate of the digital cameras can be utilised.

5.5 Robot Experiments

In order to perform the experiments with the robot, the plate and one digital camera had to be attached to the gripper first (compare Figure 5.16). If the robot is positioned appropriately, the two-dimensional movement of the plate can be executed using two joints (joint 4 and joint 6, compare Figure 5.14).

The controller was designed for an input signal with unit meter. Since we control the process in the image, we need to transform from pixel to meter. This can be done by measuring a corresponding distance both on the plate and in the image. For the chosen setup, one pixel corresponds to approximately 1.1 mm.

5.5.1 Experiment Based on Visual Feedback

The experimental setup is represented in Figure 5.17 and contains a robot together with the plate, one digital camera, the robot server and workstations for the different tasks. Three different communication-types are used: FireWire for the camera, MatComm for the data trans-



Figure 5.16: The plate and digital camera attached to the robot

mission to the robot server and LabComm for the rest. This experiment allows to test all parts of this thesis: the LabComm communication tool, the image processing and camera interface as well as the controller.

The feedback signal acquired with the digital camera is sent to a Windows machine where the image is processed (refer to Chapter 3). The coordinates of the center of mass of the ball are transmitted to the controller running on a Linux machine. Moreover, a computer-generated reference signal is also sent to the controller which calculates the control signals for the robot. This data is transferred to Matlab running on a Sun computer and from there sent to the robot server via MatComm.

The sampling frequency is limited to 30 Hz. However, the Windows machine used in this experiment is too slow to handle 30 Hz and thus, 15 Hz was used instead. A PD controller is responsible for the controlling of the ball. The results are shown in Section 5.6.

5.5.2 Experiment Based on Two Digital Cameras

This experiment is similar to the one previously described, but the reference signal is acquired with a digital camera. A white object is slowly moved on a black background and the robot has to tilt the plate in a way such as the ball follows this reference. Figure 5.18 illustrates the setup. The results can be found in Section 5.6.

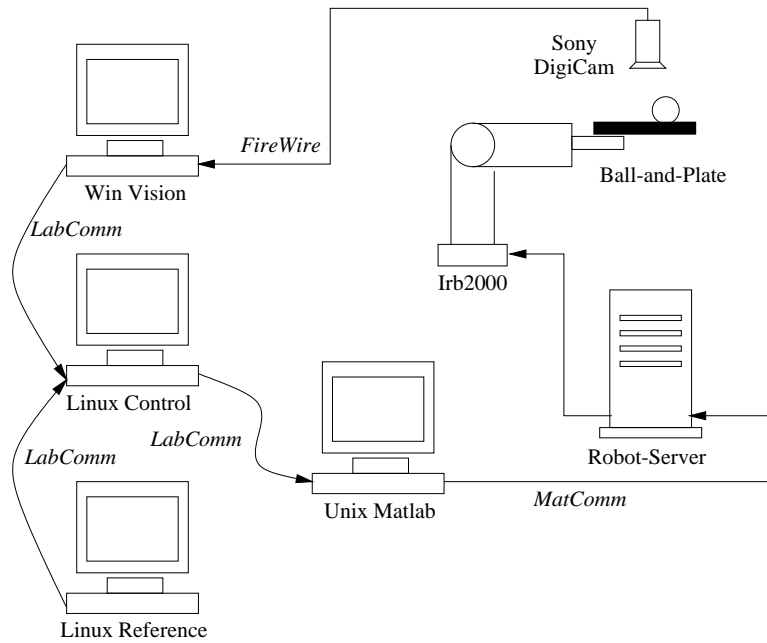


Figure 5.17: Robot experiment based on visual feedback

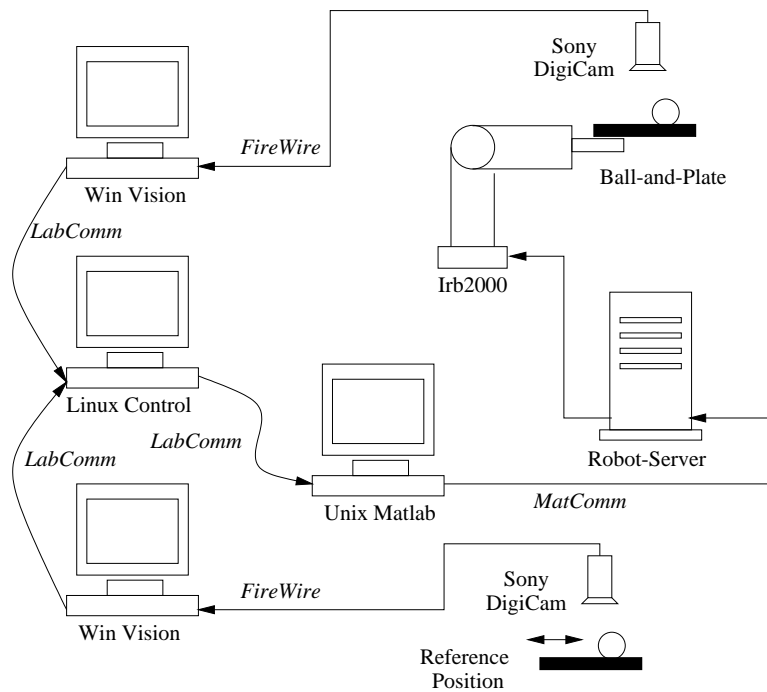


Figure 5.18: Robot experiment based on two digital cameras

5.5.3 Experiment Based on Two Robots

Since there is only one robot available in the laboratory, this experiment could not be performed. However, its idea is described in this section. Basically, the same setup than in the experiment discussed in Section 5.5.2 is used but this time, a robot generates the reference signal by tilting a second plate appropriately. The other robot needs to be controlled in a way such as the ball follows the trajectory defined by the reference robot.

5.5.4 Problems and Possible Solutions

Several problems had to be solved in order to run the experiments. If similar experiments are performed in the future, a lot of time can be saved by considering this enumeration.

- **Trajectory handling:**
If trajectories have to be sent with a sampling frequency higher than 5 Hz, an increasing delay regarding the robot's movement can be observed. Without any precautions, this delay can reach up to a second after only a few seconds of execution which makes the controlling impossible. To solve this problem, the time intervals of the trajectories have to be somewhat shorter than the interval of the sending procedure. If for instance trajectories are sent every 100 ms, they should not last longer than 80 ms. See also Appendix E.5.
- **Robot gripper:**
The plate has to be attached to the gripper which is equipped with a safety-system based on compressed air. If the forces exceed a certain limit, the air flows out and the system breaks down. However, since the plate is rather heavy, the gripper turned out to be too weak for fast movements. It has to be blocked by putting wood or rubber between the two parts of the gripper.
- **Ball-and-plate apparatus:**
In order to obtain a sufficiently large image, the camera has to be installed not closer than 50 cm to the plate. The existing apparatus is somewhat flexible which leads to oscillations in the position measurement. This can be avoided by either installing the camera at a fixed point or by rebuilding the apparatus.
- **Controller:**
As mentioned in Section 5.3.2, we could not stabilise the plant with a PID controller. The PD controller worked satisfactorily although it could possibly be improved by adding for instance feed-forward.

5.6 Results

In order to visualise the controlled process, the signals had to be logged. The controller receives the feedback- and the reference signal and writes these data to a file. Furthermore, the control signal is logged in Matlab and compared to the real position of the joints ¹. Figures 5.19 to 5.22 illustrate the results for the experiments described in Section 5.5.1 and 5.5.2. The parameters for the PD controller were calculated with Matlab and manually optimised. Table 5.1 shows the final settings.

	Joint 4	Joint 6
K	0.16	-0.18
T_D	2.2	1.9
N	5	10

Table 5.1: Experimental parameters for the PD controller

The reason why the parameters for the two robot joints are different lies in the setup of the system. The positive x-axis of the image corresponds to a positive angle for joint 4 whereas the positive y-axis corresponds to a negative angle for joint 6 ². Moreover, whereas the y-axis of the plate is identical to the rotational axis of joint 6, the x-axis is translated with respect to the rotational axis of joint 4. The unmodeled dynamics in joint 4 could be one reason why the results for joint 4 show ringing effects (see Figures 5.19 and 5.20). Moreover, the oscillations could also be related to an unevenness of the plate and eventually be decreased by replacing the wooden plate by a glass disc. Note that cancellations of zeros were not considered in the design.

Although the control signal depicted in Figure 5.21 looks rather violent, the robot has no problems to follow it (Diagram 5.22). The delay between the control signal and the joint position does not increase with time and is smaller than the sampling interval.

¹MatComm provides a function to read the robot's current joint angles.

²Equations (5.2) and (5.3) are based on the setup illustrated in Figure 5.4.

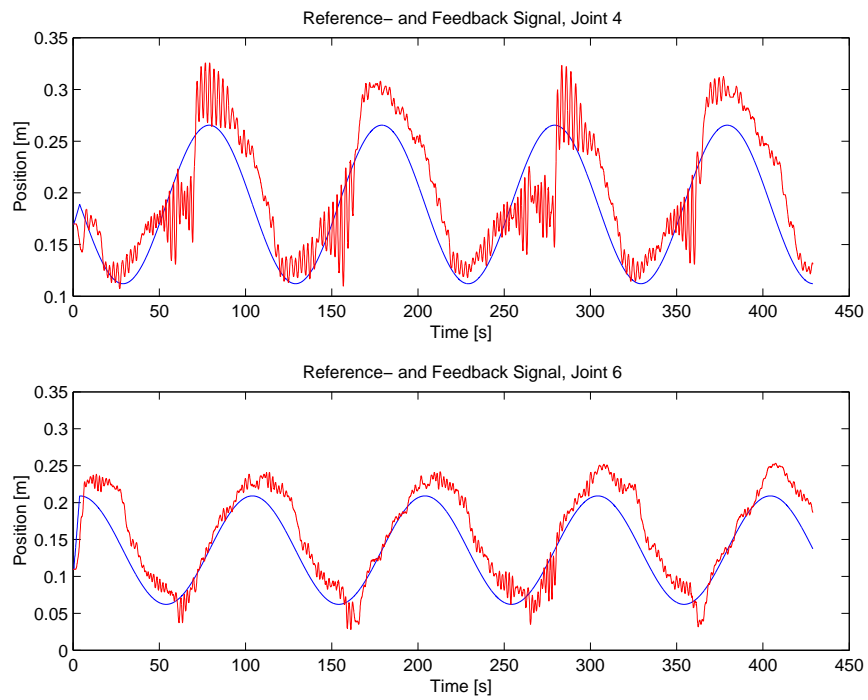


Figure 5.19: Reference- and feedback signal, experiment with one camera

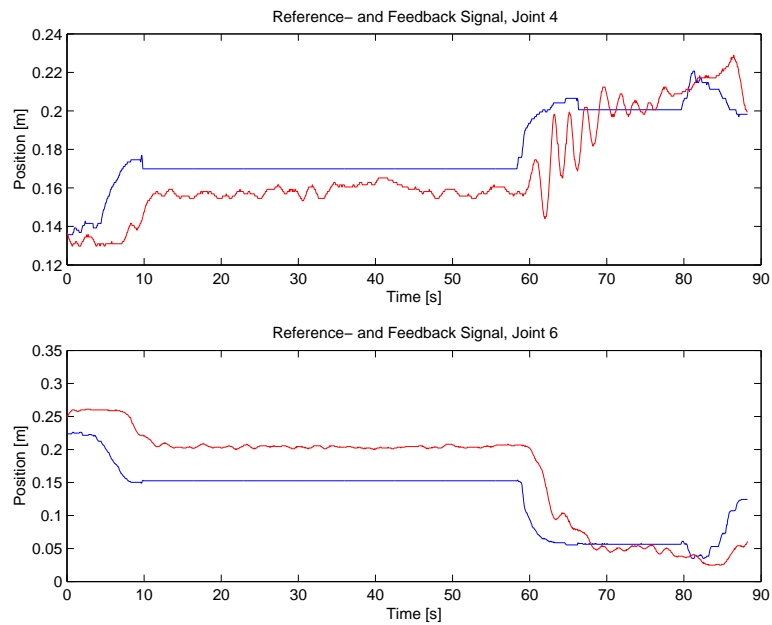


Figure 5.20: Reference- and feedback signal, experiment with two cameras

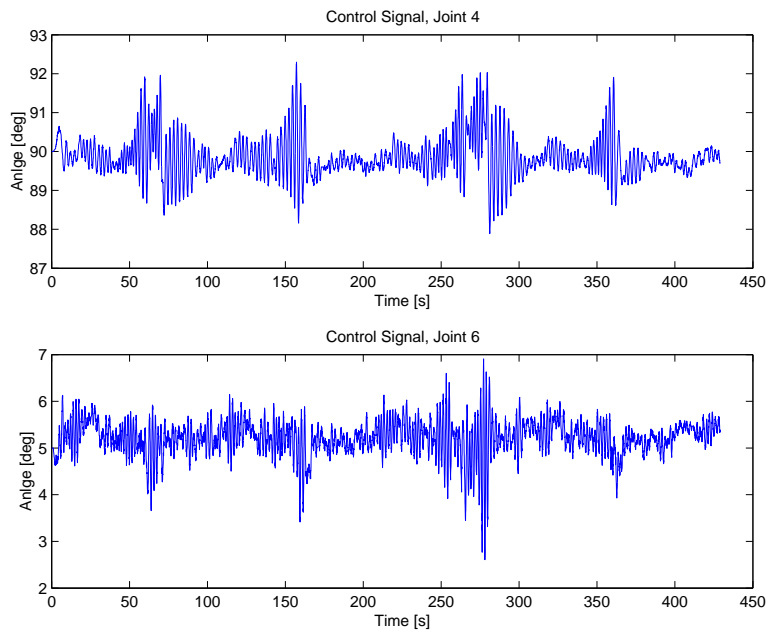


Figure 5.21: Control signals for joint 4 and 6 referred to Figure 5.19

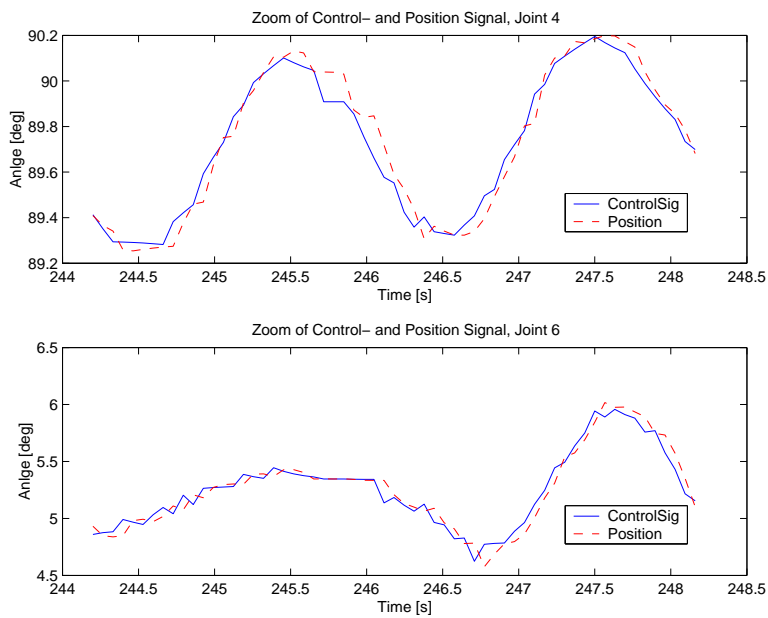


Figure 5.22: Comparison of control- and position signal (extraction of Figure 5.21)

Chapter 6

Conclusions

6.1 Summary

This thesis consists of three main parts. In the following, the results of each part are summarised.

- **Communication tool:**

The communication tool LabComm is based on MatComm which was developed at the Department of Automatic Control of LTH Lund and which uses TCP as underlying network protocol. LabComm provides the features of MatComm, but it is more powerful since it also includes the User Datagram Protocol (UDP) what makes it more convenient for distributed systems. UDP makes communication faster and more predictable than TCP.

In addition, the Java concept of serialisation was applied. As a consequence, LabComm allows to send and receive more sophisticated data like structs. Other software packages than Matlab are now able to use it as a communication medium. Finally, the nameserver was also implemented in Java. There is no need for a default nameserver running on a specific host anymore.

All in all, LabComm is platform independent, easily extendible and provides the classes for rapid data communication in a distributed real time environment.

- **Image processing, interfaces:**

The digital cameras are initialised and accessed with the aid of a C-based API. Instead of rewriting the existing code, the concept of Java Native Interfaces, which makes native methods accessible, was applied. The developed vision package provides functions for Java-based camera control.

The Java Advanced Imaging API is useful in order to implement image operators in a relatively simple way. It turned out that operations like thresholding or edge detection are executed in a reasonable time with respect to a digital camera providing 30 frames per second.

Because controllers are often implemented in Matlab Simulink, an interface which allows the combination of Java and Matlab is required. Matlab is equipped with a Java virtual machine and hence can make use of Java classes. The developed Matlab script shows how data sent via LabComm is received and used in Matlab.

- Control, experiment:
The ball-and-plate process was chosen as a demonstrator for both the computer vision and the communication tool. An industrial robot serves as an actuator by tilting the plate in a way such as the ball follows a predefined trajectory.

A PD- as well as a PID controller were simulated with Simulink but it turned out that the process is very sensitive with respect to the integral part. However, the plate can be controlled with the aid of a PD controller.

Two robot experiments were successfully performed, one using a computer generated reference signal and visual feedback and the other making use of two digital cameras. Several workstations handled the different tasks (distributed system) and the data was transmitted with the aid of LabComm.

6.2 Outlook

This thesis can be used as a basis for further work. Some possible improvements are outlined in the following.

- Communication:
So far, only basic functions like sending and receiving data are available. More functionality could be integrated, for example a graphical user interface or methods to observe the transmissions. Additionally, the robot server could be modified so that it would be able to use LabComm.
- Image processing:
Extended algorithms would enable a more general setup of the ball-and-plate process. Additionally, other processes could also be taken into account, for instance object grasping in three dimensions using stereo vision.

- **Control:**
With the aid of the virtual robot, it should be possible to tune the PID controller in order to stabilise the process. Better results are expected with more advanced control design techniques. Discrete time designs and non-linear dynamics could also be taken into account.
- **Experiments:**
Providing six degrees of freedom, the robot can serve as an actuator for a wide range of additional experiments. The controller and computer vision would have to be adapted whereas the communication tool could be used without modifications.

Bibliography

- [1] Automatic Control Homepage:
<http://www.control.lth.se/>
- [2] LTH Homepage:
<http://www.lth.se/english/default.asp>
- [3] A. Cervin and D. Henriksson and B. Lincoln and K. Årzén. *Analysis and Simulation of Controller Timing*. IEEE Control Systems Magazine, 2003.
- [4] J. Bengtsson and M. Haage and R. Johansson. *Variable Time Delays in Visual Servoing and Task Execution Control*. Lund Institute of Technology, 2002.
- [5] T. Olsson and J. Bengtsson and R. Johansson and H. Malm. *Force Control and Visual Servoing Using Planar Surface Identification*. Lund Institute of Technology, 2002.
- [6] E. Hashimoto. *Visual Control of Robots: High Performance Visual Servoing*. World Scientific, 1993.
- [7] Martin Rentsch. *Controlling a Ball and Plate Process Using Computer Vision*. Lund Institute of Technology, 1998.
- [8] Sun's Java Homepage:
<http://java.sun.com>
- [9] Java API Specification:
<http://java.sun.com/j2se/1.4.1/docs/api/>
- [10] D. Flanagan. *Java in a Nutshell*. O'Reilly, 1999.
- [11] E. Harold. *Java Network Programming*. O'Reilly, 1997.
- [12] Java Object Serialisation Specification:
<http://java.sun.com/j2se/1.4.1/docs/guide/serialization/spec/serialTOC.doc.html>

- [13] L. Van Gool. *Computer Vision and Visual Inspection*. ESAT, ETH, 2001.
- [14] Java Native Interface Specifications:
<http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html>
- [15] Java Native Interface Tutorial:
<http://java.sun.com/docs/books/tutorial/native1.1/index.html>
- [16] Matlab External Interfaces:
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/ch_java2.shtml
- [17] Sony DFW-V300 Digital Camera:
<http://www.scimeasure.com/Sony/Camera%20%5BIndustrial%5D/dfw-v300.htm>
- [18] FireWire:
<http://developer.apple.com/firewire/>
- [19] Unibrain Fire-iAPI:
http://www.unibrain.com/1394_products/fireiapi/fire-iapi.htm
- [20] RedHat Linux:
<http://www.redhat.com/>
- [21] Java Advanced Images Package:
<http://java.sun.com/products/java-media/jai/>
- [22] Visual C++ Homepage:
<http://msdn.microsoft.com/visualc/>
- [23] K. Nilsson. *Industrial Robot Programming*. Lund Institute of Technology, 1996.
- [24] Mathworks Homepage:
<http://www.mathworks.com/>
- [25] Simulink Homepage:
<http://www.mathworks.com/products/simulink/>
- [26] LaTeX Typesetting System:
<http://www.latex-project.org/>
- [27] Emacs Editor:
<http://www.gnu.org/software/emacs/emacs.html>
- [28] Xfig Drawing Tools:
<http://www.xfig.org/>

Appendix A

Used Software Tools

Operating Systems

SunOS 5.8

RedHat Linux 7.3 [20]

Windows NT4, 2000

Java Software

Java 2 SDK 1.4.1 [8]

Java Advanced Imaging 1.1.2-beta [21]

C++ Software

Visual C++ [22]

Robot Software

Open Robot Control System [23]

Applications

Matlab 6.1 [24]

Simulink 4.1 [25]

L^AT_EX 2_ε [26]

GNU Emacs 21.4 [27]

Xfig 3.2 [28]

Appendix B

Used Hardware

Workstations

- Unix Sun Ultra 10, Ultra 60
- PC Pentium P4 2400 MHz, 512 MB (Linux),
PC Pentium P4 1800 MHz, 512 MB (Win2000),
PC Pentium P3 450 MHz, 256 MB (WinNT)

Digital Cameras

Sony DFW-V300 colour digital cameras, 30 Hz [17]

FireWire

Standard FireWire cards [18]

Robot

ABB industrial robot 2000 (Irb2000)

Appendix C

Analysis of Serialised Streams

This appendix shows how Java's build-in data-types as well as objects like arrays are represented in an output stream. In addition, the resulting differences between the serialisable and the externalisable interface are discussed. The generated output files are binary and thus are not readable directly. With the aid of *hexdump* or a similar tool, it is possible to make them legible.

Let us start with the simplest example. A class called *struct1* only consisting of the 32 bit integer *8* and a version unique identifier is written to an output stream, using the serialisable interface. Here is what comes out:

```
00000000 aced 0005 7372 0007 7374 7275 6374 3100 ....sr..struct1.
00000010 0000 0000 0000 ff02 0001 4900 0169 7870 .....I..ixp
00000020 0000 0008
```

The first line contains the class identity. In the second line, *ff* is the version UID whereas *49* identifies the data type (integer). Finally, the four bytes of data (number 8) are found in the third line.

If the externalisable interface is used instead of the serialisable one, the class identity remains unchanged. However, there is no information about the data type available. The data itself is terminated by the number *78* indicating the end of the block-data:

```
00000000 aced 0005 7372 0007 7374 7275 6374 3200 ....sr..struct2.
00000010 0000 0000 0000 ff0c 0000 7870 7704 0000 .....xpw...
00000020 0008 78 ..x
```

Similar outputs result for the other build-in data types like double, float, character or string.

In a next step, a one-dimensional array of integers is used instead of a single integer. Now, more information is written to the stream since

the array needs to be represented as well. The resulting output looks like that:

```
00000000 aced 0005 7372 0007 7374 7275 6374 3100 ....sr..struct1.
00000010 0000 0000 0000 ff02 0001 5b00 0469 6172 .....[.iar
00000020 7274 0002 5b49 7870 7572 0002 5b49 4dba rt..[Ixpur..[IM.
00000030 6026 76ea b2a5 0200 0078 7000 0000 0200 `&v.....xp.....
00000040 0000 0500 0000 08 .....

```

The first line remains the same whereas *5b* in the second line stands for a */* indicating the array. In the third and fourth line, there is the information about the array of integers while the data (number 5 and 8) is found in the last line.

This time, the externalisable interface leads to a considerably shorter output. A lot of type information can be avoided. Again, the data is found in the last line terminated by *78*:

```
00000000 aced 0005 7372 0007 7374 7275 6374 3200 ....sr..struct2.
00000010 0000 0000 0000 ff0c 0000 7870 7572 0002 .....xpur..
00000020 5b49 4dba 6026 76ea b2a5 0200 0078 7000 [IM.`&v.....xp.
00000030 0000 0200 0000 0500 0000 0878 .....x

```

Let us consider a two-dimensional array of integers. The array written as Matlab matrix looks like that: `[[5 8][7 9]]`. The data can easily be identified. There is only some additional information put in between the two one-dimensional arrays. As expected, the matrix itself requires more bytes for its representation:

```
00000000 aced 0005 7372 0007 7374 7275 6374 3100 ....sr..struct1.
00000010 0000 0000 0000 ff02 0001 5b00 0769 6d61 .....[.ima
00000020 7472 6978 7400 035b 5b49 7870 7572 0003 trixt..[[Ixpur..
00000030 5b5b 4917 f7e4 4f19 8f89 3c02 0000 7870 [[I...O...<...xp
00000040 0000 0002 7572 0002 5b49 4dba 6026 76ea ....ur..[IM.`&v.
00000050 b2a5 0200 0078 7000 0000 0200 0000 0500 .....xp.....
00000060 0000 0875 7100 7e00 0500 0000 0200 0000 ...uq.~.....
00000070 0700 0000 09 .....

```

Again, using the externalisable interface instead of the serialisable one results in a more compact output:

```
00000000 aced 0005 7372 0007 7374 7275 6374 3200 ....sr..struct2.
00000010 0000 0000 0000 ff0c 0000 7870 7572 0003 .....xpur..
00000020 5b5b 4917 f7e4 4f19 8f89 3c02 0000 7870 [[I...O...<...xp
00000030 0000 0002 7572 0002 5b49 4dba 6026 76ea ....ur..[IM.`&v.
00000040 b2a5 0200 0078 7000 0000 0200 0000 0500 .....xp.....
00000050 0000 0875 7100 7e00 0400 0000 0200 0000 ...uq.~.....
00000060 0700 0000 0978 .....x

```

Finally, it is investigated if a stream has to be reset or not. For this purpose, a class like described above is written twice to the same

output stream; first without resetting the stream, then by resetting it before the class is written to the stream again. In addition, the data (integer 8) is changed (to integer 9) before the rewriting process starts. Here are the results for the serialisable interface:

```
00000000 aced 0005 7372 0007 7374 7275 6374 3100 ....sr..struct1.
00000010 0000 0000 0000 ff02 0001 4900 0169 7870 .....I..ixp
00000020 0000 0008 7100 7e00 01 .....q.~..

00000000 aced 0005 7372 0007 7374 7275 6374 3100 ....sr..struct1.
00000010 0000 0000 0000 ff02 0001 4900 0169 7870 .....I..ixp
00000020 0000 0008 7973 7200 0773 7472 7563 7431 ....ysr..struct1
00000030 0000 0000 0000 00ff 0200 0149 0001 6978 .....I..ix
00000040 7000 0000 09 .....p....
```

As can be seen, it does not work without resetting the stream. In the first sample, there is only some information added saying that the class is written again. However, the changed data can not be seen. In the second sample, 79 in the third line indicates the *reset* operation. The entire class representation follows including the new data.

As a consequence, the *reset* operation must be performed if a class is written to a stream more than once, especially if the data is updated.

Appendix D

Short User Manual

This appendix explains briefly how the implemented software packages are organised and what has to be considered. It also provides a guideline for the experiments.

D.1 General Information

First of all, a Java 2 Standard Edition including the runtime environment must be installed. Moreover, the Java Advanced Imaging API needs to be added in order to run the computer vision classes. The versions should not be older than the ones mentioned in Appendix A.

To run the Matlab scripts *matcommserver* and *labcomminterface*, the paths of the jar files containing the *MatlabServer*- and *LabComm* classes have to be added to the Matlab *classpath.txt*.

The packages consist of the following files and folders:

- Makefile:
Compiling and running the code
- Doc:
HTML-based Java documentation
- Src:
Source code
- Compiled:
Compiled source code (Java classes)
- Jar:
Classes packaged with the ZIP file format
- Addum:
Additional files and data

To compile the entire package, type *make all*. The main classes of a package can be executed by typing *make runXY* where *XY* stands for a specific process. If new Java classes are added to the package, the makefile has to be modified. The make-commands work under Unix, Windows and Linux.

The documentation lists all classes belonging to a packet including fields and methods. It should be considered in order to become quickly acquainted with the code. The start page is called *overview-summary.html*.

The packages *LabComm*, *RobotVision* and *Controller* contain the basic classes for communication, computer vision and control. The first two provide several demos that can be started using the *makefile*. Since the packages *ControlServer*, *ReferenceServer* and *VisionServer* are based on the basic packages, they need to be updated as soon as basic classes were modified. The appropriate jar files have to be copied to the jar folder for this purpose.

D.2 Running the Experiments

Refer to Section 5.5 in order to have an overview of the experiments. The processes can either be started via terminal or via ssh. For the first experiment (see Section 5.5.1), the following packages are required: *LabComm*, *ControlServer1Cam*, *VisionServer*, *ReferenceServer* and the Matlab script *matcommserver*. Start the nameserver on the desired host by executing *make runnameserver*. The makefile is located in the *LabComm* package. Start the control server on another host by typing *make runcontrol*. Start the vision server (*make runvision1*) as well as the reference server (*make runref1*). Execute the Matlab script and check if all hosts are registered to the nameserver. Finally, follow the instructions according to the program outputs.

For the second experiment (see Section 5.5.2), the following packages are required: *LabComm*, *ControlServer2Cam*, *VisionServer* and the Matlab script *matcommserver*. Start the nameserver on the desired host by executing *make runnameserver*. The makefile is located in the *LabComm* package. Start the control server on another host by typing *make runcontrol*. Start the vision servers (*make runvision1* and *make runvision2*). The vision servers should preferably run on two different machines since the image processing is potentially time consuming. Execute the Matlab script and check if all hosts are registered to the nameserver. Finally, follow the instructions according to the program outputs.

Appendix E

Source Code Extract

E.1 LabComm

TCPClient.java

```
package se.lth.control.tcp;
import se.lth.control.nameserver.*;
import se.lth.control.labmatrix.*;
import se.lth.control.labcomm.*;
import java.net.*;
import java.io.*;

/**
 * This class provides functions to connect (disconnect) to a TCPserver.
 * In addition, data can be sent and received.
 * @version
 * @author
 */

public class TCPClient {

    /**
     * Creates a TCPClient (Default-Constructor).
     * @throws UnknownHostException
     */
```

```
    public TCPClient() throws UnknownHostException {
        nameServer = new StorkClient();
    }

    /**
     * Creates a TCPClient that uses a specific nameserver.
     * @param nameserver_host The hostname of the custom nameserver
     * @param nameserver_port The portnumber of the custom nameserver
     * @throws UnknownHostException
     */
    public TCPClient(String nameserver_host, int nameserver_port) throws UnknownHostException {
        nameServer = new StorkClient(nameserver_host, nameserver_port);
    }

    /**
     * Allows acces to the private socket.
     * @return The TCP socket
     */
    public Socket getSocket() {
        return connection;
    }

    /**
     * Allows acces to the private StorkClient.
     * @return The TCP socket
     */
    public StorkClient getStorkClient() {
        return nameServer;
    }

    /**
     * Creates a TCP connection to the desired server.
     * @param servername The name of connected server
     * @throws IOException,UnknownHostException
     */
    public void connect(String servername) throws IOException, UnknownHostException {
        StorkNameEntryList entries = nslookup(servername);
        if (entries.getSize() > 0) {
            StorkNameEntry entry = entries.get(0);
            connection = connect2server(entry);
            return;
        }
        throw new UnknownHostException();
    }

    /**
     * Closes a TCP connection.
     * @throws IOException
     */
```

```

    public void disconnect() throws IOException {
        connection.close();
    }

    /**
     * Packs a LabMatrix into a MatrixPacket and sends it
     * using a TCP socket. Writes the matrix
     * via externalizable interface to the stream.
     * @param matrix A LabMatrix of type double, char, float, int or String
     * @throws IOException
     */
    public void sendMatrix1(LabMatrix matrix) throws IOException {
        MatrixPacket p = new MatrixPacket(matrix);
        OutputStream outputStream = new BufferedOutputStream(connection.getOutputStream());
        ObjectOutputStream s = new ObjectOutputStream(outputStream);
        s.writeObject(p);
        s.flush();
    }

    /**
     * Packs a LabMatrix into a MatrixPacket and sends it
     * using a TCP socket. Writes the matrix via a loop to the stream.
     * @param matrix A LabMatrix of type double, char, float, int or String
     * @throws IOException
     */
    public void sendMatrix2(LabMatrix matrix) throws IOException {
        MatrixPacket packet = new MatrixPacket(matrix);
        DataOutputStream outputStream = new DataOutputStream(new BufferedOutputStream(
            connection.getOutputStream()));
        packet.toStream(outputStream);
        outputStream.flush();
    }

    /**
     * Writes the LabObject via externalizable interface to the stream.
     * @param obj A LabObject (LabMatrix or LabStruct)
     * @throws IOException
     */
    public void writeLabObject(LabObject obj) throws IOException {
        OutputStream outputStream = new BufferedOutputStream(connection.getOutputStream());
        ObjectOutputStream s = new ObjectOutputStream(outputStream);
        MatrixPacket p = null;
        if (obj instanceof LabMatrix) {
            p = new MatrixPacket((LabMatrix)obj);
            s.writeObject(p);
            s.flush();
            s.reset();
        }
        else {
            s.writeObject(obj);
        }
    }

```

```

        s.flush();
        s.reset();
    }

    /**
     * Receives a MatrixPacket over a TCP connection. Reads the matrix
     * via externalizable interface from the stream.
     * @return A MatCommMatrix of type double, char, float, int or String
     * @throws IOException, ClassNotFoundException
     */
    public MatrixPacket receivePacket1() throws IOException, ClassNotFoundException {
        MatrixPacket p = new MatrixPacket();
        InputStream inputStream = new BufferedInputStream(connection.getInputStream());
        ObjectInput s = new ObjectInputStream(inputStream);
        p = (MatrixPacket)s.readObject();
        return p;
    }

    /**
     * Receives a LabMatrix over a TCP connection. Reads the matrix
     * via a loop from the stream.
     * @return A MatCommMatrix of type double, char, float, int or String
     * @throws IOException
     */
    public MatrixPacket receivePacket2() throws IOException {
        DataInputStream inputStream = new DataInputStream(new BufferedInputStream(
            connection.getInputStream()));
        MatrixPacket packet = new MatrixPacket();
        packet.fromStream(inputStream);
        return packet;
    }

    /**
     * Receives a LabObject over a TCP connection. Reads the LabObject
     * via externalizable interface from the stream.
     * @return A LabObject (LabMatrix or LabStruct)
     * @throws IOException, ClassNotFoundException
     */
    public LabObject readLabObject() throws IOException, ClassNotFoundException {
        InputStream inputStream = new BufferedInputStream(connection.getInputStream());
        ObjectInput s = new ObjectInputStream(inputStream);
        LabObject obj = (LabObject)s.readObject();
        return obj;
    }

    /**
     * Sends a file containing of raw bytes using an TCP connection.
     * @param path The path where the inputfile is found
     * @throws IOException
     */

```

```

        */
        public void sendByteFile(String path) throws IOException {
FileInputStream fi = null;
OutputStream outputStream = new BufferedOutputStream(connection.getOutputStream());
ObjectOutputStream s = new ObjectOutputStream(outputStream);
try {
    fi = new FileInputStream(path);
} catch (IOException io) {}

byte[] buffer = new byte[fi.available()];
fi.read(buffer);
s.write(buffer);
s.flush();
fi.close();
}

/**
 * Receives a file containing of raw bytes using an TCP connection.
 * @param path The path where the file is written
 * @throws IOException,ClassNotFoundException
 */
    public void receiveByteFile(String path) throws IOException,
        ClassNotFoundException {

byte[] buffer = new byte[1024];
FileOutputStream fo = null;
InputStream inputStream = new BufferedInputStream(connection.getInputStream());
ObjectInputStream s = new ObjectInputStream(inputStream);
try {
    fo = new FileOutputStream(path);
    while(s.read(buffer) != -1) {
        fo.write(buffer);
    }
} catch (Exception e) {}
}

/**
 * Sends a file containing of Strings using an TCP connection.
 * @param path The path where the inputfile is found
 * @throws IOException
 */
    public void sendStringFile(String path) throws IOException {
FileReader fi = null;
BufferedReader br = null;
OutputStream outputStream = new BufferedOutputStream(connection.getOutputStream());
ObjectOutputStream s = new ObjectOutputStream(outputStream);
try {
    fi = new FileReader(path);
    br = new BufferedReader(fi);
} catch (FileNotFoundException fnf) {}

```

```

try{
    String theLine = br.readLine();
    while(theLine != null) {
        s.writeObject(theLine);
    }
    s.flush();
    theLine = br.readLine();
} catch (IOException io) {}
br.close();
}

/**
 * Receives a file containing of Strings using an TCP connection.
 * @param path The path where the file is written
 * @throws IOException,ClassNotFoundException
 */
    public void receiveStringFile(String path) throws IOException,
        ClassNotFoundException {
FileWriter fo = null;
BufferedWriter bw = null;
InputStream inputStream = new BufferedInputStream(connection.getInputStream());
ObjectInputStream s = new ObjectInputStream(inputStream);

try {
    fo = new FileWriter(path, false);
    bw = new BufferedWriter(fo);
    String theLine = (String)s.readObject();
    while(theLine != null) {
        bw.write(theLine, 0, theLine.length());
        bw.newLine();
        bw.flush();
        theLine = (String)s.readObject();
    }
} catch (Exception e) {}
bw.close();
}

/**
 * Creates a TCP connection to the desired server.
 * @param name A StorkNameEntry containing protocoltype, IP address, port and name
 * @return The socket
 * @throws IOException,UnknownHostException
 */
    private Socket connect2server(StorkNameEntry name) throws IOException,
        UnknownHostException {
InetAddress serveraddress = StorkNameEntry.getAddressFromIP(name.getIP());
int serverport = name.getPort();
Socket s = new Socket(serveraddress, serverport);
return s;
}

```



```

/**
 * Calls the StorkClient's lookup function.
 * @param name Name of the server to be looked up
 * @return List with all the NameEntries
 * @throws IOException
 */
private StorkNameEntryList nslookup(String name) throws IOException {
return nameServer.lookup(name);
}

private StorkClient nameServer = null;
private Socket connection = null;
}

```

TCPServer.java

```

package se.lth.control.tcp;
import se.lth.control.nameserver.*;
import java.net.*;
import java.io.*;

/**
 * This class provides functions to register (unregister) the server to a
 * nameserver.
 * Important: the TCP server has to be started BEFORE the TCP client is
 * started. The server listens for a request and accepts the connection.
 * Then, data can be received as well as sent.
 * @version
 * @author
 */

public class TCPServer {

    /**
     * Default constructor. Creates a TCPServer on localhost. If the BASE_PORT is
     * already in use, it tries to run the server on a higher port.
     * @param servername The name of this server
     * @throws IOException,UnknownHostException
     */
    public TCPServer(String servername) throws IOException, UnknownHostException {
nameServer = new StorkClient();
InetAddress serveraddress = InetAddress.getLocalHost();
int serverport = BASE_PORT;

while(true){
    try {
server = new ServerSocket(serverport);
break;

```

```

    } catch (IOException e) {
if(serverport < 65535)
    serverport++;
else serverport = 1024;
    }
}

this.name = new StorkNameEntry(StorkNameEntry.PROTOCOL_TCPMATCOMM,
    StorkNameEntry.getIPFromAddress(serveraddress),
    serverport,
    servername);
nsRegister();
}

/**
 * Creates a TCPServer on localhost. If the BASE_PORT is already in use, it tries
 * to run the server on a higher port. The nameserver can be specified.
 * @param servername The name of this server
 * @param nameserver_host The hostname of the custom nameserver
 * @param nameserver_port The portnumber of the custom nameserver
 * @throws IOException,UnknownHostException
 */
public TCPServer(String servername, String nameserver_host, int nameserver_port)
    throws IOException, UnknownHostException {
nameServer = new StorkClient(nameserver_host, nameserver_port);
InetAddress serveraddress = InetAddress.getLocalHost();
int serverport = BASE_PORT;

while(true){
    try {
server = new ServerSocket(serverport);
break;
    } catch (IOException e) {
if(serverport < 65535)
    serverport++;
else serverport = 1024;
    }
}

this.name = new StorkNameEntry(StorkNameEntry.PROTOCOL_TCPMATCOMM,
    StorkNameEntry.getIPFromAddress(serveraddress),
    serverport,
    servername);
nsRegister();
}

/**
 * Gets the server's InetAddress.
 * @return The InetAddress of this host
 * @throws UnknownHostException

```

```

        */
        public InetAddress getHost() throws UnknownHostException {
            return name.getAddressFromIP(name.getIP());
        }

        /**
         * Gets the serversocket.
         * @return The serversocket of this host
         */
        public ServerSocket getServerSocket() {
            return server;
        }

        /**
         * Deletes the server's entry in the nameserver and unblocks the serverport.
         * @throws IOException
         */
        public static void terminateServer() throws IOException {
            nsUnregister();
            server=null;
        }

        /**
         * Registers the server to a nameserver by calling the register()
         * function of the StorkClient.
         * @throws IOException
         */
        private static void nsRegister() throws IOException {
            nameServer.register(name);
        }

        /**
         * Unregisters the server from the nameserver by calling the unregister()
         * function of the StorkClient.
         * @throws IOException
         */
        private static void nsUnregister() throws IOException {
            nameServer.unregister(name);
        }

        /**
         * Allows the server to connect to another registered host.
         * @return The TCP socket to the specified host (and port)
         * @throws IOException,UnknownHostException
         */
        private Socket connect(StorkNameEntry name) throws IOException,
            UnknownHostException {
            InetAddress address = StorkNameEntry.getAddressFromIP(name.getIP());
            int port = name.getPort();
            Socket s = new Socket(address, port);

```

```

return s;
    }

    private static final int BASE_PORT = 1024;
    private static ServerSocket server = null;
    private static StorkClient nameServer = null;
    private static StorkNameEntry name = null;
}

```

UDPNode.java

```

package se.lth.control.udp;
import se.lth.control.nameserver.*;
import java.net.*;
import java.io.*;

/**
 * This class provides functions to start a UDP node, to register it,
 * to connect to another host and to terminate the node.
 * @version
 * @author
 */

public class UDPNode {
    public static final int UDP_MAXPACKETSIZE = 65507;

    //If the data fits into a single Ethernet packet, the efficiency is increased.
    //1452 bytes = 1460 - 28(UDP-Header)
    public static final int ETHERNET_PACKETSIZE = 1452;

    /**
     * Default constructor. Creates a UDP node on localhost. If the BASE_PORT is
     * already in use, it tries to run the udpSocket on a higher port.
     * @param nodename The name of this udpSocket
     * @throws IOException,UnknownHostException
     */
    public UDPNode(String nodename) throws IOException, UnknownHostException {
        nameServer = new StorkClient();
        nodeaddress = InetAddress.getLocalHost();
        nodeport = BASE_PORT;

        while(true){
            try {
                udpSocket = new DatagramSocket(nodeport);
                //System.out.println("The udpSocket is running on port " + nodeport);
                break;
            } catch (SocketException e) {
                //System.out.println("There is already a udpSocket on port " + nodeport);

```

```

if(nodeport < 65535)
    nodeport = nodeport+1;
else nodeport = 1024;
}
}

this.name = new StorkNameEntry(StorkNameEntry.PROTOCOL_UDPMATCOMM,
    StorkNameEntry.getIPFromAddress(nodeaddress),
    nodeport,
    nodename);
nsRegister();
}

/**
 * Creates a UDP node on localhost. If the BASE_PORT is already in use, it
 * tries to run the udpSocket on a higher port. Nameserver can be specified.
 * @param nodename The name of this udpSocket
 * @param nameserver_host The hostname of the custom nameserver
 * @param nameserver_port The portnumber of the custom nameserver
 * @throws IOException,UnknownHostException
 */
public UDPNode(String nodename, String nameserver_host, int nameserver_port)
    throws IOException, UnknownHostException {
nameServer = new StorkClient(nameserver_host, nameserver_port);
nodeaddress = InetAddress.getLocalHost();
nodeport = BASE_PORT;

while(true){
    try {
udpSocket = new DatagramSocket(nodeport);
        //System.out.println("The udpSocket is running on port "+ nodeport);
break;
    } catch (SocketException e) {
//System.out.println("There is already a udpSocket on port " + nodeport);
if(nodeport < 65535)
        nodeport = nodeport+1;
else nodeport = 1024;
    }
}

this.name = new StorkNameEntry(StorkNameEntry.PROTOCOL_UDPMATCOMM,
    StorkNameEntry.getIPFromAddress(nodeaddress),
    nodeport,
    nodename);
nsRegister();
}

/**
 * Gets the node's InetAddress.
 * @return The InetAddress of this host

```

```

*/
public InetAddress getHost() {
return nodeaddress;
}

/**
 * Gets the node's portnumber.
 * @return The portnumber of this host
 */
public int getPort() {
return nodeport;
}

/**
 * Gets the node's UDPSocket.
 * @return The UDPSocket of this host
 */
public DatagramSocket getSocket() {
return udpSocket;
}

/**
 * Returns the StorkClient.
 * @return The StorkClient
 */
public static StorkClient getStorkClient() {
return nameServer;
}

/**
 * Given a name, it looks for this name in the nameserver and calls
 * the connect2node() method.
 * @param node The name of the node to connect
 * @return DatagramPacket containing the connection-information
 * @throws IOException,UnknownHostException
 */
public DatagramPacket connect(String node) throws IOException {
DatagramPacket dummy = null;
StorkNameEntryList entries = nslookup(node);
if (entries.getSize() > 0) {
    if(entries.getEntry(node).getProtocol() == 3) {
StorkNameEntry entry = entries.get(0);
return connect2node(entry);
    }
    else {
System.out.println("Error: " + node + " is not a UDP node");
return dummy;
    }
}
System.out.println("Error: Server not registered!");
}

```

```

return dummy;
}

/**
 * Creates a DatagramPacket given a StorkNameEntry.
 * @param name A StorkNameEntry containing protocoltype, IP address,
 *           port and name
 * @return DatagramPacket containing the connection-information
 * @throws IOException,UnknownHostException
 */
private DatagramPacket connect2node(StorkNameEntry name) throws IOException,
                                   UnknownHostException {
InetAddress remoteaddress = StorkNameEntry.getAddressFromIP(name.getIP());
int remoteport = name.getPort();
byte[] buffer = new byte[UDP_MAXPACKETSIZE];
DatagramPacket s = new DatagramPacket(buffer, buffer.length,
                                     remoteaddress, remoteport);

return s;
}

/**
 * Unregister the udpSocket in the Nameserver.
 * @throws IOException
 */
public static void disconnect() throws IOException {
nsUnregister();
}

/**
 * Closes the UDP socket.
 * @throws IOException
 */
public void closeConnection() throws IOException {
System.out.println("Connection to node " +udpSocket.getInetAddress().getHostName()
                  + "closed!");
udpSocket.close();
udpSocket = null;
}

/**
 * Calls the StorkClient's lookup function.
 * @param name Name of the server to be looked up
 * @return A List with NameEntries
 * @throws IOException
 */
private StorkNameEntryList nslookup(String name) throws IOException {
return nameServer.lookup(name);
}

/**

```

```

 * Registers the node to a nameserver by calling the register()
 * function of the StorkClient.
 * @throws IOException
 */
private static void nsRegister() throws IOException {
nameServer.register(name);
}

/**
 * Unregisters the node from the nameserver by calling the unregister()
 * function of the StorkClient.
 * @throws IOException
 */
private static void nsUnregister() throws IOException {
nameServer.unregister(name);
}

private static final int BASE_PORT = 1024;
private static StorkClient nameServer = null;
private static StorkNameEntry name = null;
private int nodeport = 0;
private InetAddress nodeaddress = null;
private DatagramSocket udpSocket = null;
private DatagramPacket udpconnect = null;
}

```

StorkNameServer.java

```

package se.lth.control.nameserver;
import java.net.*;
import java.io.*;
import java.util.Date;

/**
 * This class provides methods to start a StorkNameServer on a specific port and to
 * register itself to a NameEntryList. In addition, the server's main functions
 * (Lookup, SaveRegistration and DeleteRegistration) are implemented in this class.
 * Finally, a logfile is written.
 * @version
 * @author
 */

public class StorkNameServer {

/**
 * Constructor. Starts the server on the given port, creates the NameEntry and
 * registers itself to the NameEntryList.
 * @param port The port where the NameServer is listening

```

```

        * @throws IOException,UnknownHostException
        */
        public StorkNameServer(int port) throws IOException, UnknownHostException {
nodeaddress = InetAddress.getLocalHost();
nodeport = port; //DEFAULT_PORT
path = dir.getAbsolutePath()+"/addum/log/";

try {
    server = new DatagramSocket(port);
} catch (SocketException e) {}

try {
    fw = new FileWriter(path + "nslog.txt",false);
    br = new BufferedWriter(fw);
} catch(IOException e){}

this.nsentry = new StorkNameEntry(StorkNameEntry.PROTOCOL_UDP,
    StorkNameEntry.getIPFromAddress(nodeaddress),
    nodeport,
    DEFAULT_NAME);
SaveRegistration(nsentry);
ServerThread ns = new ServerThread(server);
ns.start();
}

/**
 * Adds a given NameEntry to the NameEntryList.
 * @param reg The StorkNameEntry to register
 * @return True if registration was successful, false otherwise
 */
public static boolean SaveRegistration(StorkNameEntry reg) {
if(nslst.add(reg)) {
    String s = "NameEntry '"+reg.getName()+"'
                successfully registered to StorkNameServer!";

    Date now = new Date();
    String stamp = now.toString();
    try{
br.write(stamp, 0, stamp.length());
br.newLine();
br.flush();
br.write(s, 0, s.length());
br.newLine();
br.flush();
    }catch(Exception e){}
    System.out.println(s);
    return true;
}
return false;
}

```

```

/**
 * Deletes a given NameEntry from the NameEntryList.
 * @param unreg The StorkNameEntry to unregister
 * @return True if deregistration was successful, false otherwise
 */
public static boolean DeleteRegistration(StorkNameEntry unreg) {
if(nslst.delete(unreg)) {
    String s = "NameEntry '"+unreg.getName()+"'
                successfully deleted from StorkNameServer!";

    Date now = new Date();
    String stamp = now.toString();
    try{
br.write(stamp, 0, stamp.length());
br.newLine();
br.flush();
br.write(s, 0, s.length());
br.newLine();
br.flush();
    }catch(Exception e){}
    System.out.println(s);
    return true;
}
return false;
}

/**
 * Makes a lookup.
 * @return The entire NameEntryList
 */
public static StorkNameEntryList Lookup() {
//System.out.println("Lookup: NameEntryList will be sent to the server!");
return nslst;
}

private static final int DEFAULT_PORT = 2001;
private static final String DEFAULT_NAME = "StorkNameServer";
private static StorkNameEntryList nslst = new StorkNameEntryList();
private int nodeport = 0;
private InetAddress nodeaddress = null;
private DatagramSocket server = null;
private StorkNameEntry nsentry = null;
private String path = null;
private File dir = new File("");
private FileWriter fw = null;
private static BufferedWriter br = null;
}

```

LabObject.java

```
package se.lth.control.labcomm;
import java.io.*;

/**
 * This class is the superclass for LabMatrix and LabStruct.
 * A future version could also include arrays.
 * LabMatrix is the superclass for matrix classes of type double,
 * float, char, int and String.
 * LabStruct can be used as the superclass for custom structs.
 * @version
 * @author
 */

public abstract class LabObject implements Serializable {}
```

LabStruct.java

```
package se.lth.control.labcomm;
import java.io.*;

/**
 * This class provides the fields and methods for a more general
 * datastructure than just matrices. Since it implements the
 * externalizable interface, it can be written to a stream and also
 * read from a stream.
 * This class is also the superclass for further even more specific
 * structs that may be used in combination with specific softwarepackages.
 * @version
 * @author
 */

public class LabStruct extends LabObject implements Externalizable {

    /**
     * Mandatory no-arg constructor
     */
    public LabStruct() {}

    /**
     * This inner class provides the attributes of the LabStruct.
     */
    public static class Attribute implements Externalizable{

        /**
         * Mandatory no-arg constructor
         */
        public Attribute() {}
```

```
        public static String name;
        public static LabObject value;

        /**
         * This function is used to save the state of the object. Since the class
         * implements Externalizable, this method must be implemented. Only the
         * identity of the class is automatically saved by the stream.
         * @serialData Write name and value field as objects
         * @param out The ObjectOutputStream
         * @throws IOException
         */
        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeObject(name);
            out.writeObject(value);
        }

        /**
         * This mandatory method reads in the data that was written out
         * in the writeExternal method. It restores its own fields. These fields
         * must be in the same order and type as they were written out.
         * @param in The ObjectInputStream
         * @throws IOException, ClassNotFoundException
         */
        public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
            name = (String) in.readObject();
            value = (LabObject) in.readObject();
        }

        /**
         * Gets the name field of the inner class.
         * @return The attribute's name
         */
        public String getName() {
            return Attribute.name;
        }

        /**
         * Gets the value field of the inner class.
         * @return The attribute's value
         */
        public LabObject getValue() {
            return Attribute.value;
        }

        /**
         * Sets the name and value of the inner class.
         * @param name The name of the attribute
         * @param value The value of the attribute
         */
```

```

        */
        public void setAttribute(String name, LabObject value) {
            Attribute.name = name;
            Attribute.value = value;
        }

        Attribute[] field;

        /**
         * This function is used to save the state of the object. Since the class
         * implements Externalizable, this method must be implemented. Only the
         * identity of the class is automatically saved by the stream.
         * @serialData Write field-array field as object
         * @param out The ObjectOutput
         * @throws IOException
         */
        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeObject(field);
        }

        /**
         * This mandatory method reads in the data that was written out
         * in the writeExternal method. It restores its own fields. These fields
         * must be in the same order and type as they were written out.
         * @param in The ObjectInput
         * @throws IOException, ClassNotFoundException
         */
        public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
            field = (Attribute[]) in.readObject();
        }

        private static final long serialVersionUID = 8563170928982865348L;;
    }

```

LabMatrix.java

```

package se.lth.control.labmatrix;
import se.lth.control.labcomm.*;
import java.io.*;

/**
 * This abstract class is the superclass for the following classes:
 * LabCharMatrix, LabFloatMatrix, LabDoubleMatrix,
 * LabIntMatrix and LabStringMatrix. It extends LabObject.
 * @version
 * @author
 */

```

```

public abstract class LabMatrix extends LabObject implements Serializable {

    public LabMatrix() {}

    public LabMatrix(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
    }

    /**
     * Allows access to the protected number of rows of the matrix
     * @return The number of rows
     */
    public int getRows() {
        return rows;
    }

    /**
     * Allows access to the protected number of cols of the matrix
     * @return The number of cols
     */
    public int getCols() {
        return cols;
    }

    /**
     * The print method prints a matrix to the standard output.
     */
    public void printMatrix() {}

    protected int rows = 0;
    protected int cols = 0;
}

```

E.2 Computer Vision

JAIImageBuilder.java

```

package se.lth.control.IO;
import java.awt.image.IndexColorModel;
import java.awt.image.RenderedImage;
import java.awt.image.DataBuffer;
import java.awt.image.BufferedImage;
import java.awt.image.Raster;
import java.awt.image.WritableRaster;

```

```

/**
 * This class builds a buffered image given a buffer of
 * bytes or integers. It works for gray-scaled images, but
 * it can be used as template for other formats as well.
 */
public class JAImageBuilder {

    public static final int width = 320;
    public static final int height = 240;
    public static final int noOfBands = 1;
    public static final int imageType = BufferedImage.TYPE_BYTE_GRAY;

    /**
     * Builds the image given a databuffer.
     * @param picdata The data buffer
     * @return A buffered (rendered) image
     */
    public static BufferedImage build(Object picdata) {

BufferedImage image = null;
int dataType = 0;

if (picdata instanceof byte[]) {
    picdata = (byte[])picdata;
    dataType = DataBuffer.TYPE_BYTE;
}

if (picdata instanceof int[]) {
    picdata = (int[])picdata;
    dataType = DataBuffer.TYPE_INT;
}

image = new BufferedImage(width, height, imageType);
WritableRaster rast = Raster.createBandedRaster(dataType, width,
                                                height, noOfBands, null);
rast.setDataElements(0, 0, width, height, picdata);
image.setData(rast);

        return image;
    }
}

```

JAThreshold.java

```

package se.lth.control.process;
import java.awt.*;
import java.awt.image.RenderedImage;
import java.awt.image.Renderable.*;
import javax.media.jai.*;

```

```

/**
 * This class provides a threshold operation.
 * The Threshold operation takes one rendered image and maps all
 * the pixels of this image whose values fall within a specified
 * range to a specified constant. The range is specified by a low
 * value and a high value.
 */
public class JAThreshold {

    static int bands = 1; //only gray-scaled images are considered
    static double [] low = new double[bands];
    static double [] high = new double[bands];
    static double [] map = new double[bands];

    public static String getDemoName() {
        return "Threshold";
    }

    /**
     * Performs the thresholding.
     * @param im A rendered image
     * @param lowlev The lower limit for the pixels interest
     * @param highlev The upper limit for the pixels interest
     * @param maplev The new value for the pixels within the band
     * @return The processed image
     */
    public static RenderedImage process (RenderedImage im, int[] lowlev,
                                        int[] highlev, int[] maplev) {

for (int i = 0; i < bands; i++) {
    low[i] = (double)lowlev[i];
    high[i] = (double)highlev[i];
    map[i] = (double)maplev[i];
}

ParameterBlock paramBlock = new ParameterBlock();
    paramBlock.addSource(im);
    paramBlock.add(low);
    paramBlock.add(high);
    paramBlock.add(map);
    return JAI.create("threshold", paramBlock);
}
}

```


ImageAnalysis.java

```
package se.lth.control.process;
import java.awt.image.DataBuffer;
import java.awt.image.Raster;
import java.awt.image.RenderedImage;
import java.awt.Point;

/**
 * This class is used to determine the center of mass of a ball in a
 * image. The image (gray-scaled) must be thresholded before. A tracking
 * rectangle limits the searching area to the region of interest. Moreover,
 * a fixpoint can be tracked as well.
 */
public class ImageAnalysis {

    //The size of the tracking-rectangle for the center (fix)
    public final static int cwidth_def = 20;
    public final static int cheight_def = 20;

    //The size of the tracking-rectangle for the ball (fix)
    public final static int bwidth_def = 100;
    public final static int bheight_def = 100;

    public static int elem = 0;
    public static int n = 0;
    public static int xsum = 0;
    public static int ysum = 0;

    public static Point center = new Point();
    public static Point ball = new Point();
    public static Point[] coord = new Point[2];
    public static double xcenter = 0.0;
    public static double ycenter = 0.0;
    public static double xball = 0.0;
    public static double yball = 0.0;

    /**
     * Computes the center of mass in relation to a fixpoint.
     * @param iml A rendered image (thresholded, grayscaled)
     * @param xcinit The x-coord of the fixpoint
     * @param ycinit The y-coord of the fixpoint
     * @param xbinit The x-coord of the center of mass of the previous image
     * @param ybinit The y-coord of the center of mass of the previous image
     * @return The fixpoint and the center of mass
     */
    public static Point[] compute(RenderedImage iml, int xcinit, int ycinit,
        int xbinit, int ybinit) {

        int cwidth = cwidth_def;
```

```
        int cheight = cheight_def;
        int bwidth = bwidth_def;
        int bheight = bheight_def;

        //The top-left corner of the tracking-rectangle for the fixpoint
        int cx = xcinit - cwidth_def / 2;
        if (cx < 0)
            cx = 0;
        int cy = ycinit - cheight_def / 2;
        if (cy < 0)
            cy = 0;
        //Check if the rectangle is within the image-borders
        int hx = 320 - cx;
        int hy = 240 - cy;
        if (hx < cwidth_def)
            cwidth = hx;
        if (hy < cheight_def)
            cheight = hy;

        //The top-left corner of the tracking-rectangle for the ball
        //(center of mass resulting from the previous picture)
        int bx = xbinit - bwidth_def / 2;
        if (bx < 0)
            bx = 0;
        int by = ybinit - bheight_def / 2;
        if (by < 0)
            by = 0;
        //Check if the rectangle is within the image-borders
        hx = 320 - bx;
        hy = 240 - by;
        if (hx < bwidth_def)
            bwidth = hx;
        if (hy < bheight_def)
            bheight = hy;

        //Get the raster
        Raster imraster = iml.getData();

        //Get the data-buffers (picture must be gray-scaled!)
        int[] iarray = new int[cwidth*cheight];
        int[] jarray = new int[bwidth*bheight];
        int[] centbuff = imraster.getPixels(cx,cy,cwidth,cheight,iarray);
        int[] ballbuff = imraster.getPixels(bx,by,bwidth,bheight,jarray);

        int offset = 0;

        //Loop for the specified searching regions
        for (int j = 0; j < cheight; j++) {
            offset = j*cwidth;
```

```

        for (int i = 0; i < cwidth; i++) {
elem = centbuff[offset + i];

if (elem == 255) {
    n++;
    xsum = xsum + i;
    ysum = ysum + j;
}
}

//Calculate the center of mass
if (n != 0) {
    xcenter = (double)xsum / (double)n + (double)cx;
    ycenter = (double)ysum / (double)n + (double)cy;
}
center.setLocation(xcenter, ycenter);
n = 0;
xsum = 0;
ysum = 0;
//System.out.println("xcenter: "+xcenter);
//System.out.println("ycenter: "+ycenter);

for (int j = 0; j < bheight; j++) {
    offset = j*bwidth;

        for (int i = 0; i < bwidth; i++) {

elem = ballbuff[offset + i];
//System.out.println(elem);

if (elem == 255) {
    n++;
    xsum = xsum + i;
    ysum = ysum + j;
}
}
}
//Calculate the center of mass of the ball
if (n != 0) {
    xball = (double)xsum / (double)n + (double)bx;
    yball = (double)ysum / (double)n + (double)by;
}
ball.setLocation(xball, yball);
n = 0;
xsum = 0;
ysum = 0;
//System.out.println("xball: "+xball);
//System.out.println("yball: "+yball);

```

```

coord[0] = center;
coord[1] = ball;
return coord;
    }
}

```

E.3 Interfaces

camera.c

```

/*
 * Camera server execution entry point.
 *
 * Author: Mathias Haage
 * Date: 2001-07-26
 *
 * Modified: Samuel Kasper
 * Date: 2003-01-16
 */

#include "cameraserver.h"

fiCameraServerType fiCameraServer;

JNIEXPORT jboolean JNICALL
Java_VisionServer_StartupCameraServer (JNIEnv *env, jobject obj) {

fiCameraServerType* fiCameraServer1;
int i;
int iCmdShow = 1;

fiCameraServer1 = &fiCameraServer;

// Reset activation states
for (i=0; i<MAX_SERVER_INITIALIZATION_LEVELS; i++) {
fiCameraServer1->active[i] = FALSE;
}

// Set process priority
SetPriorityClass (
GetCurrentProcess (),
HIGH_PRIORITY_CLASS);
fiCameraServer1->active[SERVERSTATE_PROCESSPRIORITYSET] = TRUE;
}

```

```

// System is started in an offline mode
fiCameraServer1->SystemOnline = FALSE;

// Create thread synchronization events
if ((fiCameraServer1->LeftNewImageEvent =
    CreateEvent (NULL, FALSE, FALSE, NULL)) == NULL) {
    MessageBox (
        NULL,
        TEXT ("Could not create LeftNewImageEvent event"),
        TEXT ("Camera Server Error"),
        MB_OK );
    TerminateCameraServer (fiCameraServer1);
    return FALSE;
}
fiCameraServer1->active[SERVERSTATE_EVENT1CREATED] = TRUE;

/*if ((fiCameraServer1->RightNewImageEvent =
    CreateEvent (NULL, FALSE, FALSE, NULL)) == NULL) {
    MessageBox (
        NULL,
        TEXT ("Could not create RightNewImageEvent event"),
        TEXT ("Camera Server Error"),
        MB_OK );
    TerminateCameraServer (fiCameraServer1);
    return FALSE;
}
fiCameraServer1->active[SERVERSTATE_EVENT2CREATED] = TRUE;
*/

// Initialize Fire-iAPI
if ( !InitializeFIREiAPI () ) {
    MessageBox (
        NULL,
        TEXT ("Initialization of FIRE-iAPI failed"),
        TEXT ("Camera Server Error"),
        MB_OK );
    TerminateCameraServer (fiCameraServer1);
    return FALSE;
}
fiCameraServer1->active[SERVERSTATE_FIREiINITIALIZED] = TRUE;

// Get list of cameras on FireWire bus
if (!GetCameraList (
    fiCameraServer1->fiCameraGuidArray,
    MAX_BUS_CAMERAS,
    &fiCameraServer1->fiCamerasFound)) {
    MessageBox (
        NULL,
        TEXT ("Failed to acquire list of cameras from FireWire bus"),

```

```

    TEXT ("Camera Server Error"),
    MB_OK );
    TerminateCameraServer (fiCameraServer1);
    return FALSE;
}
fiCameraServer1->active[SERVERSTATE_CAMERALISTACQUIRED] = TRUE;

// Proceed only if at least one camera is found
if ( fiCameraServer1->fiCamerasFound < 1 ) {
    MessageBox (
        NULL,
        TEXT ("Less than one camera found on FireWire bus"),
        TEXT ("Camera Server Error"),
        MB_OK );
    TerminateCameraServer (fiCameraServer1);
    return FALSE;
}
fiCameraServer1->active[SERVERSTATE_ONECAMERAFOUND] = TRUE;

// Start up left camera (assumed to be the first found on bus)
if (!InitializeCamera (
    &fiCameraServer1->fiCameraGuidArray[0],
    (UCHAR) 0,
    &fiCameraServer1->LeftNewImageEvent,
    &fiCameraServer1->LeftCamera) ) {
    MessageBox (
        NULL,
        TEXT ("Could not initialize left camera"),
        TEXT ("Camera Server Error"),
        MB_OK );
    TerminateCameraServer (fiCameraServer1);
    return FALSE;
}
fiCameraServer1->active[SERVERSTATE_LEFTCAMERAINITIALIZED] = TRUE;

// Start up right camera (assumed to be the second found on bus)
/*
if (!InitializeCamera (
    &fiCameraServer1->fiCameraGuidArray[1],
    (UCHAR) 1,
    &fiCameraServer1->RightNewImageEvent,
    &fiCameraServer1->RightCamera) ) {
    MessageBox (
        NULL,
        TEXT ("Could not initialize right camera"),
        TEXT ("Camera Server Error"),
        MB_OK );
    TerminateCameraServer (fiCameraServer1);
    return FALSE;
}
*/

```

```

fiCameraServer1->active[SERVERSTATE_RIGHTCAMERAINITIALIZED] = TRUE;
*/

return TRUE;
}

BOOL TerminateCameraServer () {
fiCameraServerType* fiCameraServer1;
int i;

fiCameraServer1 = &fiCameraServer;

// Terminate left camera
if (fiCameraServer1->active[SERVERSTATE_LEFTCAMERAINITIALIZED]) {
fiCameraServer1->active[SERVERSTATE_LEFTCAMERAINITIALIZED] = FALSE;
TerminateCamera (&fiCameraServer1->LeftCamera);
}

// Terminate right camera
if (fiCameraServer1->active[SERVERSTATE_RIGHTCAMERAINITIALIZED]) {
fiCameraServer1->active[SERVERSTATE_RIGHTCAMERAINITIALIZED] = FALSE;
TerminateCamera (&fiCameraServer1->RightCamera);
}

// Close event objects
if (fiCameraServer1->active[SERVERSTATE_EVENT1CREATED]) {
fiCameraServer1->active[SERVERSTATE_EVENT1CREATED] = FALSE;
CloseHandle (fiCameraServer1->LeftNewImageEvent);
}

if (fiCameraServer1->active[SERVERSTATE_EVENT2CREATED]) {
fiCameraServer1->active[SERVERSTATE_EVENT2CREATED] = FALSE;
CloseHandle (fiCameraServer1->RightNewImageEvent);
}

// Post quit message
if (fiCameraServer1->active[SERVERSTATE_WINDOWSCREATED]) {
fiCameraServer1->active[SERVERSTATE_WINDOWSCREATED] = FALSE;
PostQuitMessage (0);
}

// Terminate Fire-iAPI
if (fiCameraServer1->active[SERVERSTATE_FIREiINITIALIZED]) {
fiCameraServer1->active[SERVERSTATE_FIREiINITIALIZED] = FALSE;
TerminateFIREiAPI ();
}

```

```

// Reset unattended activation states
for (i=0; i<MAX_SERVER_INITIALIZATION_LEVELS; i++) {
fiCameraServer1->active[i] = FALSE;
}

return TRUE;
}

JNIEXPORT jboolean JNICALL
Java_VisionServer_StopCameraServer (JNIEnv *env, jobject obj) {
fiCameraServerType* fiCameraServer1;
int i;

fiCameraServer1 = &fiCameraServer;

// Terminate left camera
if (fiCameraServer1->active[SERVERSTATE_LEFTCAMERAINITIALIZED]) {
fiCameraServer1->active[SERVERSTATE_LEFTCAMERAINITIALIZED] = FALSE;
TerminateCamera (&fiCameraServer1->LeftCamera);
}

// Terminate right camera
if (fiCameraServer1->active[SERVERSTATE_RIGHTCAMERAINITIALIZED]) {
fiCameraServer1->active[SERVERSTATE_RIGHTCAMERAINITIALIZED] = FALSE;
TerminateCamera (&fiCameraServer1->RightCamera);
}

// Close event objects
if (fiCameraServer1->active[SERVERSTATE_EVENT1CREATED]) {
fiCameraServer1->active[SERVERSTATE_EVENT1CREATED] = FALSE;
CloseHandle (fiCameraServer1->LeftNewImageEvent);
}

if (fiCameraServer1->active[SERVERSTATE_EVENT2CREATED]) {
fiCameraServer1->active[SERVERSTATE_EVENT2CREATED] = FALSE;
CloseHandle (fiCameraServer1->RightNewImageEvent);
}

// Post quit message
if (fiCameraServer1->active[SERVERSTATE_WINDOWSCREATED]) {
fiCameraServer1->active[SERVERSTATE_WINDOWSCREATED] = FALSE;
PostQuitMessage (0);
}

// Terminate Fire-iAPI
if (fiCameraServer1->active[SERVERSTATE_FIREiINITIALIZED]) {

```

```

fiCameraServer1->active[SERVERSTATE_FIREiINITIALIZED] = FALSE;
TerminateFIREiAPI ();
}

// Reset unattended activation states
for (i=0; i<MAX_SERVER_INITIALIZATION_LEVELS; i++) {
fiCameraServer1->active[i] = FALSE;
}

return TRUE;
}

JNIEXPORT void JNICALL
Java_VisionServer_getImageDataBuffer(JNIEnv * env, jobject obj) {
fiCameraType* fiCamera1;
int ImageIndex;
int i;
jbyte* buffpoint;

jbyte* imageDataBuffer;
jsize len = 320*240; //Gray-scaled image
jclass cls = (*env)->GetObjectClass(env, obj);
jfieldID fid;
jintArray arr;

fiCamera1 = &fiCameraServer.LeftCamera;
ImageIndex = fiCamera1->iImageIndex;
imageDataBuffer = (jbyte*)fiCamera1->pImageBuffer[ImageIndex].ImageData;

fid = (*env)->GetFieldID(env, cls, "image", "[B");
if (fid == 0)
return;

arr = (*env)->GetObjectField(env, obj, fid);
arr = (*env)->NewByteArray(env, len);
len = (*env)->GetArrayLength(env, arr);
buffpoint = (*env)->GetByteArrayElements(env, arr, 0);

for(i=0;i<len; i++)
buffpoint[i] = imageDataBuffer[2*i +1]; //odd indices carry intensity information

(*env)->ReleaseByteArrayElements(env, arr, buffpoint, 0);

(*env)->SetObjectField(env, obj, fid, arr);
}

```

VisionServer.h

```

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class VisionServer */

#ifdef _Included_VisionServer
#define _Included_VisionServer
#endif
#ifdef __cplusplus
extern "C" {
#endif
/* Inaccessible static: threadInitNumber */
/* Inaccessible static: stopThreadPermission */
#undef VisionServer_MIN_PRIORITY
#define VisionServer_MIN_PRIORITY 1L
#undef VisionServer_NORM_PRIORITY
#define VisionServer_NORM_PRIORITY 5L
#undef VisionServer_MAX_PRIORITY
#define VisionServer_MAX_PRIORITY 10L
/*
 * Class:      VisionServer
 * Method:     StartupCameraServer
 * Signature:  ()Z
 */
JNIEXPORT jboolean JNICALL Java_VisionServer_StartupCameraServer
(JNIEnv *, jobject);

/*
 * Class:      VisionServer
 * Method:     getImageDataBuffer
 * Signature:  ()V
 */
JNIEXPORT void JNICALL Java_VisionServer_getImageDataBuffer
(JNIEnv *, jobject);

/*
 * Class:      VisionServer
 * Method:     StopCameraServer
 * Signature:  ()Z
 */
JNIEXPORT jboolean JNICALL Java_VisionServer_StopCameraServer
(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif

```

E.4 Controller

PIDParameters.java

```
package controller;
/**
 * This class provides the parameters required for a
 * PID-Controller.
 */
public class PIDParameters implements Cloneable {
    double K; //Proportional factor
    double Ti; //Time constant integrator
    double Tr; //Tracking: LP-Filter with time constant Tr
    double Td; //Prediction horizon (derivative part)
    double N; //Maximum derivate gain (often 10 - 20)
    double Beta; //Setpoint weighting: 0<=Beta<=1
    double H; //Discretisation interval
    boolean integratorOn; //Conditional integration

    public Object clone() {
    try {
        return super.clone();
    } catch (Exception x) {return null;}
    }
}
```

PID.java

```
package controller;

/**
 * This class provides the implementation of a PID-Controller.
 */

public class PID {
    private PIDGUI GUI;
    private PIDParameters p;
    private double y, yref, yold, I, D, ad, bd, v;

    public PID(String name) {
        PIDParameters params = new PIDParameters();
        params.K = -0.15;
        params.Ti = 25.0;
        params.Tr = 0.2;
        params.Td = 1.8;
        params.N = 10.0;
        params.Beta = 1.0;
    }
}
```

```
params.H = 0.0667;
params.integratorOn = false;
I = 0.0;
D = 0.0;
yold = -1000.0;
setParameters(params);
GUI = new PIDGUI(this, params, name);
}

    public synchronized void setParameters(PIDParameters newParameters) {
    p = (PIDParameters)newParameters.clone();
    ad=p.Td/(p.N*p.H+p.Td);
    bd=p.N*p.K*p.Td/(p.N*p.H+p.Td);

    System.out.println();
    System.out.println("PID Settings:");
    System.out.println("K: "+p.K);
    System.out.println();
    System.out.println("Ti: "+p.Ti);
    System.out.println();
    System.out.println("Td: "+p.Td);
    System.out.println();
    System.out.println("N: "+p.N);
    System.out.println();
    System.out.println("Integrator on: "+p.integratorOn);
    }

    public synchronized double calculateOutput(double y, double yref) {
    this.y = y;
    this.yref = yref;

    if (yold<-999.0)
        yold=y;

    D = ad*D - bd*(y - yold);
    v = p.K*(p.Beta*yref - y) + D;
    if (p.integratorOn) {
        v = v + I;
    }
    return v;
    }

    public synchronized void updateState(double u) {
    if (p.integratorOn) {
        I = I + p.H*p.K/p.Ti*(yref - y) + p.H*(1.0/p.Tr)*(u - v);
    } else {
        I = 0.0;
    }
    }
}
```

```

yold = y;
}

    public synchronized long getHMillis() {
return (long)(p.H*1000.0); //Sampling interval in milliseconds
}
}

```

BallAndPlateRegul.java

```

package controller;

/**
 * This class provides the methods to regulate a 2D process
 * using two P(I)D controllers, one for each dimension.
 */

public class BallAndPlateRegul {
    private PID xpos = new PID("X-Pos Control");
    private PID ypos = new PID("Y-Pos Control");
    private double xposition,xposref,yposition,yposref,vx,ux,vy,uy;
    private double[] output = new double[]{0,0};
    private double uMax = 0.1745; //~10 degrees
    private double uMin = -0.1745; //~-10 degrees

    public BallAndPlateRegul() {}

    /**
     * Saturation
     */
    private double limit(double v, double min, double max) {
if (v < min) {
    v = min;
} else {
    if (v > max) {
        v = max;
    }
}
return v;
}

    /**
     * Generates the controller's output signal
     */
    public double[] regulate(double[] reference, double[] feedback) {
xposref = reference[0];
yposref = reference[1];

```

```

xposition = feedback[0];
yposition = feedback[1];

synchronized(xpos) { // To avoid parameter changed inbetween
    vx = xpos.calculateOutput(xposition,xposref);
    ux = limit(vx, uMin, uMax);
    output[0] = ux;
    xpos.updateState(ux);
}

synchronized(ypos) { // To avoid parameter changed inbetween
    vy = ypos.calculateOutput(yposition,yposref);
    uy = limit(vy, uMin, uMax);
    output[1] = uy;
    ypos.updateState(uy);
}
return output;
}
}

```

E.5 Matlab

labcomminterface.m

```

%Script: Matlab Interface for Java-LabComm

%This script shows how third-party Java classes can be used with Matlab.
%It starts a UDPNode together with send- and receive threads. All Java
%classes that have a constructor can be called from Matlab.
%All variables used in the script are global. Consequently, they are
%accessible from the workspace.
%This script can also be used as a template for other LabComm specific
%scripts.

%In the following, some useful Matlab-Java commands are listed:

%which classpath.txt; //checks the loaded classpath
%[M,X,J] = innmem; //returns all loaded Java-Classes in the variable J
%import; //returns current import list
%clear import; //clears the list of imports
%J = javaObject('class_name',x1,...,xn); //builds a Java object
%fieldnames(obj,'-full'); //returns the fieldnames of a object
%class(Object); //returns the class of any object
%isjava(Object); //determines if the object is a Java object
%isa(obj, 'class_name'); //find out if obj is an instance of 'class_name'

```

```

%methodsview class_name; //returns a window containing all the methods of a class
%methods('class_name','-full'); //returns the methods of a class
%which -all method; //determine what loaded classes define a method
%disp(X); //displays text or array
%isequal(A,B,...); //determines if arrays are numerically equal
%javaArray('package_name.class_name',x1,...,xn); //builds a Java array

%-----
%import the packages and define some variables

import se.lth.control.nameserver.*;
import se.lth.control.udp.*;
import matlab_interface.*
import java.lang.*;
import java.io.*;
import java.net.*;

clear all;
disp('Matlab Interface for Java-LabComm');
disp('-----');
contr = char('ControlServer');
vis = char('VisionServer');
true = 1;
false = 0;
%h = 33; %Sampling interval for fast WinMachine
h = 66; %Sampling interval for slow WinMachine

%-----
%clean the nameserver

try
    c = StorkClient('watt.control.lth.se' , 2006);
    list = c.lookup(contr);
    if (list.getSize > 0)
        for i=0:list.getSize-1
            c.unregister(list.get(i));
            disp('Entry deleted');
        end
    else
        disp('Nameserver ready');
    end
catch
    error(sprintf('Exception ns'));
end

%-----
%start and register the UDPNode, start the receive-thread

try
    str = input('Register node: y/quit ','s');

```

```

    if(strcmp(str,'quit'))
        break;
    else
        un = UDPNode(contr, 'watt.control.lth.se' , 2006);
        disp('UDPNode registered!');
    end
end

catch
    error(sprintf('Exception IO'));
end

rec = UDPReceiveMat(un.getSocket);
rec.start;

%-----
%connect to the desired node, start the send-thread

try
    while(true)
        vis = input('Input the Nodename to connect with or quit: ','s');
        if(strcmp(vis,'quit'))
            un.disconnect;
            break;
        end
        list = un.getStorkClient.lookup(vis);
        if (list.getSize ~= 0)
            if(strcmp(vis,'VisionServer'))
                dp = un.connect(vis);
                send = UDPSendMat(dp, un.getSocket, vis);
                send.start;
                break;
            end
        end
        disp('Server not connected or wrong server! Try again.');
```

matcommserver.m

```

%Script matcommserver.m

%-----
%import java packages

import se.lth.control.nameserver.*;

```



```

import se.lth.control.udp.*;
import matlab_interface.*
import java.lang.*;
import java.io.*;
import java.net.*;

disp('Matlab MatComm Server');
disp('-----');

global chIgrrip;
contr = char('ControlServer');
name = char('matserv');
true = 1;
false = 0;

%Note: The sampling intervall for trajectory generating must be
%somewhat shorter than the intervall of the loop. Otherwise,
%an increasing delay occurs.

%h=0.080; %sampling intervall for trajectory generating
%h1=0.100; %sampling interval
h=0.045; %sampling intervall for trajectory generating
h1=0.066; %sampling interval

%-----
%clean the nameserver

try
    c = StorkClient('watt.control.lth.se' , 2006);
list = c.lookup(name);
if (list.getSize > 0)
    for i=0:list.getSize-1
        c.unregister(list.get(i));
        disp('Entry deleted');
    end
    else
        disp('Nameserver ready');
    end
catch
    error(sprintf('Exception ns'));
end

%-----
%start and register the UDPNode, start the receive-thread

try
    str = input('Register node: y/quit ', 's');
    if(strcmp(str,'quit'))
        break;
    else

```

```

        un = UDPNode(name, 'watt.control.lth.se' , 2006);
        disp('UDPNode registred!');
    end

catch
    error(sprintf('Exception IO'));
end

rec = UDPReceiveMat(un.getSocket);
rec.start;

%-----
%open a connection to the IgrripServer
chIgrrip=matcomm('open','r2d2','IgrripServer');

%-----
%build and send the first trajectory
r=2*pi/360; %radiants must be used

j4s=90*r;
j6s=5*r;
j4e=j4s;
j6e=j6s;

js=[-15*r, 15*r, 35*r, j4s, 90*r, j6s]; %start position
je=[-15*r, 15*r, 35*r, j4e, 90*r, j6e]; %end position

dv=(je-js)/h;
traj=[eps:0.25:1]' [interp1q([0:1],[js;je],[0:0.25:1]')] [zeros(1,6);[dv;dv;dv;dv]];
traj(2:5,1)=traj(2:5,1)*h;
matcomm(chIgrrip,traj); %first trajectory is sent to the IgrripServer

t=h;
pause(h1);

%wait until data is available from the rec thread, then start the loop.
while true
    if rec.xcontr<-100 | rec.ycontr<-100
        disp('no control values available, wait...');
        pause(h1);
    else xangle=rec.xcontr;
        yangle=rec.ycontr;
        break;
    end
end

tic; %timer is set to 0
nexttime=h1;
irb2000openpose;
log_data=[];

```

```

while true
    %check if no stopcommand was received, otherwise break.
    if xangle<-100 | yangle<-100
        disp('StopSignal has been received!');
        break;
    end

    j4e=90*r+xangle; %x -> +j4

    if(j4e-j4s)>1*r %angle testing
        j4e=j4s+1*r;
    end
    if(j4e-j4s)<-1*r
        j4e=j4s-1*r;
    end

    if (j4e>100*r) %saturation testing
        j4e=100*r;
    end
    if(j4e<80*r)
        j4e=80*r;
    end

    j6e=5*r+yangle; %y -> -j6

    if(j6e-j6s)>1*r %angle testing
        j6e=j6s+1*r;
    end
    if(j6e-j6s)<-1*r
        j6e=j6s-1*r;
    end

    if(j6e>15*r) %saturation testing
        j6e=15*r;
    end
    if(j6e<-5*r)
        j6e=-5*r;
    end

    log_data=[log_data; j4e j6e irb2000readpose' toc];

    js=[-15*r, 15*r, 35*r, j4s, 90*r, j6s]; %start position
    je=[-15*r, 15*r, 35*r, j4e, 90*r, j6e]; %end position

```

```

dv=(je-js)/h;
traj=[eps:0.25:1]' [interp1q([0;1],[js;je],[0:0.25:1]')] [[dv;dv;dv;dv;dv]];
traj(1:5,1)=traj(1:5,1)*h+t*ones(5,1);
t=t+h;
matcomm(chIgrip,traj);

j4s=j4e; %previous endposition is the new startposition
j6s=j6e;

xangle=rec.xcontr; %new control-values are read in
yangle=rec.ycontr;

a=toc;
if a<nexttime
    pause(nexttime-a);
else disp('loop too fast!');
end

nexttime=nexttime+h1;
end

%last trajectory to terminate the transfer; sets the roboter to the initial
%postion for the ball-and-plate experiment. this trajectory is required since
%otherwise, the system has to be rebooted in order to sent new trajectories.

disp('last trajectory performed!!');

js=[-15*r, 15*r, 35*r, j4s, 90*r, j6s];
je=[-15*r, 15*r, 35*r, j4s, 90*r, j6s];

traj=[eps:0.25:1]' [interp1q([0;1],[js;je],[0:0.25:1]')] [eps*ones(4,6);zeros(1,6)];
traj(1:5,1)=traj(1:5,1)*h+t*ones(5,1);
matcomm(chIgrip,traj);

mjdeg([-15,15,35,90,90,5],5,chIgrip); %startposition for ball-and-plate

%-----
%close the connection to the IgripServer
matcomm('close',chIgrip);
irb2000closepose;
%-----

%Unregister the node (nameserver)
un.disconnect;

```