Miro

Manual

Version 0.9.4

April 25, 2005

For more paintings, see `http://www.bcn.fjmiro.es/`

Contents

Chapter 1

Introduction

Miro is a distributed object oriented framework for mobile robot control, based on CORBA (Common Object Request Broker Architecture) [5] technology. The *Miro* core components have been developed in C++ for Linux. But due to the programming language independency of CORBA further components can be written in any language and on any platform that provides CORBA implementations.

The *Miro* core components have been developed under the aid of ACE (Adaptive Communications Environment) [7, 8], an object oriented multi-platform framework for OS-independent interprocess, network and real time communication. They use TAO (The ACE ORB) [6] as their ORB (Object Request Broker), a CORBA implementation designed for high performance and real time applications. Therefore *Miro* should be easily portable to any other OS, where ACE and TAO run on. These are many Unix clones, Windows NT and some real time operating systems.

Miro is currently available for the RWI B21 platform (pre rflex), the Activmedia Pioneer family (namely the Performance PeopleBot), and the vi*Sparrow* architecture developed at the University of Ulm. We are convinced, that other ports can be done straight forward.

Miro was built since the existing robot control architectures didn't suffice our needs of usability, reliability, scalability and portability.

We used C++ due to its advantages in big projects, since it was especially designed for big projects. We have learned the hard way, that this is a serious requirement, for projects like a mobile robot control architecture.

We use multi processing, multithreading and the CORBA technology since most robotics applications are inherently concurrent and distributed. The hardware devices, like sensors and actuators run concurrently and due to the constant lack of computing power especially in computer vision, tend to reside on multiple computers. And as soon as cooperative behavior of multiple autonomous robots is of interest, the system as a whole is distributed anyways.

And last but not least we use ACE and TAO since these are multi platform, high performance libraries which proved to be very sophisticated in terms of usability, portability and scalability. — Additionally, they are open source libraries. They haven't disappointed us yet, to the contrary.

1.1 The *Miro* Group

The *Miro* core developers are (in alphabetical order):

- Stefan Enderle
- Gerhard Kraetzschmar
- Gerd Mayer
- Guillem Pages
- Stefan Sablatnög
- Steffen Simon
- Hans Utz

1.2 *Miro* Directory Structure

Miro comes with the full source code and documentation as well as a set of test programs and examples that should facilitate your first steps when writing your own programs. To help you navigate through the directory tree of a *Miro* distribution, we give a brief overview of the directories present in the *Miro* root directory:

bin: This directory contains links to the binaries of *Miro*. For an explanation of the individual binaries, see Chapter 4 about the robot services.

doc: Here, the available documentation is gathered. In the subdirectory `tex` you find the postscript version of this manual and in the directory `html` resides the auto generated online documentation of all *Miro* classes and their methods (the starting page of this documentation resides at:
`$MIRO_ROOT/doc/html/idl/index.html` respectively
`$MIRO_ROOT/doc/html/cpp/index.html`).

etc: Config files for the individual robots.

examples: Examples on how to use individual interfaces of *Miro*. If you want to write your own programs utilizing the *Miro* framework, this is a good place to look for inspirations.

idl: The IDL sources of the CORBA interfaces and data types.

performance-tests: Some tests that measure the performance of *Miro*'s services. — Not too much there at the moment.

scripts: Utilities for source code formatting and handling sequences of datafiles.

src: Here, all sources of the *Miro* services reside. New services for further robot platforms should go in here.

templates: Templates for Makefiles and the headers for source files. Copy the corresponding template, if you want to start a new subproject within the *Miro* source hierarchy. If you start your own new project on top of *Miro*, the Makefile templates might still be useful for you. They handle all the stuff concerning ACE/TAO and the multi-platform build process. See Chapter 6.1 for details.

tests: Small test programs to monitor or test isolated interfaces of the *Miro* robot control architecture.

utils: Utilities made for convenience. No magic to be expected here.

Chapter 2

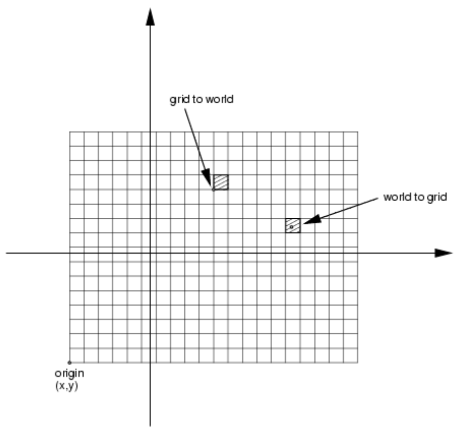
Definitions

2.1 Units

When it comes to parameter passing, distances are specified in mm and angles in radiant. Velocities are specified in mm/s and rad/s respectively.

2.2 Coordinates

Miro uses the cartesian coordinate space. Angles are specified mathematically, that is, counter-clock wise.


grid to world
world to grid
origin
(x,y)

Chapter 3

Using *Miro*

In order to use *Miro* programs, the following steps have to be performed:

- start the naming service
- start the services
- start the client program(s)

These steps are described in the following.

3.1 Starting the CORBA Naming Service

The CORBA naming service is some sort of telephone book service for the lookup of remote objects. Its organized like a file system. All entries in the naming service can be either object references, the pointer to a remote object, or naming context references that point to another subdirectory of the naming service. All CORBA objects of *Miro* are registered by *name* at a CORBA *naming service*. In the *Miro* project, the TAO implementation of the CORBA naming service is used and therefore must be started before any service is run.

The TAO naming service is started by the following command:

```
cd ${TAO_ROOT}/TAO/orbsvcs/Naming_Service
./Naming_Service
```

In order to get debug messages it can also be started like this:

```
cd ${TAO_ROOT}/TAO/orbsvcs/Naming_Service
./Naming_Service -ORBDebug -ORBDebugLevel <level>
```

where `<level>` is a debug level of 1 to 10 (10 gives most debug information).

3.1.1 Naming Service Lookup by IP-Multicast

Programs that use the TAO ORB can locate the naming service by IP multicast as long as it resided in the same subnet, or within the same multicast group. This is a special (proprietary) feature of the TAO naming service. You have to enable this feature explicitly by the command line option `-m 1`. To avoid conflicts, there should only be one naming service in a subnet. To make sure no other naming service is active or to check the entries in an active naming service, TAO provides a utility called `nslist`. It can be started by calling:

```
${TAO_ROOT}/TAO/utils/nslist/nslist
```

It lists the contents of the currently running naming service and returns an error, if no naming service is found.

3.1.2 Naming Service Lookup by Environment Variable

Additionally it is possible to not use the multicast feature of the TAO naming service and provide the IOR of the naming service as command line arguments or via environment variables to any program that uses TAO. This is explained in detail in the TAO documentation under:

```
${TAO_ROOT}/TAO/docs/INS.html
```

Note, that for retrieving the IOR of an already running naming service the little tool `nsIOR` resides in the bin directory of *Miro*.

You can also tell the Naming Service to listen to a specific endpoint instead of choosing its own by the use of the `-ORBEndpoint` command line option. Ensuring that the IOR of the naming service is the same, after each restart.

```
\${MIRO_ROOT}/scritps/naming_service
```

is a small init script for the SuSE Linux distribution, that can be installed in `/etc/rc.d` for starting the naming service automatically at boot time.

3.1.3 Naming Contexts

To provide the possibility to operate multiple robots in parallel, *Miro* uses the concept of naming contexts to distinguish between services of individual robots. One can imagine *naming contexts* as different folders, entries in different folders do not interact. The default naming context *Miro* services use is `Miro`. The canonical naming convention in a multi robot scenario is, that every robot uses its own name. Therefore every *Miro* service and client accepts an additional command-line argument that sets the *naming context*: `-MiroNamingContext <name>`. Or as short form: `-MNC <name>`. To access services of different robots simultaneously, the *naming context* has to be specified for the resolution of each individual service (see Chapter 5).

3.2 Starting Services

When the naming service is running and can be accessed from all machines, the most difficult part of the setup of the distributed system environment is mastered. From now on we are back to the unbounded problem set of robotics.

Simply change to the directory `$MIRO_ROOT/bin` and start the service you want to run. For example `b21Base`, `laserServer`, `videoService`, `dpPanTilt`. For an extensive list of all available services see Chapter 4. Since the services depend on the `*.conf` files provided in this directory, strange effects are bound to occur if the services are started from a different directory. Keep in mind, that the hardware dependent services must run on the computers the corresponding hardware is attached to.

When a service is ended disgracefully, it probably doesn't deregister its interfaces at the naming service. This will prevent the restated service from registering the new interface references. TAO provides a utility to manually remove references from the naming service.

```
${TAO_ROOT}/TAO/utils/nslist/nsdel
```

But *Miro* also provides a command line option to force the service to rebind the interface: `-MiroRebindIOR`.

For a complete list of command line option see Chapter 4.1.

3.3 Starting Client Programs

Miro comes with a rich set of client programs and utilities that help you to explore the robots sensory and actuatory devices. Starting a client program is basically the same as starting a service. Just change to the directory `$MIRO_ROOT/bin` and start whatever you want. Since these programs aren't bound to specific hardware devices they can be run on almost any machine.

Note, that to address a service that is registered within other than the default naming context at the naming service you also have to provide the `-MiroNamingContext` *context* argument at the command line (in short `-MNC` *context*).

Chapter 4

Available Services

The *Miro* framework abstracts the robots hardware devices as active services, that export the sensors and actuators functionality via CORBA interfaces that can be accessed transparently from other programs, probably running on totally different machines. Which services there are available depends on the individual robot type. To provide easy access to the services interface, each service registers itself under a standard name at the CORBA Naming Service. To allow multiple robots to be accessed at a time, each individual robot creates its one naming context within the Naming Service under which its services register. The default naming context is `Miro`, but a given robot should register itself under its own name (for example `stanislav`). The naming context to use can be specified at the command line of every service and example program via the `-MiroNamingContext` option.

4.1 Command Line Options

For each service the following command line options can be specified:

-MiroConfigFile *fileName*: The services are widely configurable. These parameters are specified in config files within the directory `$MIRO_ROOT/etc`. The file name is derived from the computers `$HOST` environment variable: `$HOST`.xml. If you want to specify another configuration file, you can do with this command line option.

-MiroNamingContext *contextName*: The short form of this option is **-MNC**. As mentioned before all services register their interfaces at the naming service within a naming context. The default naming context is `Miro`. In multi robot scenarios for instance it would be necessary to assign a different naming context to each robot. This can be done by this command line option. Note that since it is necessary to specify none default naming context names also at the client side, this option is also recognized by every client program.

-MiroRebindIOR: To prevent services from different robots to accidentally deregister each other from the naming service, an already existing entry in the naming service isn't unbound by a service, if it tries to register under the same name. Anyhow, if you need to overwrite an existing entry, specify this option at the command line.

4.2 Individual Interfaces

In the following, we list the different services and their interface names in the naming service, all interfaces are registered within the specified naming context in the naming service. For multi robot scenarios the canonical naming convention for the individual robots naming contexts is the robots name. Don't get confused by the fact that many services are collocated within a single binary, while for others there exists a dedicated binary. Note that in general the specialized interfaces of individual robots register at the Naming Service under their ancestors name. The interface documentation can be looked up within the *Miro* reference and in the *Miro* online documentation as well. For some services specializations for individual robot types do exist. Those are described in the subsequent sections.

Odometry: The motion service registers the `Miro::Odometry` interface as `Odometry`. It encapsulates the odometry (dead reckoning) sensor.

Motion: The motion service registers the `Miro::Motion` interface as `Motion`.

RangeSensor: The range sensor interface is the general abstraction of all range sensor devices, such as sonar, laser range finders etc. It offers a method for querying the sensors physical configuration as well as the latest sensor reading. See the `QtRangeSensor` program in the utils directory for an example on how to use this information. The range sensor interface becomes registered at the naming service under the name of the actual sensory device.

Sonar: The sonar service registers as `Sonar`. It provides an interface to the very common ultrasonic sensors for robots.

Infrared: It registers as `Infrared`.

Tactile: Some robots have bumpers as some sort of "it's too late" sensors. Still it is better to stop when you crashed than just moving on. It is registered as `Tactile`. Since it can be interpreted as a (very limited) range sensor, it is also a `Miro::RangeSensor`.

Laser: The laser scanner service registers as `Laser`. It provides very accurate 180**?** distance measurements, with up to 48Hz.

Stall: Similar to the bumpers, the stall detection monitors the robots motion and detects, when the robot is stuck in its movement. The `Miro::Stall` interface registers at the naming service as `Stall`.

Video: The video service registers the `Miro::Video` interface as `Video`. It provides camera images at a rate of up to 25 images per second captured by frame grabber cards. Currently supported are `matrox meteor` and cards that are supported by *video for linux* (bttv8). Since the bandwidth needed for uncompressed image data transportation exceeds the capabilities of most todays network devices and even memory copying can introduce excessive overhead, the video service enables access to the grabbed images via shared memory keys.

PanTilt: Cameras often are mounted on top of a pan-tilt unit, allowing the robot to look sideways while moving in another direction. Lightweight versions of such a device are panning or tilting units. The associated services register the `Miro::Pan`, `Miro::Tilt` and `Miro::PanTilt` interfaces as `Pan, Tilt` and `PanTilt` at the naming service.

Buttons: For simple user interaction, some robot types provide push buttons. The buttons service registers the `Miro::Buttons` interface as `Buttons`.

Speech: To provide a more natural way of communication, some robots are equipped with speech synthesizer cards. The speech service registers the `Miro::Speech` interface as `Speech`.

4.3 Asynchronous Sensory Information

Another feature of the *Miro* framework is the asynchronous distribution of sensory data via the CORBA Notification Service [4]. These event channels allow filtered and priority based event processing for time critical sensory information distribution under high load. Therefore in every robots naming context exists a reference `EventChannel` under which the robots notification service can be accessed.

The usage of the notification service within the *Miro* framework is explained in more detail in 6.5. In the following we give only a brief description. The notification service allows simple event filtering on the bases of the domain name and the type name of the event. This way the individual consumer can easily subscribe for the messages it is interested in. As the domain name, the naming context is used within the *Miro* framework (note that in a multi robot scenario, an event channel can transport data from multiple robots). In the following the type names of the events and the data contained in the event are listed for the individual services:

Odometry: The odometry servant and all derived services generates periodical `Miro::MotionStatusIDL` events), that propagate the robots current position and velocities. The event type name is `Odometry`.

RangeSensor: The range sensor servant and all derived services can generate periodical range sensor events. The event type name is that of the derived sensor interface (Sonar, Laser etc.). It is also returned by the `getScanDescription` method within the `ScanDescriptionIDL` struct. The event data is one of three types, as also specified within `Miro::SensorDescriptionIDL`.

Miro::RangeScanEventIDL The data of a range sensor that acquires data in a continuous scan pushes its data within this struct. (Currently, there does exist no actual sensor that behaves in this way, but anyways...)

Miro::RangeGroupEventIDL The data of a range sensor that acquires data groupwise pushes its data within this struct. A range sensor group normally is formed by a set of sensors that are mounted on the same hight, pointing in different directions. The laser scanners are organized that way.

Miro::RangeBunchEventIDL The data of a range sensor that acquires data in a discontinuous fashion pushes its data within this struct. Each sensor reading contains its own group id as well as its index within that group. Especially sonar sensor are organized that way, since neighbouring sensor mustn't be fired simultaneously to minimize crosstalk.

The event type name is that of the actual sensor: `Sonar`, `Laser`, `Tactile`, etc. as also specified within `Miro::SensorDescriptionIDL`.

Sonar: The sonar service generates range sensor events with the type name `Sonar`. Note that the sonar sensors are fired interleavingly to avoid crosstalk between them. Therefore the payload of an event emitted by a sonar device is normally a `RangeBunchEventIDL`. That way you have to analyze multiple `Sonar` messages to get a full sonar scan.

Infrared: The infrared service generates range sensor events with the type name `Infrared`.

Tactile: The tactile service generates range sensor events with the type name `Tactile`. Due to the hopefully low frequency at which those events occur it might be better to listen to the `Tactile` events, instead of polling the tactile status. Tactile devices usually contain a `RangeBunchEventIDL` struct.

Stall: The stall service also emits events. These contain the type name `Stall`. For the event data see the individual robots descriptions. Due to the hopefully rare occasions at which those events occur it might be better to listen to these events, instead of polling the stall interface.

Buttons: They also provide the event triggered communications model that uses the event type `Button`. The events payload is of type `ButtonStatusIDL`.

4.4 *B21*

The *B21* robot has a large number of available sensors and actuators. The relevant binaries for its specialized services are:

B21Base: This is a collection of the robots main services. This binary incorporates the motion service as well as the sonar, tactile and infrared services. Furthermore it gives access to the four colored buttons, that are mounted on top of the robot. The corresponding interfaces are:

- `Miro::B21Motion`
  The `Miro::B21Motion` interface is derived from the `Miro::Motion` interface and gives access to the specialized movement commands of the B21 robot. It registers under it ancestors name `Motion`.
- `Miro::Sonar`
  The derived `Miro::RangeSensor` interface registers itself as `Sonar`.
- `Miro::Infrared`
  The derived `Miro::RangeSensor` interface registers itself as `Infrared`.
- `Miro::Tactile`
  The derived `Miro::RangeSensor` interface registers itself as `Tactile`.
- `Miro::B21Buttons`
  The B21 has four buttons, that can be used as a simple user interface. The `Miro::B21Buttons` interface is supported by this service. It sends `Miro::ButtonStatusIDL` events, if a button is pressed or released.

SickLaserService: This is the service to access the SICK laser scanner. It supports the `Miro::RangeSensor` interface.

DtlkSpeech: This service lets you control the DoubleTalk Speech cards. The interface is named `Miro::Speech`

VideoService: It gives you access to the two frame grabbers of the robot. It supports the `Miro::Video` interface.

DpPantilt: It supports the `Miro::DirectedPerceptionPanTilt` interface, which is derived from `Miro::PanTilt`. It therefore registers itself as `PanTilt`

Note that the B21 has two internal computers, to which the different sensors and actuators are attached. You have to start the relevant binaries at the correct computer. The sensory/actuatory distribution in our laboratory is as follows: Start B21Base and LaserService on the left computer, VideoService and DpPantilt and DtlkSpeech on the right.

4.5 *Sparrow*

The *Sparrow* soccer robots have a similar diversity of sensors and actuators, but for efficiency and internal design reasons, they are grouped together in less binaries:

SparrowBase: Within this file the following services are grouped together: Motion, Stall, Kicker, Buttons, Sonar, Infrared and Pantilt. The corresponding interfaces are:

- `Miro::SparrowMotion`
  The `Miro::SparrowMotion` interface is derived from the `Miro::Motion` interface and gives access to the specialized movement commands of the *Sparrow* robot. It registers under it ancestors name `Motion`.
- `Miro::Stall`
  The `Miro::Stall` interface is registered as `Stall`.
- `Miro::Kicker`
  Due to its purpose as football robot, the *Sparrow* robot has a kicker, that can be accessed via this interface. It registers at the naming service as `Kicker`.
- `Miro::Buttons`
  The `Miro::Buttons` interface registers as `Buttons`.
- `Miro::Sonar`
  The `Miro::RangeSensor` interface registers itself as `Sonar`.
- `Miro::Infrared`
  The `Miro::RangeSensor` interface registers itself as `Infrared`.
- `Miro::PanTilt`
  The `Miro::SparrowPanTilt` interface is registered as `PanTilt`.

Note that the stall service sends `Miro::SparrowStallIDL` events.

VideoService: It gives you access to the frame grabber of the robot. It supports the `Miro::Video` interface.

4.6 Pioneer

Due to the limited variety of sensors and actuators on the Pioneer robots, there exists only the most fundamental services for this robot.

PioneerBase: The following services are grouped together: Motion, Stall and Sonar. The corresponding interfaces are:

- `Miro::PioneerMotion`
  The `Miro::PioneerMotion` interface is derived from the `Miro::Motion` interface and gives access to the specialized movement commands of the Pioneer robot. It registers under it ancestors name `Motion`.
- `Miro::PioneerStall`
  The `Miro::PioneerStall` interface (derived from `Miro::Stall`) is registered as `Stall`.
- `Miro::Sonar`
  The `Miro::RangeSensor` interface registers itself as `Sonar`.

Note that the stall service sends `Miro::PioneerStallIDL` events.

4.7 Frame Grabbers and Digital Cameras

The `VideoService` provides an interface to several video image sources. All BTTV cards that are supported by the video for linux project are also supported here (namely the Bt848/849/878/879 based frame grabbers). Support for IEEE 1394 digital cameras (aka Firewire) is provided. The Matrox Meteor frame grabbers cards are also supported, but note that kernel drivers are only available for the older 2.2.x kernel series. See the chaper **??** for a detailed discussion of the topic.

4.8 SICK Laser Scanner

The `sickLaserService` provides an interface to a laser range finder of type Sick PLS. This sensor is connected to the controlling PC via a serial line. It is delivered either with a RS-232 or a RS-422 compliant interface.

The idl-interface implemented by the `sickLaserService` is `Miro::Laser`, which is derived from `Miro::RangeSensor`. The server can be driven in two different modes, either the sensor automatically provides a scan about 40 times a second, or every scan is polled, which results in a maximum scan frequency of 10 times a second. The second mode also allows to choose a different (i.e. lower) scan frequency. At the moment no other special features of the Sick PLS200 are supported.

The configuration of this service is provided in xml, within the section `sick`. The following parameters are understood:

device: The filename of the device which should be used.

baudrate: The baudrate for communication with the sensor, possible values are: 9600, 19200, 38400 and 500000.

stdcrystal: For achieving the unusual baud rate of 500000 baud, we use a serial adapter, that is equipped with a 16MHz crystal, instead of the standard 14.??MHz. Due to this reason, the standard baud rates must be generated by modifying the divisor of the UART. If you have an off-the-shelf serial adapter, use `true` here, you will probably not be able to use 500000 baud then.

continousmode: If `true` is provided for this parameter the sensor will provide data automatically, about 40 times a second. If `false` is used, the service will poll a measurement after an interval that can be specified with the following parameter.

pollintervall: The interval between two consecutive measurements, measured in microseconds. This parameter has no effect if continousmode is set to `true`.

The following example shows the configuration used for our hardware:

```
<sick>
  <device>/dev/laser</device>
  <baudrate>500000</baudrate>
  <continousmode>true</continousmode>
  <stdcrystal>false</stdcrystal>
  < pollintervall >100000</pollintervall><!-- pollinterval in usec -->
</sick>
```

As the laser provides a specialization of a range sensor, we also provide a scan-description for it. See section **??** for details.

4.8.1 Possible Problems

Due to the unusual and high data rate of 500000 baud a special serial interface card is required. The card recommended by Sick provides a 16550A compatible interface, for that reason the current implementation is able to use the card through the standard linux serial device drivers, which keeps the implementation of the service more portable. Unfortunately this introduces a big drawback: receiving up to 50000 characters per second, results in at least 50000/16 = 3125 interrupts per second, this is an enormous load. It is essential, that every interrupt is handled on time, because the sensor does neither support hardware, nor software flow control, and provides a packet with data every 25 ms. If you have the chance to use an interface card with a larger FIFO buffer than the standard 16 bytes, this will provide a large improvement in performance, stability and system reactivity. We solved most of our stability problems, that were caused by lost packages without a larger FIFO: We enabled irqs throughout IDE interrupt processing (this may be extremely dangerous, depending on your configuration, see man page of *hdparm*). Another solution could be using *irqtune* to give the serial irq a higher priority, or using *RT-Linux*.

Due to the adaptations of the divisor for the serial line, the `SickLaserService` has to be suid root. If you use a standard crystal, and you do not use 500000 baud, this is not necessary.

4.9 DoubleTalk Speech Card

4.10 Directed Perception Pantilt

Chapter 5

Test and Example Programs

The *Miro* repository comes with different kinds of test and example programs that are scattered over four different directories:

examples These programs show the easiest way of using a specific service. In most cases it's just getting data and printing it to `cout`.

utils These are programs designed to fasciliate the handling of your mobile robots. For example, programs located in this directory provide a graphical view of the services output or state.

tests These programs are designed to reliably test an entire interface. They often come with a simple character dialog and the possible selections rely directly to the interfaces as described in the auto-generated online help.

performance-tests They measure the performance of individual sensory or actuatory devices. Mostly the throughput.

Chapter 6

Writing Your First Programs

This chapter tries to help you with the first steps of writing programs that use the *Miro* framework. The CORBA environment and the *Miro* framework seem to raise the bar for an easy entry into robot programming. While this can hardly be denied they facilitate tremendously the task of writing distributed programs. And since robot control software is inherently distributed (ever thought of multiple robots?) it seems the only way to go.

As you will see, most of the distributed programming complexity is initially hidden from the user:

- The programmer simply calls methods of the devices interface.
- The programmer communicates via proxy object to the remote service. The most tricky part is getting the object reference.
- The distributed environment is transparent. The remote method invocation (RMI) is hidden entirely.

6.1 Makefiles

In contrary to earlier versions of *Miro*, it is no longer based on the ACE/TAO Makefile hierarchy, but uses the GNU autotools for a simpler and more portable way to generate the Makefiles (beside that, this inherently adds new nifty features, like e.g. the automatically distribution via a simple `make dist`). Nevertheless it is still possible to use the ACE/TAO style of writing Makefiles in derived projects, as we didn't delete the *Miro* include rules and macros (but please note that we do not maintain them any longer and they may be deleted in future version).

Makefile templates reside in the directory `$MIRO_ROOT/templates`, that should facilitate the creation of further subprojects within the *Miro* directory structure but can just as well be used to start new projects outside the `$MIRO_ROOT` directory. The makefiles are designed to either build subdirectories, libraries or binaries. To use a template, simply copy it into the corresponding directory under the name `Makefile.am` and adapt it as explained in the following sections. Additionally, you have to put a line in `configure.ac` to the macro named `AC_CONFIG_FILES` to let automake and autoconf know, that they have to build a `Makefile` (`Makefile.in` respectively).

The Makefile templates are explained in more detail in the appendix **??**.

6.2 A Simple Sonar Client

To discuss things using actual code, let's look at the simple task of obtaining data from a sonar sensor device. Since this is a range sensor, the sonar is queried via the generalized range sensor interface. The only difference in querying an infrared or a sonar device is the name under which these sensors are registered within the naming service.

Listing 6.1: examples/sonar/SonarPoll1.cpp

```
#include <miro/Client.h>
#include <idl/RangeSensorC.h>

#include <ace/Log_Msg.h>

#include <iostream>

int main(int argc, char *argv[])
{
  Miro::Client client(argc, argv);           // Initialize orb.
  Miro::RangeSensor_var sonar =              // Obtain reference to sonar object.
    client.resolveName<Miro::RangeSensor>("Sonar");
  Miro::RangeScanEventIDL_var sonarScan;     // A RangeScanIDL smart pointer.

  sonarScan = sonar->getFullScan();          // Get values of all sonars.

  std::cout << "Sonar␣reading:␣";            // Print sonar scan.
  for (unsigned int i = 0; i < sonarScan->range.length(); ++i) {
    for (unsigned int j = 0; j < sonarScan->range[i].length(); ++j)
      std::cout << sonarScan->range[i][j] << "\t";
    std::cout << std::endl;
  }

  return 0;
}
```

A step by step walk trough the code:

```
#include <miro/Client.h>
```

The file miro/Client.h contains the definition of the class Miro::Client. See below.

```
#include <idl/RangeSensorC.h>
```

The file miro/RangeSensorC.h provides the classes for the interface of the sonar service. The 'C' at the end of the name stands for client. This file is automatically generated from the idl-description of the interface.

```
#include <iostream>
```

A standard main function is used.

```
int main(int argc, char *argv[])
```

This class just wraps the few lines that are necessary at the beginning of any CORBA application. It will therefore be normally instantiated in all *Miro* applications first. This is a simple helper class that sets up the CORBA environment in a standard way. Nothing tricky there. It initializes the orb etc. The call uses the command line arguments for finding the naming service, and parsing other commands...

```
{
  Miro::Client  client (argc, argv);          //          Initialize          orb     .
```

This initializes a proxy object to the sonar service. This proxy object is used as if it was the sonar service itself. Let's look at the call a bit closer. Miro::RangeSensor_var is the CORBA equivalent of a standard C++ auto pointer for a Miro::RangeSensor object. That is the proxy becomes automatically destroyed if the pointer goes out of scope. The method `resolveName()` is a template member function. Its argument specifies the name that shall be resolved in the default naming context of the naming service (see Chapter 3). The Miro::RangeSensor in French brackets specifies the type of reference that shall be returned by the call. Note that this will only succeed if the reference stored under the name of Sonar refers indeed to a `Miro::RangeSensor` object or to a derived ancestor of this class.

```
  Miro::RangeSensor_var sonar =           // Obtain reference to sonar object .
```

The RangeSensor interface returns a pointer to the sensor scan, which the caller obtains ownership of. So we use another auto pointer to hold the return value of the call and to prevent us from memory leaks. The IDL in the type name reflects the fact, that this is a IDL defined data type. (This is just a naming convention of *Miro* that shall help you to trace the roots to the documentation.) In the CORBA mapping for C++ these types are mapped to plain C structs: No methods, no inheritance, just public data members.

```
  Miro::RangeScanEventIDL_var sonarScan;    // A RangeScanIDL smart pointer.
```

This gets a sonar scan from the service. Note, that we do not see where the service runs.

```
  sonarScan = sonar->getFullScan();     //   Get   values   of   all   sonars  .

  std::cout << "Sonar reading: ";           //       Print       sonar       scan     .
  for (unsigned int i = 0; i < sonarScan->range.length(); ++i) {
    for (unsigned int j = 0; j < sonarScan->range[i].length(); ++j)
      std::cout << sonarScan->range[i][j] << "\t";
```

These lines are used to print the received sonar scan to the console.

6.3 Using Namespaces

By inserting the lines

```
using std::cout;
using std::endl;
using Miro::Client;
using Miro::RangeSensor;
```

we map the relevant "Miro" data types from the Miro namespace into our global namespace. This shortens the type name specifiers, but also hides from where they are coming from. Note that a simple

```
using namespace Miro;
```

would do the same trick. But mapping a namespace completely is generally not a good idea, since it tends to produce name conflicts and secondly code reviewers can trace the origin of the types less easily. Be especially careful in header files. You are flattening the namespace for everyone that has to include your header file, which can lead to bad surprises.

See the following listing which shows the same code again, but without the Miro:: prefixes. The initial lines containing the includes are skipped.

Listing 6.2: examples/sonar/SonarPoll2.cpp

```
using std::cout;
using std::endl;
using Miro::Client;
using Miro::RangeSensor;
using Miro::RangeSensor_var;
using Miro::RangeScanEventIDL_var;

int main(int argc, char *argv[])
{
  Client client(argc, argv);              // Initialize orb.
  RangeSensor_var sonar =                 // Obtain reference to sonar object.
    client.resolveName<RangeSensor>("Sonar");
  RangeScanEventIDL_var sonarScan;        // A RangeScanIDL smart pointer.

  sonarScan = sonar->getFullScan();       // Get values of all sonars.

  cout << "Sonar reading: ";              // Print sonar scan.
  for (unsigned int i = 0; i < sonarScan->range.length(); ++i) {
    for (unsigned int j = 0; j < sonarScan->range[i].length(); ++j)
      cout << sonarScan->range[i][j] << "\t";
    cout << endl;
  }

  return 0;
}
```

6.4 Adding Exception Handling

Handling of error conditions itself is error prone. Exceptions make the handling of error conditions easier, but errors are errors and therefore stay somehow nasty.

What's quite easy to achieve by the use of exceptions is to print some diagnostic output and exit instead of gracefully segfaulting. This is done in this example by enclosing the code in the main function in a try/catch block.

Since *Miro* defines ostream operators for every Miro ::... IDL type in miro/IO.h, we include that file and abandon the handcrafted streaming of the sonar data.

Note the different kinds off exceptions. There are *Miro* exceptions. Exception types defined in *Miro* are beginning with a big E as a naming convention. These indicate problems on the service side, like hardware problems (may be the batteries?), bad service calls (trying to accelerate the robot to warp 1?) or load problems. Then there are CORBA exceptions. Those occur if there arise some communication problems: A service went down, the robot is loosing the radio ethernet connection etc. Since all *Miro* exceptions derived from CORBA::UserException, those also are catched within the first **catch** block of the example code. You do not have to catch all exceptions. An uncaught exception will lead to program termination. Just as if you'd catch them at the end of main and exit...

Note also, that the instantiation of Client isn't within the try/catch block. This is intentionally. Exceptions that can arise in the construction of a Client instance are CORBA exceptions. The ostream operators for CORBA exceptions used by TAO need an ORB instance. Since the ORB is instantiated within the Client class, it will not exist after destruction of the Client — and this would be done at the end of the try block. Therefore we catch exceptions within the constructor of Client, print them to stderr and exit. There is little to do for the client program anyhow, if it can't access the services.

Listing 6.3: examples/sonar/SonarPoll3.cpp

```
int main(int argc, char *argv[])
{
  Client client(argc, argv);                        // Initialize orb.
  try {                                  // Obtain reference to sonar object.
    RangeSensor_var sonar = client.resolveName<RangeSensor>("Sonar");
    RangeScanEventIDL_var sonarScan;          // A RangeScanIDL smart pointer.

    sonarScan = sonar->getFullScan();   // Get values of all sonars.

    cout << "Sonar reading: " << sonarScan.in() << endl; // Print sonar scan.
  }
  catch (const CORBA::Exception& e) { // Catch any CORBA exception.
    cerr << "Exception on sonar query:" << endl << e << endl;
    return 1;
  }

  return 0;
}
```

6.5 An Asynchronous Sonar Client

By now we were actively requesting for the data of a service. But think of polling for tactile events that way. Having a good collision avoidance, those events should hardly ever occur. Nevertheless, as soon as there is a tactile signalling a collision, the robot should immediately react to this event. By polling it would have to call the tactile interface thousands of times, just not to miss the one event it can't effort to miss. And since the sensors are actively collecting their data, shouldn't they be able to trigger the data processing within the robot? - Oh yes, they can.

For this purpose the notification framework within *Miro* does exist. It is based on the CORBA Notification Service [**?**], and precustomized by some utility classes. They enable clients to subscribe to arbitrary events of a notification channel. The data gets pulled to them as soon as it becomes available at the producer (e.g. a range sensor device).

To show how asynchronous event processing works within *Miro*, lets look at a small example. First we look at the code to handle the events.

Listing 6.4: examples/sonar/SonarNotify.cpp

```
struct SonarNotify : public StructuredPushConsumer
{
    // Initializing Constructor .
  // Registers for the events , that it wants to get pushed .
  SonarNotify(EventChannel_ptr _ec , const string& domainName) :
    StructuredPushConsumer(_ec)
  {
    EventTypeSeq added, removed;
    added.length (1);
    added [0]. domain_name = CORBA::string_dup(domainName.c_str());
    added [0]. type_name = CORBA::string_dup("Sonar");

    setSubscriptions (added);
  }

    // Inherited IDL interface .
  // Called for every event by the event channel .
  void push_structured_event (const StructuredEvent& notification
                              ACE_ENV_ARG_DECL_WITH_DEFAULTS)
    throw(SystemException, Disconnected)
  {
    // Get a pointer to the sensor data of the structured event .
    const RangeBunchEventIDL * pSonarEvent;
    if ( notification .remainder_of_body >>= pSonarEvent)
      cout << *pSonarEvent << endl; // Print to standart out .
    else // Crises ? What crises ?
      cerr << "No␣␣RangeBunchEventIDL␣message." << endl;
  }
};
```

A step by step walk trough the code:

The class SonarNotify provides a callback for the event channel. It is derived from Miro::StructuredPushConsumer, which handles the registration at the notification service etc. All the initialization and registering is done within the constructor of this super class. The method push_structured_event is the callback that is called from the notification service.

```
  {
```

The EventTypeSeq class is for specifying the events that you want to be subscribed for. It is an incremental protocol. So you make a list of events you want to receive from now on and another with the events you wish to no longer become informed about. Since we are just subscribing, the second list is empty.

```
    added.length (1);
    added [0]. domain_name = CORBA::string_dup(domainName.c_str());
    added [0]. type_name = CORBA::string_dup("Sonar");
```

Events are subscribed by domain_name and type_name. Never forget to set the length of a sequence explicitly. Specifying the length as a constructor parameter just reserves the number of elements, the length of the sequence is still zero.

```
    setSubscriptions (added);
```

Tell the consumer admin what we want to subscribe for.

```
    if ( notification .remainder_of_body >>= pSonarEvent)
      cout << *pSonarEvent << endl;  // Print to standart out .
```

The payload of a structured event is contained within the remainder_of_body field with is of type CORBA::Any. Therefore you can get a const pointer to the data with overloaded the **operator** >>= (). The return value is a CORBA::Boolean indicating success of the operation. Even though a CORBA::Any can hold any IDL defined struct, you can only extract type T from a CORBA::Any if it actually contains an object of type T.

Listing 6.5: examples/sonar/SonarNotify.cpp

```
int main(int argc, char *argv[])
{
  Server server (argc, argv);     // Create a server orb .
  try {     // Resolve the channel by name .
    EventChannel_var ec( server .resolveName<EventChannel>("EventChannel"));
      // The consumer , that gets the events .
    SonarNotify pushConsumer(ec.in(), server .namingContextName);

    server .run ();     // Enter CORBA event loop .
  }
  catch (const CORBA::Exception & e) { // Catch CORBA and Miro exceptions.
    cerr << "Uncaught␣CORBA␣exception:␣" << e << endl;
    return 1;
  }
  return 0;
}
```

Lets now have a look at the main function:

```
  Server server (argc, argv);     // Create a server orb .
```

Since a push consumer is called (pushed) by the event producer, it is actually a server instead of a client. Therefor the Miro::Server class is instanciated. It performs the necessary calls to init the CORBA environment. It is basically the same as the Miro::Client but we also need a POA to register the consumer object.

```
EventChannel_var ec(server.resolveName<EventChannel>("EventChannel"));
```

Just like the former Miro::RangeSensor interface, we resolve the EventChannel by name at the naming service.

```
SonarNotify pushConsumer(ec.in(), server.namingContextName);
```

Instanciate the push consumer. It does all the necessary initialization within its constructor.

```
server.run();   // Enter CORBA event loop.
```

Now we enter the CORBA event loop. This will not return, until the process is signaled by SIGINT or SIGTERM.

Chapter 7

Parameter Sets

Robot control programs and robotic algorithms are full of parameters. Different robot types vary in shape and sensor configuration, but also different robots of the same type tend to vary slightly. For instance some of them are equipped with further sensory or actuatory devices not present in the standard configuration. Also damage and repair of parts of a robot during its lifetime result in an individual robot, which has its own unique configuration.

Furthermore the environment provides a complete set of parameters. Lighting conditions, the shape and location of rooms and corridors, the position of obstacles like tables or stairs. The list could be continued indefinitely. These parameters vary between different environment, but also tend to change slightly over time. Light bulbs are changed, cupboards added, tables moved etc.

A third source of parameters are defined by the task, the robot has to perform or the scenario, the robot is designed to operate in. Generalized tool boxes like planners or knowledge bases have to be populated with the actions and objects of relevance and a reactive execution engine need to be configured with what actions to take in which situation.

The configuration of these parameter sets can be done either by hand or tool supported or fully autonomously. Nevertheless it is a crucial part of the setup of robot for every scenario. The handling of those parameter sets therefore has to be well defined in order to keep the robots control software maintainable and adaptable to new tasks.

In this chapter we will therfore introduce the parameter framework of *Miro*. The parameter framework was primarily designed to handle the first source of parameters, namely the configuration of the robot and its sensory and actuatory services for a task. But meanwhile it is also used for the third part of the above sketched three sources of parameter sets: With slight extensions, it is used for the parameterization and configuration management of the reactive execution engine (the BAP framework) introduced in sectin 12. Environmental modeling for different scenarios lies out of the scope of the problem set currently addressed by *Miro*.

The remaining of this chapter is organized as follows. First the general concept of parameter sets and configuration files is sketched. Then the location and naming of the robot configuration files of *Miro* and how to create a configuration for an individual robot will be explained. In the following section the description and implementation of parameter sets and the mapping to the actual parameter files will be discussed. In section 7.3.1 the syntax and semantics of the description language will be explained.

7.1 Overview

Parameter sets are a quite powerfull concept within *Miro*, that therfore also encapsulates a lot of functionality. To facilitate the usage of the parameter framework of miro, tools support the handling of parameter sets for the different user communities. In this section we will describe the usage of parameter sets from the end-user as well as from the programmers perspective

7.1.1 End-User Perspective

An end-user has to handle parameter files for its robots. A parameter file contains various parameter sets, thatare described by an XML syntax. The server parses those files on startup, to initialize the parameter sets of its services. Parameters in general are strictly typed name value pairs. They are grouped into sections within the file. They can be nested, structured types, including arrays of variable length as well as sets. The parameter names start with an upper case initial letter.

To edit parameter files, the GUI-based `ConfigEditor` can be used. It enforces the XML-syntax as well as the correctness of the configuration section placement and the type correctness. It is discussed in more detail in section **??**.

7.1.2 Programmers Perspecitve

From a programmers perspective, a parameter set is represented by a struct within the target programming language. Currently only C++ is supported as target. For this language, a parameter set is translated into a class with all member variables declared public, just like a C struct. The member variable names begin with a lower case initial letter. Names consisting of multiple words are concatenated, the beginning of a new word is indicated by an upper case letter. Parameter classes support single, public inheritance. Apart from the member variables, the following methods are declared:

Default Constructor Initializes all member variables to their default values.

void operator ≪= (QDomNode) Parses an XML node and sets the member variables accordingly.

QDomElement operator ≫= (QDomElement) Appends the current values of the member variables as a child to the XML node.

void printToStream(ostream&) Prints the current values of the member variables to the specified output stream.

std::ostream& operator≪ (std::ostream&, Miro::Parameters& const) is declared for the parameter parent class.

Optionally parameter set classes contain a static member method `instance()`, that returns a pointer to the global heap allocated instance of the parameter set class (see also singleton pattern, and double checked locking [**?**]).

As parameter set classes have a very generic structure, their coding, including the XML and ostream operators can be automated. Therefore parameter sets normally are described not within a programming language, but within an XML syntax. The mapping to target programming language is then performed by a small compiler, which emits the necessary (i.e. C++) code. The parameter set descriptions are also used by the tools for GUI-based configuration editing. Those are the `ConfigEditor` as well as the `PolicyEditor` of the behaviour framework.

7.2 Configuration Files

Every service of *Miro* tries to find a configuration file on startup to adapt itself to the individual robots configuration if necessary. If no configuration file is found, the service has to use its default settings. The locations the configuration files are expected are:

1. `$MIRO_ROOT/etc`
2. The current directory.

The name of the configuration file is expected to be the host name of the computer the service is running on. The file extension is .xml:

`$HOST.xml`

For example in the case of our famous Sparrow99 soccer robot goal keeper named haeltnix, the *Miro* service SparrowBase, started on the computer mounted on the robot will try to load the configuration file:

`$MIRO_ROOT/etc/haeltnix.xml`

The default name and location of the configuration file can be overwritten by the command line option `-MiroConfigFile` *filename* or `-MCF` for short. The search order of directories stays the same. Additionally there can also be an absolute path specified.

In `$MIRO_ROOT/etc` there are sample configuration files for every robot model currently supported by *Miro*. They are named *model*`.xml`. Copy the respective configuration file of your robot model to `$HOST.xml` to get a configuration file to start with. XML files are text files and quite easily readable. Therefore opening the configuration file into your favorite editor and changing the *value* attributes to different settings should be sufficient to solve the first few configuration problems like differing device names etc. For anything else, read on.

7.2.1 Configuration File Syntax

The available tags and there nesting capabilities look like follows:

<!–MiroConfigurationDocument–>

configuration One file always describes one entire configuration. A configuration describes all the services parameters, that can be executed on the particular robot.

There can only be one configuration tag per file.

section Each component of *Miro* can have its own section within a configuration file. Configuration file sections are used to group services of similar scope. For instance the vision system, the configuration of the filter tree, as well as the the specification camera paramters, is considered to be a component within a *Miro* configuration file. Normally each service defines its own section within the configuration file.

There are multiple section tags permitted within a configuration.

Attributes:

- *name* The name of the section (required).

instance Parameters of a component are grouped and described as set of parameters that belong together. Note that within the configuration file there is a hardware centric view taken. That is, sensors and actuators that result in multiple service interfaces, are most likely to be grouped into one component if they are controlled by the same low level controller board. For instance the parameters for the pioneer board, that controls the motors as well as the sonars etc., are grouped within just one component.

An instance of a certain parameter set can be specified by the type and name attributes of the instance tag. There are multiple instance tags permitted within a section.

Attributes:

- *name* Name of the parameter (required).
- *type* The type name of the parameter. The parameter type definitions are described in section 7.3.

parameter Parameters can be specified within the instance tag, for each available parameter of described the type.

There are multiple nested parameter tags permitted, but each has to hold a name attribute that is unique within the enclosing tag.

Attributes:

- *name* Name of the parameter.
- *value* The value of the parameter.

In case the parameter describes a structured type instead of a basic type (like bool, int, string etc.), parameter tags can be nested. In this case the value attribute is ignored.

parameter For many parameter types there is a default instance available. Especially if there can only be one parameter set of this type per robot (i.e. the Robot parameters set, that holds the name of the robot etc.). This default instance is not referenced by the `instance` tag, but by a `parameter` tag, that holds the type name of the parameter set as its `name` attribute.

There are multiple parameter tags permitted within a section. Apart from the unified `type/name` attribute, a parameter tag behaves just like an `instance` tag, and can hold multiple, probably nested `parameter` tags.

Attributes:

- *name* Type name of the default parameter instance.

7.2.2 Example Configuration

See `$MIRO_ROOT/examples/params/TestConfig.xml` for a small initial example.

TODO: include and comment the example code here.

Section tags are structurising elements and component instance tags are mainly defined by the underlying hardware devices. But what defines which parameters are accepted within by which component and of what types they are? While the configuration files contain comments to explain the purpose of the different parameters, there do also exist formal definitions of the components parameter sets, as described within the next section.

7.3 Parameter Set Generation

Parameter sets, described as component parameters within a configuration file, correspond to strictly typed classes within C++. Those structures can be initialized by parsing the configuration file. To facilitate the generation of parameter sets, parameter classes needn't to be hand coded, but are auto-generated from parameter description files by a little parameter class compiler (`MakeParams`). The parameter description files are once again based on the XML syntax.

Furthermore the description files are not only used for the configuration of the services, but also by other tools within the *Miro* framework. The configuration editor used this information for the generic editing of configuration files. Additionally, the behaviour framework (sec. **??**) uses this functionality for parameterization of behaviours and arbiters and the `PolicyEditor` (sec. 12.8 use the parameter description files for GUI based editing of the behaviour parameters). To facilitate the reuse of the parameter descriptions by multiple tools, the syntax of the file is somehow obfuscated by additional tags and attributes.

7.3.1 Description File Syntax

Parameter classes support single inheritance.

<!–MiroParametersConfigDocument–>

config A parameters config file describes the types, names and defaults for parameter classes as used within the *Miro* framework.

> **config_group** Configurations can be grouped together for various reasons. A group within a description file in general corresponds to a section within the configuration file. Within a configuration file, a parameter set can only be instanciated within the section, named as the config group.
>
> Attributes:
>
> - *name* The name of the config group.
>
> > **config_item** A configurable item to built a parameters class for. Each config item will be mapped to a parameter class by the compiler.
> > Attributes:
> >
> > - *name* The name of the parameter class. The suffix Parameters will be added to the name.
> > - *parent* The name of the super class. The suffix Parameters will be added to the name. Note that namespaces have to be fully qualified here, i they differ from the current namespace.

- *instance* Whether or not a singleton instance of the parameter class shall be created (sec. [?]). Default is *false*.
- *final* Corresponds to the finalize keyword of java. It is only relevant for the editing tools, not for code generation. It indicates, that a toplevel instance of this parameter can be created.Default is *true*.
- *dummy* Sometimes you have to fake a parameter class to the editing tools that does not exist as such. No code will be generated for such a class. Default is *false*.

config_parameter A parameter for the class.
Attributes:

- *name* Guess what.
- *type* Allowed basic types are: bool, char, int, unsigned int, short, unsigned short, long, unsigned long, double and std::string. There are two more predefined types: Angle and angle. *Angle* corresponds to a Miro::Angle instance. It is an angle ϕ normalized to $-\pi \leq \phi \leq \pi$ while *angle* corresponds to a none normalized angle represented by a simple double. Note that while the internal representation of angles is in radiant, the values specified in the configuration files are in degrees. But you can also specify parameter classes described within a parameter description file as parameter type. Additionally, *std::vector*<> and *std::set*<> are supported. Note however that we are currently lacking support for specifying default values for these none basic types.
- *default* Default value as set in the constructor (optional).
- *measure* The measure the type represents, such as: mm, msec, **?**, mm/s, **?**/s (optional).
- *inherited true/false* This attribute is used to overwrite defaults for inherited variables within the constructor of the class. If set to true, no new member variable will be added to the class, and only the attributes name and default will be evaluated.

config_global Global configuration items those are mainly used for code generation. They specify things, that the compiler can not easily derive autonomously, such as additional include directives, etc.
Attributes:

- *name* The name of the config item. There are currently three items supported:

  namespace Specifies the namespace of the generated code. While multiple parameter classes can be described within one file, they all have to reside within the same namespace.

  include Specifies a user include directive.

  Include Specifies a system include directive.

  value The name of the namespace, the local or global include directive.

7.3.2 Example Description

See `$MIRO_ROOT/examples/params/Parameters.xml` for a small initial example.

TODO: include and comment the example code here.

7.3.3 MakeParams

The parameter description compiler has the following command line parameters.

-f <file> name of the input file (default is *Parameters.xml*)

-n <name> base name of the output file (default is *Parameters*)

-s <extension> extension of the generated source file (default is *cpp*)

-h <extension> extension of the generated header file (default is *h*)

-v verbose mode

-? help: emit this text and stop

7.3.4 Example Header File

See `$MIRO_ROOT/examples/params/Parameters.h` for a small initial example.

TODO: include and comment the example code here.

7.3.5 Makefile magic

To enable automatic generation of the C++ source and header files from XML parameter descriptions, there does allready exist a rule within the make files of *Miro*. Therefore, you can simply add the XML based description file to the `target_SOURCES` variable in the corresponding `Makefile.am`. Make sure however, that it is placed before the first source file, that includes the generated header file, to ensure its timely generation. Furthermore the resulting source and header files have to be added to the BUILT_SOURCES variable.

7.4 Configuration Management Runtime Environment

??

7.5 Config File Editor

??

For advanced config file editing, like filter trees of the video image processing framework **??**, a simple text editor is a fairly errorprone tool. XML editors do a better job in this case, but for real comfort, *Miro* provides the config file editor.

Chapter 8

Video Image Processing

Video image acquisition is one of the most common sensory devices used in robotics. Therefore *Miro* offers a common `VideoService` that can be adapted by the configuration file to any supported video device.

Also, image processing is computationally expensive and many standard filters exist in literature as well as some high performance implementations such as [**?**]. To facilitate the unified usage within robotics scenarios, the video service provides a filter tree framework.

Real-time image streams need a lot of bandwidth. This is usually higher than the bandwidth available on most mainstream network devices. Therefore the `Video`-interface of *Miro* also provides methods for usage of shared memory for sharing of images with its clients on the same machine. Those methods are also designed for minimizing copying overhead.

This section is organized as follows. First, the supported image devices along with their bugs and features are discussed. Then the filter tree framework is presented. First from an end user perspective and second from a developer perspective. In section **??** the video interface itself along with its two use cases and helper classes is discussed.

8.1 Video Device Access

The VideoService currently supports the following devices for images acquisition:

8.1.1 Bttv Frame Grabbers

These frame grabber cards are supported via video for linux []. This is the standard way of connecting standard analog video cameras to the computer.

Note, that the default number of frames used by the kernel driver for video capture are two. In order to get the full frame rate (25/30Hz), this has to be set to 4 buffers. This can be specified as a module parameter.

8.1.2 Firewire Digital Cameras

Miro supports the fire wire digital camera protocol using libraw1394[] and libdc1394 [].

Note that most cameras 1394 controllers do not support being bus master for dma transfers. Therefore the 1394 controller of the computer has to be the highest numbered node. This can be configured by a module parameter. Otherwise, unplugging the firewire cable from the camera for some seconds helps.

8.1.3 Matrox Meteor Frame Grabbers

These rather old frame grabber cards are also supported by *Miro*. Kernel drivers can be found at []. This device however, is mostly unmaintained within *Miro*.

8.2 Video Filter Trees

For image processing many standard filters do exist that usually perform a function $M_{n+1} = f_i(M_n)$. Where M_n denotes the input image (matrix), M_{n+1} the output image and f_i is the actual filtering function.

Often filter chains are used (see figure 8.1). I.e. first undistorting the image f_0 and then performing some color indexing f_2. So the filtering function producing the output image M_o from the input image M_i could be expressed as:

$$M_o = f_2(f_1(M_i))$$

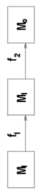


Figure 8.1: A simple filter chain.

More complex vision processing could also include the computation of an edge image M_e in addition to the color indexing above. This can be done i.e. by producing a grey image f_3 and applying a canny filter f_4, leading to the second filter chain:

$$M_e = f_4(f_3(f_1(M_i)))$$

To avoid computing the undistorted image twice, a filter tree should be used instead of two separate filter chains, as illustrated in fiugre 8.2.

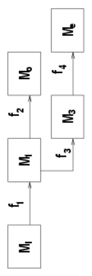


Figure 8.2: A still quite simple filter tree.

As an additional complication, many sophisticated filters use more then one input filter. For instance the Canny filter computes first two images of the Sobel operator (one for the x axis D_x and one for the y axis D_y). So the above described filter tree is actually an acyclic directed graph of filters, as illustrated in figure 8.3.

8.3 VideoService

The program provided by *Miro* for image acquisition and preprocessing is called `VideoService`.

Miro comes with a set of basic filters. The term filter is used here in a very general way, as almost every processing unit of the video service is designed as a filter. Actually the image acquisition from the different video devices is implemented as a filter, forming the first set of filters available in *Miro*. The second set of filters are basic image conversions, like byte order swapping, YUV to RGB transformations etc. The third set of filters are the classic image filters. Unluckily, there do not exist any of them currently in *Miro*. It introduces further library dependencies for the middleware, which we have to be careful about. And i.e. for Ipp based filters, we also have a licensing problem. How to extend the video service to process your own filters is covered in the following subsection.

Filter graphs and their individual parameters can be specified within the configuration file of the robot. The VideoService actually does not support every possible directed acyclic graph. The implications on control flow and synchronisation are too cumbersome. Instead the filters in the VideoService are organised as follows.

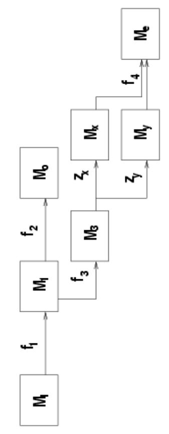


Figure 8.3: A not that simple filter graph.

The filters are organised as a filter tree. That is, every filter has one predecessor and n successors. The filtertree is processed in depth first order. This scheme is extended by so called filter links: A filter can specify additional input filters. To guaranty the correct order processing the filters, those additional filters have to reside before the specifying filter, in terms of the depth first evaluation scheme of the filter tree. Figure 8.4 tries to visualize this constrain.

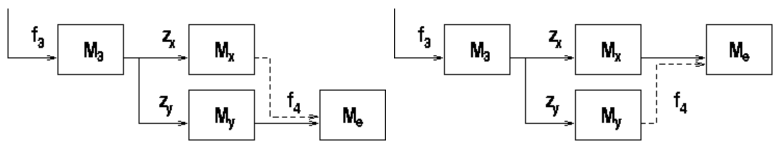


Figure 8.4: A legal vs an illigal filter graph.

8.3.1 VideoService Parameters

The video service expects its configuration parameters with the section Video of the configuration file. Its parameter is also called Video. The video service has the following parameters:

Width The width of the input image (unsigned int). It has to be supported directly by the used video device.

Height The height of the input image (unsigned int). It has to be supported directly by the used video device.

Palette The palette of the input image (string). It has to be supported directly by the used video device. The recognized palettes of the video service are:

> **grey** 8 bit grey values.
>
> **grey16** 16 bit grey values.
>
> **rgb** 24 bit RGB format.
>
> **bgr** 24 bit BGR format (byte swapping).
>
> **rgba** 32 bit RGB format (plus alpha channel).
>
> **abgr** 32 bit BGR format (plus alpha channel).
>
> **yuv** 24 bit YUV format.
>
> **yuv411** compressed YUV format from 1394.
>
> **yuv422** compressed YUV format from 1394.

Filter The root of the filter tree. The filter parameter has three entries:

> **Type** The filter type name (string).
>
> **Name** The name of the corresponding configuration file parameter entry (string).
>
> **Successor** The list of successor filters (Filter).
>
> **BackLink** The list of additional predecessor filters, specified by name (string). Note that this is the name of the filter instance, not the name of the filters video interface.

8.3.2 Video Filter Parameters

For each filter in the filter tree, filter parameters can be given within the parameter whose name matches the filter name given in the filter tree. Those are also located within the section Video.

The base parameter set of each filter is as follows:

InterfaceInstance Whether the filter has an instance of the video interface associated with it (bool). Each filter can have an instance of the video interface. That way every single node of the filter tree can be passed on to the client.

Interface The parameters of the interface. For each interface the following parameters can be specified:

> **Name** The name under which the interface is registered at the naming service (string). The default is Video. Note that if you have more than one Video object, you have to specify a unique name for each instance.

Buffers The number of memory buffers, reserved by the buffer (unsigned int). The default is 4. While a client is accessing an image, the video service guarantees, that it does not become overridden with a new image from the video stream. If clients hold multiple images (i.e. for temporal integration) it has to be made sure, that there is still a buffer left, to put the next image from the video stream into.

8.3.3 Video Device Parameters

As mentioned above, all supported video devices are implemented as filters. Therfore they inherit the base parameters. Note however, that the device filters are not capable of having an interface instance. Sharing memory mapped files is a bit tricky, and we didn't want to go into that, yet. The common DeviceFilter parameters are:

Device The fully qualified path of the video device (string). The individual device filters set this parameter to a common default (like /dev/video/0 for bttv and /dev/video1394/0 for firewire).

Common parameters for frame grabber based video devices are:

Format The analog format of the video signal. This is either `pal, ntsc, secam` or `auto`.

Source The source of the video signal. Available sources are `composite1, composite2, composite3, svideo` and `tuner`.

The available video device filters are (listed by type name):

VideoDeviceBttv The bttv device filter defines the following additional parameter:

- **Subfield** The subfield chosen (string). As the PAL and NTSC images are interleaved, the bttv frame grabbers can select, which of the half images to scan, if the height of the output image is half the height of the full image (384). Possible values are odd, even and all. Note however, that many frame grabber cards do not support this feature. In this case, use the FilterHalfImage described below.

VideoDevice1394 The firewire digital camera standard gives access to many of the internal camera parameters. These can be configured in the parameter section of the 1394 device. For all these values -1 denotes, that this parameter should be controlled by the camera.

- **Buffers** The number of image buffers used for dma transfers (unsigned int).
- **Brightness** (int).
- **Exposure** (int).
- **Focus** (int). Default is auto.
- **Framerate** (int). Default is 30Hz.
- **Gain** (int).
- **Gamma** (int).
- **Hue** (int).
- **Iris** (int).
- **Saturation** (int).
- **Sharpness** (int).
- **Shutter** (int).
- **Temperature** (int).

Trigger (int).

WhiteBalance0 (int).

WhiteBalance1 (int).

VideDeviceMeteor The matrox meteor device filter does not define any additional parameters.

VideoDeviceDummy A dummy device which can be used for offline testing. It reads an image specified by the *Device* parameter. If the *Device* specifies a directory instead of a file, all **.ppm* files of the directory are used by the device.

Timeout The image refreshing time (sec).

Cyclic (bool) The default is *true*. If true the device will restart with the first image after all images were loaded.

8.3.4 Basic Video Filters and Their Parameters

Miro also provided the following basic filters with the standard VideoService.

FilterCopy As the video device filters cannot map the image data directly into shared memory for access by the clients, this filter is provided. It copies the image internally, to allow for an interface instance.

FilterSwap3 Byte swapping for 24 bits per pixel images (BGR to RGB).

FilterSwap4 Byte swapping for 32 bits per pixel images (ABRG to RGBA).

FilterFlip We have a camera that's mounted upside down. This filter flips the image. Also useful if your fly imitating robot has successfully landed on the ceiling.

FilterHalfImage This is a filter to extract a half image from an interlaced image as the bttv frame grabbers provide. It copies each second scan line. It defines the following additional parameter:

Odd Take the second half image by starting with the second line (bool). Default is false.

8.3.5 Configuration Example

Putting it all together the configuration section for the `VideoService` might look like the following. This is actually the configuration of our Performance PeopleBot. It has an analog video camera and a bttv frame grabber card. The camera is mounted upside down.

```
<!-- The video configuration section. -->
<section name="Video" >

 <!-- Parameter section of the VideoService -->
 <parameter name="Video" >

  <parameter value="bgr" name="Palette" />         <!-- Input image palette. -->
  <parameter value="384" name="Width" />          <!-- Input image width. -->
  <parameter value="288" name="Height" />         <!-- Input image height. -->
  <parameter name="Filter">                       <!-- Filter tree root. -->
   <parameter value="DeviceBTTV" name="Type" />   <!-- It's a bttv device. -->
   <parameter value="DeviceBTTV" name="Name" />   <!-- Params section name. -->
   <parameter name="Successor" >                  <!-- Filter tree leafs. -->
    <parameter name="Filter" >
     <parameter value="FilterSwap3" name="Type" /> <!-- Byte swapping filter -->
     <parameter value="FilterSwap3" name="Name" />
     <parameter name="Filter" >                   <!-- Filter tree leafs. -->
      <parameter value="FilterFlip" name="Type" /> <!-- Upside down filter. -->
      <parameter value="FilterFlip" name="Name" />
     </parameter>
    </parameter>
   </parameter>
  </parameter>
 </parameter>

 <!-- Parameter section of the bttv device. -->
 <parameter name="DeviceBTTV">
  <parameter value="/dev/video0" name="Device" /> <!-- Path of the device. -->
 </parameter>

 <!-- Parameter section of the byte swapping filter. -->
 <parameter name="FilterSwap3">
  <parameter value="true" name="InterfaceInstance" /> <!-- Video interface. -->
 </parameter>

 <!-- Parameter section of the upside down filter. -->
 <parameter name="FilterFlip">
  <parameter value="true" name="InterfaceInstance" /> <!-- Video interface. -->
  <parameter name="Interface">                        <!-- Interface params. -->
   <parameter value="Flipped" name="Name" />          <!-- Interface name. -->
  </parameter>
 </parameter>

</section>
```

8.4 QtVideo

`QtVideo` is a test client for the `VideoService` and the `Video` interface. It displays an image stream from a `Video` interface instance and has an additional button, to save snapshots to disk.

The `QtVideo` tool accepts the following command line parameters:

-n Name of the `Video` interface instance within the CORBA naming service. Default is *Video*.

-r Remote access of the images. QtVideo will be using the methods for location transparent image access of the `Video` interface. These are much slower, than the methods using shared memory buffers, but are the only option when running QtVideo on another machine.

-v Verbose mode.

-? Emitting command line help and exits.

8.5 Video Interface

The `Video` interface is used to manage the access of image data by client programs. It supports location transparent as well as optimized local image access. Additionally connection management is used, to switch off filter subtrees that are not accessed by any other program.

The general access pattern of a client of a `Video` object looks like follows:

1. Get a `Video` interface IOR from the naming service.
2. Connect to the `Video` object.
3. Get images until done.
4. Disconnect from the `Video` object.

Note that due to the performance centric design of the Video interface, the VideoService can easily be jammed by client programs violating the access protocol of the interface. To facilitate the correct usage, some helper classes are provided by *Miro*. These are discussed in section **??**.

8.5.1 Location Transparent Image Access

To access an image as a client running on a different computer than the `VideoService`, the `Video` interface offers the methods

- `exportSubImage`
- `exportWaitSubImage`

They both return a copy of the image as return value. The first method returns immediately the current image, while the second one waits until a new image becomes available before returning. The image will be scaled down to the size specified by method parameters.

Note, that due to the copying and network overhead of those methods, they are only useful for debugging and monitoring purposes.

8.5.2 Local Image Access

For clients running on the same machine as the `VideoService`, the `Video` interface offers the following methods for image access via shared memory buffers:

- `acquireCurrentImage`
- `acquireNextImage`

- `releaseImage`

While the first method returns immediately the buffer of the current image, the second one waits until a new image becomes available before returning. Note that the clients have to release each image buffer after processing. Otherwise the VideoService will soon run out of buffers, to share new images with its clients.

8.5.3 C++ Helper Classes

To facilitate the usage of the `Video` interface and the adherence of the connect/disconnect, acquire/release protocol, *Miro* provides two simple helper classes. They are defined in the file `$(MIRO_ROO)/src/miro/VideoHelper.h`

Miro::VideoConnection This class connects to a `Video` interface instance on construction and disconnects on destruction. The constructor takes a pointer to the `Video` object, to connect to as argument.

Miro::VideoAcquireImage This class acquires an image from a `Video` interface instance on construction and releases it on destruction. The constructor takes a reference to a `Miro::VideoConnection` object, as first argument, the second selects whether the current or the next image is to be acquired.

8.5.4 Example Video Client

Listing 8.1: examples/video/VideoExample.cpp

```
#include "idl/VideoC.h"
#include "miro/Client.h"
#include "miro/VideoHelper.h"

#include <iostream>

using std::cout;
using std::cerr;
using std::endl;
using std::flush;
using std::cin;


int main(int argc, char * argv[])
{
  int rc = 0;

  Miro::Client client(argc, argv);
  try {
    Miro::Video_var video =     // Get reference to video service.
      client.resolveName<Miro::Video>("Video");

    cout << "connection" << endl;

    Miro::VideoConnection connection(video.in());  // Build up connection.
  cout << "local␣copy" << endl;

  // Get a local copy of the current image ,
  // using full resolution .
  CORBA::ULong x = connection.handle->format.width;
  CORBA::ULong y = connection.handle->format.height;
  Miro::SubImageDataIDL_var image1 =
    video->exportSubImage(connection.handle->format.width,
                          connection.handle->format.height);

  cout << "local␣copy␣next" << endl;

  // Get a local copy next image ,
  // scaled down somehow .
  x = connection.handle->format.width / 2;
  y = connection.handle->format.height / 3;
  Miro::SubImageDataIDL_var image2 =
    video->exportWaitSubImage(x, y);

  cout << "acquire␣current" << endl;

  // Acquire the current image buffer .
  Miro::VideoAcquireImage image3(connection,
                                 Miro::VideoAcquireImage::Current);

  cout << "acquire␣next" << endl;

  // Acquire the next image buffer .
  Miro::VideoAcquireImage image4(connection,
                                 Miro::VideoAcquireImage::Next);

  cout << "clean␣up" << endl;

  // Automatic resource cleanup by object destructors :

  // - image4 destructor releases its buffer to the VideoService .
  // - image3 destructor releases its buffer to the VideoService .
  // - image2 destructor releases the heap memory of its image copy .
  // - image1 destructor releases the heap memory of its image copy .
  // - connection destructor disconnects the client at the VideoService .
  // - video destructor destroys the video interface proxy object .
}
catch (const Miro::ETimeOut& e) {
  std::cerr << "Miro␣Timeout␣Exception:␣" << e << endl;
  rc = 1;
}
catch (const Miro::EDevIO & e) {
  std::cerr << "Miro␣Device␣I/O␣exception:␣" << e << endl;
  rc = 1;
}
catch (const Miro::EOutOfBounds & e) {
  std::cerr << "Miro␣out␣of␣bounds␣exception:␣" << e << endl;
  rc = 1;
}
```

```
  catch (const CORBA::Exception & e) {
    std :: cerr << "Uncaught␣CORBA␣exception:␣" << e << endl;
    rc = 1;
  }
  return rc;
}
```

8.6 Video Broker Interface

In addition to the Video interfaces of individual filters, the VideoService also offers an interface that accesses the whole filter tree of the service. It is called VideoBroker and also registers as VideoBroker at the NamingService. It offers methods for synchronised simultanious access to multiple filters and for inspection of the filter tree.

8.6.1 Synchronised Image Access

The client side evaluation of the results of the various filters often requires access to multiple filters, that originate from the same input image. The following methods are available in the VideoBroker interface for this purpose. They work quite like the corresponding methods of the Video interface of the individual filters.

- `acquireNextImageSet`
- `releaseImageSet`

8.6.2 Filter Tree Meta Information

Statistical information about the filter tree can be optained at the VideoBroker interface via the method:

- `filterTreeStats`

It returns the number of connections, the processing time of the filter as well as the processing time of the corresponding filter subtree (not including the processing time of the filter), the name of the filter, the ior and the name of its Video interface (if available), the filter successors as well as the successor links.

8.7 Writing Filters

Sooner or later, it will become handy for a user to add custom filters to the video image processing framework. This is a two step process. First the filter has to be implemented as a class derived from the `Filter` base class. Second a customized version of the `VideoService` has to be built. The later has two reasons. The first is, that *Miro* currently does not contain a plug in architecture for dynamically loadable modules. The second is, that dynamic loadable modules require the usage of shared libraries. As this is has some runtime performance impact, that might not really desirable for performance critical tasks like image processing. Nevertheless, patches are welcome.

8.7.1 The Filter Base Class

The base class of all filters is `Video::Filter`. It provides methods for filter tree setup and configuration, a harness for recursive filter tree processing and buffer management in interaction with a `Video` interface instance.

The most important protected methods and data members for child classes are.

inputBuffer() Returns a pointer to the input buffer.

inputFormat_ Specifies the data format of the input buffer.

outputBuffer() Returns a pointer to the output buffer.

outputFormat_ Specifies the data format of the output buffer.

8.7.2 Methods to Overwrite

The most important methods for child classes are.

Constructor The constructor takes a struct of type `Miro::ImageFormatIDL` as parameter. It describes the input format of the filter (hence the output format of the predecessor filter). The constructor of the `Filter` base class copies this parameter into the two member variables `inputFormat_` and `outputFormat_`. The derived filter has to ensure, that the input format is a valid format for the filter and modify the member `outputFormat_`, to correctly describe the output format of the filter. If the input format is not valid a `Miro::Exception` has to be thrown.

process The actual filtering method. Put your filter code here.

8.7.3 Configuration and Parameter Processing

If your filter needs additional parameters for configuration, your can add them to the configuration file framework by specifying them in a parameter file description, deriving them from `Miro::FilterParameters`. You have to overwrite the factory methods for `Miro::FilterParamters` in the `Filter` class to return an instance of your derived parameters class. This can be automated by using the macros:

FILTER_PARAMETERS_FACTORY(X) For usage within the class definition.

FILTER_PARAMETERS_FACTORY_IMPL(X) For usage within the class implementation.

Where X is the name of your derived Filter class.

For initialization and cleanup the following two methods exist.

init Called on initialization of the filter tree. It provides a pointer to the parameters object initialized from the configuration file. Use `dynamic_cast` to convert it to the type of your derived parameter class.

fini Called before destruction of the filter tree.

8.7.4 Enabling and disabling features

The interface also contains some flags that allow the derived classes to specify the existens of certain properties for the base class. Overwrite their settings in the constructor. Those are namely:

interfaceAllowed Specifies, whether the filter result can be exported by a video interface. The default for Video::Filter is true. For Video::Device it is false (see section **??** for details).

inplace Not implemented yet. (Allways false)

8.7.5 Filter Meta Information: FilterImageParameters

Sophisicated filters may also extract additional information from the images, such as a region of interest etc. To hand this information to subsequent filters, an additional interface does exist. It is based on the same mechanisms as the filter parameters, except that they can't be specified within configuration files. The base class is empty.

The factory methods also reside in the corresponding `Filter` class and are wrapped by the two macros:

IMAGE_PARAMETERS_FACTORY(X) For usage within the class definition.

IMAGE_PARAMETERS_FACTORY_IMPL(X) For usage within the class implementation.

Where X is the name of your derived Filter class. The base class is named `FilterImageParameters` and the macros expect the name of any child class to end on `ImageParameters`.

The `Filter` interface provides accessor methods for the parameters of the individual buffers:

inputBufferParameters Returns a const pointer to the parameters of the input buffer.

inputBufferParametersLink Returns a const pointer to the parameters of the linked input buffer specified by its index.

outputBufferParameters Returns a pointer to the parameters of the output buffer. Note that the parameter instances is not initialized with defaults before processing of the filter. That is, the filters `process` method is responsible, that all data members of the image parameters instance contain meaningfull values on return.

8.7.6 Example Video Filters

To illustrate the framework, a simple example filter is provided under `$(MIRO_ROOT)/examples/videoFilter`.

8.7.7 Calculating a Gray Image

`FilterGray` implements a simple gray filter for images in 24 bit rgb format. The gray value is calculated as the average of the weighted sum of the three color values. The weighting can be specified by the filters parameters.

Start the example service in the directory videoFilter with
`GrayVideoService -MiroConfigFile GrayVideoConfig.xml`

It registers two interfaces at the naming service: `Video` and `Gray`. To examine the original and the filtered image, use `QtVideo` and `QtVideo -n Gray`.

8.7.8 Image time series

`FilterDiff` illustrates the use of multiple input filters. It calculates the difference of two input images. For that purpose it locks images of one image source to provide a view into the past. The number of images locked can also be specified by the filters parameters. Note however, that it has to be less then the number of output buffers provided by the input buffer.

Start the example service in the directory videoFilter with
`GrayVideoService -MiroConfigFile DiffVideoConfig.xml`
Note that the usage of an input filter link is a bit artificial in this configuration, as both input filters are the same.

It registers two interfaces at the naming service: `Video` and `DiffImage`. To examine the original and the filtered image, use `QtVideo` and `QtVideo -n DiffImage`.

8.8 Writing a new Input Device

The root of a filter tree is a video device. A video device is actually not an image filter but an image producer. But as mentioned above, it is implemented as as specialization of the video filter class. Some additional requirements and features do exist to handle this case.

8.8.1 BufferManger

The video framework does use a buffer manager class to organize the concurrent acces to the images of a filter by the filters and remote clients. Video devices normally aint capable of having directly a video interaface instance attached. The video interface uses shared memory to allow for zero copy acces to the clients. Video devices often use memory mapped io to acquire the images. Combining the two is at least tricky, if possible at all. So most video devices just skip it.

Instead the buffer manager class is used for synchronising the access to the mmaped images between the filters and the hardware.

Chapter 9

Group Communication in Robot Teams

When it comes to multi-robot scenarios, group communication becomes an issue of interest. As *Miro* is designed for usage in distributed environments, most of the prerequisites for communication in teams of robots are in place. But in the presence of WLANs, that have extremely varying bandwidth, that dependents on the location of the computer etc. some more requirements have to be met. Especially the TCP/IP based transport protocol can become a bottleneck, when temporary network breakdowns can regularly happen during operation.

9.1 Event Channel Federation

The event based communication of *Miro* is based upon the CORBA Notification Service. In a team of robots, one central event channel could be set up and used for team communication. While this is most convenient from a user perspective, it creates a single point of failure for a team of otherwise independent autonomous robots.

Using a single virtual event channel, that is distributed over multiple channels, that run on the different robots helps both, the programmers, as well as the systems fault tolerance. As such an event channel federation is not part of the specification of the CORBA Notification Service, *Miro* provides its own implementation. Namely the Notify Multicast (NMC) module. Its implementation is based upon the work done for the event channel federation of the RT-EC of TAO [2]. A successful application of the NMC module is described in [11].

9.2 Notify Multicast

One notification service is run on each robot (computer). The NMC module attaches one consumer and one supplier to the robots local event channel. Additionally it subscribes at a multicast group. Events that are subscribed by other robots are exchanged via the multicast group.

For the automatic arbitration of the set of events that need to be exchanged via the multicast group, the offer/subscription protocol of the Notification Service is used.

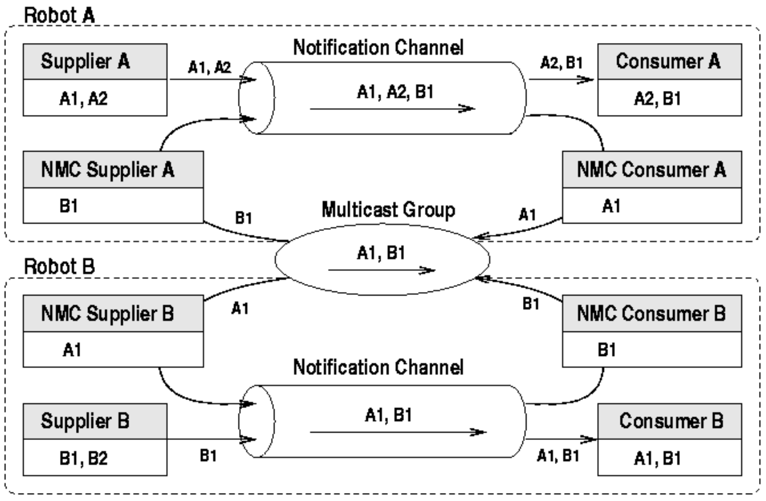


Figure 9.1: A Federated Notification Channel Setup

Offers to and subscriptions at the event channel are determined by the `domain_name` and `type_name` fields of the event header. In *Miro* the domain name is by convention the name of the robot. The NMC module posts all events that subscribed for at the local channel, but that are not offered locally to the multicast group and collects these posts from the other robots. Events that are requested by other robots and offered locally, are posted to the multicast group on arrival on the local event channel. Those events, that are subscribed for at the local channel, but not offered locally, are published to the local channel as they become posted on the multicast group.

See also figure 9.1 for a possible configuration of subscriptions and offers in a federated event channel.

9.3 Usage

The NMC module is part of most robot servers by default. It can be enabled by the command line option `-MiroNotifyMulticast` or `-MNMC` for short. It also exists, as a separate program: NotifyMulticast.

9.3.1 Parameters

The parameters can be set in the robots configuration file. They are located in the configuration group `Notification`. The available parameters are.

MulticastGroup The multicast group used for group communication.

Timeout The maximum age for events send over the group. This requires proper synchronization of the team clock (i.e. by NTP [3]). A value o zero indicates, that all events should be processed regardless of their age.

Subscription These events are shared with other robots event if no one seems to listen. - Events that are exchanged on the bases of the offer/subscription management protocol need some time, until the request is acknowledged by the suppliers. By providing events with this option, this time can be skipped. Note, that only the type name is specified here. The domain name is assumed to be the robots name.

Chapter 10

Logging

The logging of debug, information and error messages can help a great deal in the recovery from failure situations. This is especially true for robotic applications. Unfortunately the sheer mass of console output from the different modules can become too complex quite fast. So *Miro* comes with a set of logging facilities that cover different levels of the system functionality. The first framework, that is introduced within this chapter is based on the logging framework provided by ACE. It is a system log oriented framework to organize printf like output on various levels. Chapter 11 will cover another, more sophisticated set of logging functionality.

10.1 Log Levels and Categories

Miro uses two mechanisms to customize logging output. Log levels are used to give fine grained control over the verbosity of log output. *Miro* defines log levels from 0 to 9. The higher log levels are considered to be used for debugging purposes only. They are therefore also referred to as debug levels. They usually are only meaningful for developers. For debug levels log categories are used to restrict the logging output to one or more subsystems of the robot application. Output from debug levels is only displayed if its log category is enabled, too. Additional log categories can be defined for your own programs.

The different log levels are:

0. *Emergency* Log level of messages reporting an emergency. Your robot is on fire etc. This log level is not maskable, except if you turn of logging at configure time.

1. *Alert* Log level of messages reporting an alert. This is when the red lights start blinking and this unnerving honking sound is played.

2. *Critical* Log level of messages reporting a critical condition. This usually is an unrecoverable error, that leads to the termination of reporting program.

3. *Error* Log level of messages reporting an error. This indicates a real error, but the program will usually try to continue anyway.

4. *Warning* Log level of messages reporting a warning. A warning should be fixed, but the program is likely to work anyways.

5. *Notice* Log level of messages reporting a notice. Make a post-it and add it to the other 500 ones.
6. *Ctor_Dtor* Debug level of messages reporting a constructor/destructor entry. This debug level is designed to hunt segmentation faults on startup and exit. — This is when all the big ctors/dtors are run.

The debug levels are:

7. *Debug* Log level of messages reporting debug output.
8. *Trace* Log level of messages reporting program trace output.
9. *Prattle* Log level of messages reporting really verbose comments on the program execution.

The different log categories used within *Miro* are the following. (All brand names have to be marked as such, as soon as we find the time):

Miro This category is used by the *Miro* core components, namely those located in the library libmiro.so

Video Used by the components of the video image processing system. See section 8 for details.

NMC The notify multicast based event channel deliberation, as used for team communication. See section 9 for details.

B21 Used by the components exclusive to the B12 robots of RWI.

Pioneer Used by the components exclusive to the Pioneer robots or Active Media.

Sparrows The custom built Sparrows platform.

Faul The motor controllers from Faulhaber.

DP The pan-tilt unit from DirectPerception.

Sick The sick laser scanners have their own category.

Sphinx The sphinx speech components.

Dtlk The DoubleTalk speech components.

BAP The reactive control subsystem logs messages within this category. See section 12 for details.

10.2 Run-Time Configurability

The log level and log categories that are displayed can be configured at program startup by command line parameters.

-MiroLogLevel <n> or -MLL <n> for short, selects the log level up to which data is logged. The log levels are referenced by number. The default log level is 4. Log level 0 can not be masked by command line parameters, but only by turning off logging entirely at compile time.

-MiroLogFilter <name> or -MLF <name> for short, selects a category to log. This option can be specified multiple times to enable multiple categories. The category enabled by default is *Miro*.

10.3 Usage in Source Code

This logging facility is used extensively in the *Miro* sources. It can also be used and extended for the usage in robotic projects based on *Miro*.

10.3.1 Miro::Log

The class `Miro::Log` is defined in `miro/Log.h`. It works mostly as a namespace for the logging facility. It holds constants for all log levels as well as log categories. The main method is `Miro::Log::init()`. This is a static method that takes *argc* and *argv* as arguments, to parse for the `-MiroLogLevel` and `-MiroLogFilter` command line options. Additionally, accessor methods and predicates are defined to query log levels etc. at runtime.

10.3.2 Macros

`miro/Log.h` also defines a set of macros that are usable for development with, and within *Miro*.

MIRO_LOG Produce some log output, if the required log level is enabled. It takes two arguments. The first is the log level, the second is a single character string. The log level is specified by name. The canonical name format for log levels is the log level name in capitals, prefixed by `LL_`.

MIRO_LOG_OSTR Like `MIRO_LOG`, but the second parameter is used as right hand side argument for an output stream operator ≪. So it can contain some operator ≪ concatenated expression.

MIRO_DBG Produce some log output, if the required log level as well as the log category is enabled. It takes three arguments. The first is the log category, the second the level, and the third is a single character string. The canonical name format for log categories is the log category name in capitals.

MIRO_DEBUG_OSTR Like `MIRO_DBG`, but the last parameter is used as right hand side argument for an output stream operator ≪. So it can contain some operator ≪ concatenated expression.

MIRO_LOG_CTOR For constructor tracing. Accepts a single parameter containing the class name as string. Logged with log level 6.

MIRO_LOG_DTOR For destructor tracing. Accepts a single parameter containing the class name as string. Logged with log level 6.

MIRO_DBG_TRACE For method call traces. Takes the log category as argument.

MIRO_ASSERT The standard assert macro. It is provided to enable disabling of assert macros in (inline) code of *Miro* without disabling them for user code too.

10.4 Compile-Time Configurability

Log and debug messages can be disabled entirely at compile time by two configure flags:

–disable-DebugInfo disables debug information, that is namely log levels above 6. The `MIRO_DBG_...` macros will be replaced by no-op implementations, removing footprint and performance overhead of the programs, introduced by debug information. - Especially the performance overhead is mostly negligible, so this option might well be omitted even for release versions.

–disable-LogInfo disables log information entirely. Like above, this can save some footprint but is hardly measurable in performance.

10.5 Test and Example Programs

The programs located at tests/log provide some testing facilities for the logging facility and can serve as a practical example on the setup of the logging facility as well as on the usage of the logging and debug macros. The only program currently located there is TestLog.cpp. The source code is documented to help understanding.

Chapter 11

Event Channel logging

The acquisition of data during the run of a robot application from various levels of sensor processing is an essential feature for debugging, evaluation and performance assessment. The event based communication paradigm of *Miro* is designed to distribute raw sensor data as well as higher level system events, like the latest belief state of the robot. The stream of events resembles therefore a quite complete trace of the system state during the run of a robot program. *Miro* provides functionalitiy to log such an event stream generically to a file. The data can not only be reread into the system, but the events can also be redistributed over the event channel, allowing for detailed offline analysis. In [10] the various configuration and application scenarios are described along with a detailed performance analysis of the facility. This chapter is concerned with a more technical view, like the setup of the logging client, the replay as well as the maintenance of logged data and the file format.

11.1 Logging Client

The logging client for the notification service ist called `LogNotifyConsumer` and is located in the service library `libmiroSvc`. It can be used as a normal event channel client within the own application or as a standalone program.

11.1.1 Parameters

The location of the log files can be defined by the environment variable `MIRO_LOG`. The default file name is defined as:

$(MIRO_LOG)/¡domain name¿-¡time string¿.log

where ¡domain name¿ is the name of the robot and ¡time string¿ is the time the recording of the log file was started in the format *yyyy.mm.dd-hh.mm*. An alternative file name can be specified at the command line or as constructor parameter respectively.

Other parameters can be specified in the config file. They are located in the section Notification. Those are:

MaxFileSize The maximum size of the log file. Log files can become quite large over time. So it is good to have an upper limit. Also, the implementation uses a memory mapped file for maximum throughput. So the log file, along with the application should fit into the available physical memory. Otherwise swapping will jeopardize the overall system performance. The default is 150 MB. This is sufficient for 20 minute runs of our soccer robots, logging almost everything that is happening.

TCRFileSize The log file contains a type code repository, that holds descriptions for all types, stored within the log file (CORBA type codes, to be exact). The default maximum size for the type code repository is 1 MB. As a the number of different payload types is usually limited for one robot (about a dozend) and the type code size is between 0.5 – 1.5 KB, this should be sufficient for most applications.

Event A vector of domain name, type name pairs that shall be logged.

TypeName A vector of type names that shall be logged. The domain name is supposed to be the domain name of the robot. Usually a robot only logs its own events.

11.2 Standalone Logging Client

The LogNotify client is a standalone program, that can be used as an ad hoc solution for logging data. It offers the possibility to connect directly to a selected event channel, or use the IP-multicast based event channel federation described in section 9.

11.2.1 Command Line Parameters

The command line parameters are the following:

11.3 LogPlayer

The `LogPlayer` is a GUI-based application for the replay of logged data. It offers timely replay, slow motion and single stepping as well as simple maintenance operations like cutting and event filtering. It can also load multiple log files and play them synchronized, distributing the events over mutliple event channels. A screenshot of the `LogPlayer` can be seen in figure **??**.

The main panel consits of five button, a digital clock, a slider bar and a dail. The clock shows the time stamp of the coursor within the file. The slider shows the relative positon of the cursor within the log stream and can be used to repositon the cursor. The dail resembles the replay speed. Twelve o'clock is the original timing. Counterclockwise is slow motion and clockwise fast forward.

11.3.1 Main Panel

The buttons on the main panel do the following:

Stop Stop the replay of the log file and reset the coursor to the beginning of the log stream.

Play Play the log file from the current cursor position in the speed selected on the dail.

Pause Pause replay. The cursor position remains unaffected.

Forward Single step one event forward.

Backward Single step one event backward.

11.3.2 Menu

The menu bar has five entries: File, Edit, Events, Tools and Settings. The file menu offers file operations. The edit menu offers simple cutting functionality. The events menu allows to filter certain events from the log file. The tools menu holds tool windows for further log file event inspection and the settings menu holds the configuration.

File Menu

Open/Close... Displays the file set dialog that allows to add and remove log files, from the set.

Save as... Allows to save the current log file set, with event filtering and cutting applied to a new file.

Edit Menu

Cut Front Removes everything from the beginning to the current cursor position from the log file.

Cut Back Removes everything from the current cursor position to the end of the log file.

Undo all Undo all logfile cutting.

Events Menu

The events menu holds an entry for every robot, that was found in the currently loaded log file set. That is, for every distinct domain name. For each domain name a submenu with this robots events in the event stream is available. That is, every distinct type name. Each type name entry is checkable and can be switched on and off. Each unchecked event type is skipped on replay of the log file.

Tools Menu

This menu offers the event view, which is a window, listing a sequence of events around the coursor position. This is uesefull for event exact cut operations etc.

Settings Menu

The only setting currently available at the menu is the length of the history of the event view. - More to come — promised.

11.4 File Format

The file format consists of a header block of 8 bytes: The first four represent the log format magic cooky: MLOG. The next two are the version number in intel byte order. Followed by a two byte byte order flag, as defined by the CORBA common data representation format.

```
struct LogHeader
{
  unsigned long id;          //          MLOG      : 0       x474f4c4d
  unsigned short version;       //          Version      3: 0      x003
  unsigned short byteOrder;     //    Host          byte           order
};
```

The current version of the log file format is 3. In basic, all following data, including the events are stored in CORBA CDR stream format. What follows is in general a variable length array (CORBA sequence) of type:

```
struct EventEntry;
{
  TimeT timeStamp;
  unsigned long eventSize;
  CosNotification :: StructuredEvent event;
};

typedef sequence<EventEntry> EventArray;
```

The timeStamp field denotes the time, the event was received by the logging consumer. It is used for the timely replay and synchronization of log files.

The eventSize field denotes the size of the event field within the log file. This is used to speed up initial parsing of log files for the `LogPlayer`.

The event field contains the event, as delivered by the event channel. As a footpring optimization, the type codes, stored with every CORBA::Any, that is, remainder_of_body field of the events are stored as

```
typedef sequence<TypeCode> TypeCodeArray;
```

at the end of the log file, forming the type code repository. The type code in the remainder_of_body field is replaced by the index of the type code within the type code array.

The offset of the type code sequence within the log file is stored at the beginning of the CDR stream. That way the type code repository can be read before parsing of the events. This defines the log file format version 3, as follows:

```
struct LogFile
{
  LogHeader header;
  unsigned long tcrOffset;
  EventArray events;
  TypeCodeArray tcr;
};
```

11.5 Test and Example Programs

The programs provided in the tests directory for the LogNotification format, are in basic two configuration files, that restrict the size of the log file and the type code repository, to provoke overflow of the log files.

A utility program for log files is LogTruncate. If a log client dies discracefully with a segfault, the log is still saved and usable, but has the size specified by the MaxFileSize parameter, regardless of the actual footprint of the stored events. LogTruncate, removes the unused 0 bytes and truncates the log file to a reasonable size.

Chapter 12

Behaviour Engine

Controlling the actuators of an autonomous mobile robot is one of the central aspects of mobile robot research. As *Miro* exposes the interfaces to the motor controllers etc. of the mobile platform it enables researchers to easily evaluate new approaches to model sensor actor control loops. Be aware that at this point the latencies introduced by various levels of the robot architecture can become a critical issue. To eliminate the network latency and prevent yourself from occasional network bandwidth problems we recommend that you run your control programs collocated on the same computer with the service that accesses the actuators device.

Miro supports the behavioural control paradigm introduced by Brooks [1] by its own behaviour engine. It is designed to allow for a quick start into behaviour robotics writing your own behaviours. Yet, due to its open and extensible design it is also capable of handling sophisticated control tasks as demonstrated by its use within the RoboCup-scenario by our *Sparrow* an *Sparrow-2003* robots.

12.1 The Concept of Behaviours

The basic idea of the behaviour approach to robot control is as follows. Instead of the sense-plan-act paradigm of classical AI, the task is splited into a set of reactive behaviours, that each try to fulfill a small subtask of the problem set. For each of the tasks only a very limited part of world modeling is needed (often, even raw sensor readings are sufficient). By combining the output of the various behaviours by an arbiter, the emergent higher level behaviour of the system is achieved, solving the recommended high level task.

An accepted bottleneck of this approach is the arbitration and calibration of large sets of behaviours necessary to fulfill different aspects of a high level task. Therefore in the *Miro* framework the behaviour engine also support the hierarchical decomposition of behaviour sets by allowing them to be grouped in so called action patterns that can be activated alternately by so called transition messages. A set of action patterns is called a policy within the *Miro* framework.

There do exist two different kinds of transitions within the behaviour framework: local and global transitions. Local transitions are emitted by the behaviours by name. For example a transition named “GoalReached” might be emitted by a behaviour when it decides, it has fulfilled its task. Within an action pattern a transition is linked to the successor pattern, that will be activated as the transition message occurs. That way totally different action patterns can be activated, when the behaviour emits the “GoalReached” transition message within different action patterns. Global transitions are somehow simpler. They contain directly the action pattern, that is to be activated next. This mechanism is designed for modules of the robot, that are external to the behaviour framework. For example if a task planner decided it would be time for the robot to go home, it might activate the “GoHome” pattern just by issuing a global “GoHome” transition. The robot will then switch to its GoHome pattern regardless of what it was doing before.

12.2 Introductory Examples

Lets look at the simple action pattern, who’s high level task is to explore the environment by performing a random walk. This is easily splited into two distinct behaviours. The first subtask is not to collide with the environment. This can be achieved by an avoid behaviour, that reads the front sonar sensors to determine how far it is away from the nearest obstacle. If the minimal distance is below some threshold, it tells the arbiter to turn away into another direction. The second behaviour would be a wander behaviour. It selects from time to time just randomly some translational and rotational velocity, making the robot move around. It does not have to care about obstacles, since those are taken care of by the avoid behaviour.

The task of choosing the actual velocities to be applied to the motors is performed by the arbiter. It therefore plays a central role in the behaviour approach. There exist various kinds of arbiters, all choosing different policies for this task. But a simple priority based arbiter suffices for many scenarios. In *Miro* there currently just exists a priority based arbiter, but since it is an extensible framework you can easily plug in your own one. The priorities in this example could be applied straight forward. The avoid behaviour has higher priority as the wander behaviour.

Note the easy extensibility of this approach. For example, if we have some bump sensors that indicate collisions with the environment. We could just add another, let’s call it “EmergencyStop” behaviour and assign it the highest priority. If one of the bumpers is pressed it makes the robot stop and wait for rescue by one of the operators.

As a further extension we assume we have two tasks. The first is the random walk described above, the second would be a wall following behaviour, that simply is capable to drive the robot in a defined distance along a wall. The decision what to do is provided by an external source, say, a button located on the robot and pressed by the operator when demonstrating the management the capabilities of the newly bought autonomous mobile robot. The second action pattern looks quite like the first one, except that the wander behaviour is exchanged by a wall following behaviour, that drives the robot along the wall. (Note that this behaviour does not have to care about walls thar are blocking the way at the end of the corridor, since this can be taken care of by the avoid behaviour.) These two action patterns now form a policy. The pressing of the button sends a transition message, that disables the currently running action pattern and enables the other one.

The action pattern / transition message mechanism also fits naturally for coupling reactive behavioural control with deliberative planning architectures. The transition could also be raised by a path planner and the corresponding action patterns could be ’move to point’, ’drive through door’ or ’dock at power supply’. Indeed this was already done within [9].

12.3 Example Usage

As an example of the behaviour framework in use, let us start the obstacle avoid demo.

- Start all the robot main services from within `$MIRO_ROOT/bin/` with:
  `PioneerBase`
  This server has to run on the robot.

- Start the Behaviours, specifying the policy at the command line:
  `Behaviours ../examples/behaviours/PpbStraight.xml`
  This client can run on any computer, however, it is a good idea to run the behaviours on the robot, to avoid that network latencies stop the control loop.

- Start the policy controller GUI from within `$MIRO_ROOT/bin/` with:
  `PolicyController`
  to start and stop the policy run by Behaviours. Run this from any computer.

12.3.1 The Behaviour Control Loops

The core of the behaviour framework gets instantiated and initialized in the file `Behaviours.cpp` in `$MIRO_ROOT/examples/behaviours/` with its `main()` procedure. It instantiates all available behaviours and arbiters starts a thread for the timer service (ACE_Reactor) needed by timed behaviours (see section 12.5.1) and runs the CORBA event loop for its BehaviourEngine interface, as well as for the event consumers like event behaviours (see section 12.5.2) that get pushed by `PioneerBase` for instance. Behaviours themselves do not have an own `main` procedure. They are implemented in the `action()` method of ordinary C++ objects which have to be known in `Behaviours.cpp`. Necessarily, all behaviours run on one computer.

As seen in Fig. 12.1, `Behaviours.cpp` runs two loops synchronously: the timer loop for timed behaviours and the CORBA loop. The timer loop is started in a separate thread (invoked by the command ”task→open(NULL);”). It schedules the timed behaviours by invoking their `action()` methods one after another.

The CORBA loop of `Behaviours.cpp` (invoked by the command ”server.run(5);”). processes requests for the BehaviourEngine interface (which it registers under the name ”BehaviourEngine” at the CORBA Naming Service) from the PolicyController, as well as it administers all CORBA requests from `PioneerBase` and invokes the event behaviours. The `action()` method of an event behaviour is invoked whenever the CORBA notification service publishes an event to which the behaviour has subscribed. Events are subscribed to by their names, e.g. ”Tactile” or ”Sonar” (see `TactileStop.cpp` or `SonarNotify.cpp`). The published sensory events and their associated payload are described in section 4.3.

Invocation of an `action()` method by the event ”Tactile” is reminiscent of a call to a server registered as ”Tactile” by a client, e.g. `PioneerBase`. However, there are a few differences: *(i)* the event ”Tactile” calls not a program with its own `main()`, but only an `action()` method, *(ii)* not one unique method is invoked, but all `action()` methods which have subscribed to that event and *(iii)* unlike proper services, the TAO `nslist` utility does not list these event consumers, because they cannot be meaningfully invoked by an external program.

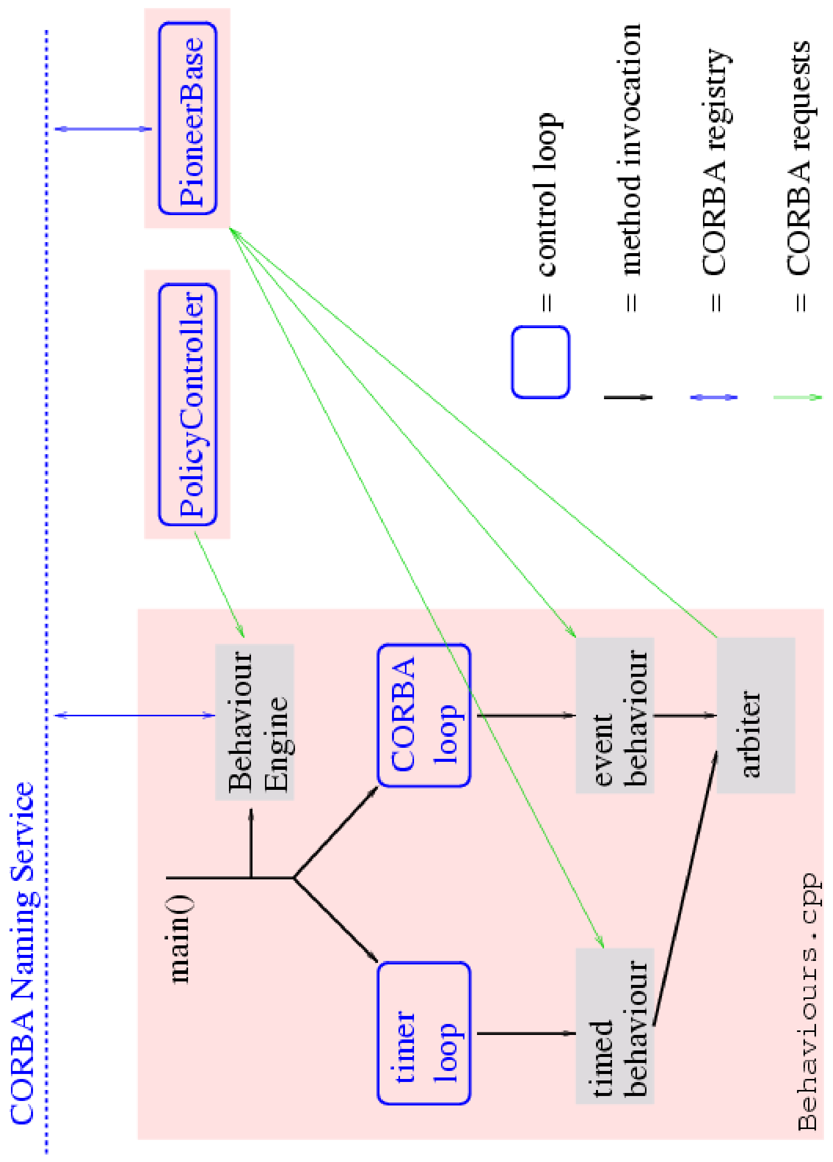


Figure 12.1: The Behaviour Architecture Overview

The Behaviour Engine Implementation loads an XML Policy File into a policy instance and starts and stops the policy. On activation of an action pattern, all behaviour that are part of the action pattern get connected to the event chanel or the timer respectively.

12.4 Arbiters and Messages

Arbiters select the action from all the suggestions the behaviours of an action pattern make. The behaviours make a suggestions by sending an arbitration message to the arbiter. Each arbiter has its own arbitration message format, depending on the arbitration scheme used and the actuatory device(s) it controls.

In *Miro* currently exist three arbiters. The `MotionArbiter` used in all examples, the specialized `PowerArbiter`, that uses the SparrowMotion interface and the experimental `WindowArbiter`.

12.4.1 MotionArbiter

The arbiter acts only on the motor output of the behaviours: instead of sending motor commands directly to the `Motion` interface, the behaviours invoke the method arbitrate() of the arbiter, passing it a message.

The `MotionArbiter` uses a simple fixed priority, winner takes all, arbitration scheme. That is, each behaviour has a fixed priority that corresponds with the order the behaviours are defined within the action pattern. In the `MotionArbiterMessage`, that is, the parameter passed by the `arbitrate()` call, the behaviour can specify whether it wants to suggest the arbiter an action or not. The action is defined by the translational and rotational velocity for the motion. The arbiter then decides whether to transform the content of the message into motor commands or whether to issue the motor commands of another, conflicting behaviour which has a higher priority.

Note that the arbitration of messages before conversion into motor commands, however, is only for the wheels of the robot by the `MotionArbiter`. The other actuators, for instance the camera pan-tilt unit or the grippers, are to be controlled directly in this case. No arbiter has to be defined for them, as long as a only single behaviour is expected to control them.

12.5 Implementing a Behaviour

All behaviours are derived from the base class `Miro::Behaviour`. The central method of this class is the method `action()`. It is to be overwritten by the programmer and has to contain the behavioural code. The method becomes invoked by the behaviour control file, i.e. `Behaviours.cpp`, which includes the behaviour header files and which links the behaviours to their corresponding control loops. However, if only a new policy is combined from existing behaviours, then `Behaviours.cpp` does not need to be modified.

The behaviours' `action()` method is expected to call an arbitration method or send a transition message. Note the inversion of control flow. A behaviour is not allowed to jump into some infinite loop, but the `action()` method is called continously as long as the behaviour is active within some action pattern. So every `action()` method must return quickly enough after its invocation to give time to the others. This implements the idea of simple reactive behaviours rather than complex strategies. If it is necessary to run a behaviour continuously in parallel to other behaviours, then own threads have to be created, e.g. using ACE.

Only one instance of each behaviour object is used within the behaviour engine, even for different action patterns. Therefore data that needs to be stored between subsequent `action()` method invocations can be easily stored within member variables.

From an implementation perspective, there are three kinds of behaviours, depending on how the data, the behaviour bases its decisions on, is delivered.

12.5.1 Miro::TimedBehaviour

This is the base class for a timer scheduled behaviour. A behaviour derived from this class runs with all its brothers and sisters cooperatively multi-threaded in one thread of control. The pace at which its `action()` method is called is selectable by a parameter of the base class.

This is the most simple form of behaviour design and in many cases most straight forward. It is especially suitable for behaviours that poll their sensory information of the world model, like for instance the current scan of the laser range finder for collision avoidance. Also the above mentioned wander behaviour - which doesn't use any sensory information would be best implemented as a child of the `Miro::TimedBehaviour` class.

See `$MIRO_ROOT/examples/behaviours/simple/Wander.cpp|h` for a complete source code example.

12.5.2 Miro::EventBehaviour

This is the base class behaviours using asynchronous sensory information published by the Notification Service. It subscribes for the events it likes to get pushed and the action method is called whenever a new message arrives. A pointer to the current structured event is then available as member variable.

The emergency stop behaviour described above would be a good candidate for such a behaviour, since it would have to poll excessively as a timed behaviour in order to minimize the latency between a bumper pressing a the actual stop, while it would only need to call an arbitration method in very rare occasions.

See `$MIRO_ROOT/examples/behaviours/simple/TactileStop.cpp|h` for a complete source code example.

12.5.3 Miro::TaskBehaviour

Behaviours derived from this calls run within their own thread of control, not blocking others even if they need fairly long for their decisions. Note that such a behaviour is likely to be miss designed, since it contradicts the behavioural approach to need excessive time to come to a decision. But if you have need for something like this, since you are doing something we didn't think of, this is the class to base your behaviour on. Note however, that since behaviours are shut down cooperatively, also a task behaviour is not allowed to loop indefinitely within its action method. If the behaviour is still part of the currently running action pattern, its action method will be call immediately after giving control back to the behaviour framework.

12.5.4 Behaviour Parameters

Behaviours usually get used within different action patterns. But they are often expected to behave slightly different within each constellation. Therefore each behaviour has an associated `BehaviourParameters` class which is designed to hold the different parameter sets for the different use cases of a behaviour. These parameter classes become initialized on startup of the behaviour framework and are expected to be static and constant during the run of an entire policy. (How to handle dynamic parameters like destination coordinates of a 'move to position' behaviour is explained in section 12.5.7)

Behaviours, which subclass the `BehaviourParameters` class, need to have their own `BehaviourParameters` subclass factory methods, for dynamic instantiation. This is captured by the two macros:

- BEHAVIOUR_PARAMETERS_FACTORY(X)
- BEHAVIOUR_PARAMETERS_FACTORY_IMPL(Y, X)

X denotes the name of the behaviour parameters class and Y denotes the name of the behaviour class. The first macro has to be placed within the class definition and the second within the cpp-file containing the behaviour implementation.

The behaviour parameters classes are handled by the parameters framework described in the previous chapter. Section **??** covers the details for parameter classes for behaviours.

12.5.5 Behaviour Initialization

Before an action pattern becomes activated all its behaviours `init()` methods are called successively. This allows behaviours to initialize their per task parameters (like destination coordinates) in a convenient way. Note however, that the `init()` method can be called while the behaviour is already active (see section 12.5.6) and so its `action()` method can be concurrently running. Therefore a mutex is needed to avoid race conditions. For an example see:

`$MIRO_ROOT/examples/behaviours/simple/Timer.cpp|h`

Parameters that are valid for the whole lifetime of a behaviour, such as references to the robots services or other objects within the behaviours address space are best to be passed during construction of the behaviour, forming a so called initializing constructor.

12.5.6 Behaviour Activation and Deactivation

When a behaviour is to become active due to it being part of an action pattern its `open()` method is called. If the behaviour is no longer part of the next to be running action pattern its `close()` method is called. If a transition from one action pattern to another is performed and the behaviour is part of both behaviour sets, then no calls to `close()` and `open()` methods will be issued. Only the behaviours `init()` method will be called, to allow it to update its parameter sets. This is useful to avoid unnecessary behaviour shutdown.

Note also, that the `close()` method can be called while the behaviour is concurrently within the `action()` method. — Its call to the arbiter will then just be ignored. On the call of `open()` however it is guaranteed that the behaviour currently is not running.

12.5.7 Changing Behaviour Parameters within an Action Pattern

Another way for interaction with the behaviour framework from outside is by changing the parameters of a behaviour within an action pattern during execution of the behaviour engine. In this case the behaviour name as well as the action pattern it belongs to has to be known. You can then query as well as set the behaviour parameters instance via the interface of the policy.

However there are some issues that have to be take care of. First, the type of the derived behaviour parameters struct has to be known, as you get a pointer to a BehaviourParameters instance, that has to be down casted appropriately. Second, to avoid memory leaks and segmentation faults, the memory handling has to be understood. When querying the parameters, the caller takes ownership of the object returned. When setting the parameters, the action pattern takes ownership of the passed parameters instance. - Therefore the object has to be allocated on the heap (that is, with `new`). But actually that's not too much of a problem, if you stick to the intended protocol: If you first retrieve the parameters instance by querying the policy, change the parameters and write them back by the set method, everything works okay.

A similar interface exists for the behaviours themselves. But they only can query and set the parameters of behaviours in action patterns that are linked to the current action pattern by a local transition. The behaviours can therefore only query the parameters by the transition name instead of the action pattern name.

12.6 Arbiters

The arbiter framework is similar to the behaviour classes. It will be explained in more detail as soon as we do some more work on arbiters.

12.7 Building Action Patterns

"Now I built all my behaviours. What code do I have to write to make them an action pattern?" Well, you don't have to write code. Action patterns and policies are defined within an XML file. Allowing for fast and convenient modifications. Which is especially cool when debugging. An Instance of each arbiter and behaviour has to be registered within repositories. Afterwards, the policy can configure the action patterns and transitions by parsing the XML file. To make live easier, the editing of action patterns, transitions and behaviour parameters can also be done using the GUI based PolicyEditor (see section 12.8), but let us first take a look at how those things are put together in the XML syntax.

12.7.1 The Policy File

A policy file can have the following tags and attributes:

<!–MiroPolicyDocument–>

policy One file always describes one entire policy. A policy defines a state machine that consists of a set of action patterns. Each action pattern describing one state of the machine.

actionpattern An action pattern describes a set of behaviours, that can run simultaneously and produce some emergent higher level behaviour.
Attributes:

- *name* The name of the action pattern.
- *start (true/false)* The action pattern to be active at startup is to be marked true, the others false or unmarked.
- *x, y* For GUI purpose, just ignore them.

behaviour That's what all the chapter is about. Read it again.
Attributes:

- *name* Name of the behaviour.

parameter Behaviours parameters can be specified within the policy file for each action pattern.
Attributes:

- *name* Name of the parameter.
- *value* The value of the parameter.

transition Message that triggers a transition to another action pattern.
Attributes:

- *target* The action pattern to activate next.

arbiter To decide which action to choose from the different outputs of the different behaviours one needs to arbitrate one way or the other.
Attributes:

- *name* The name of the arbiter to use.

12.7.2 The Repositories

A policy can be built on the basis of an XML description. For the parser of this description to be able to construct the policy, it has to be able to refer to instances of behaviours and arbiters that are mentioned within the XML file by their name. For this purpose a `Miro::BehaviourRepository` and a `Miro::ArbiterRepository` class do exist. At these Repositories an instance of each behaviour and arbiter has to be registered. The name of the behaviour has to be reported by the `behaviourName()` method. Since we only need one instance of each of these repositories, there do exist a global instance of each. A pointer to such an instance can be obtained by the classes static method `instance()`.

12.7.3 The Behaviour Factory

All the initialization stuff necessary before constructing an instance of the Policy class can be done within the main function of your program. This task consists mostly of obtaining the needed object references, instancing behaviours and arbiters and registering them at their respective repositories. This can be bundled within a so called behaviour factory class as can be seen in the behaviour example at:

`$MIRO_ROOT/examples/behaviours/simple/BehaviourFactory.cpp|h`

It uses the simple base class:

`$MIRO_ROOT/src/miro/BAFactory.cpp|h`

12.7.4 The Behaviour Engine

There does exist a small CORBA based interface for interaction with the behaviour framework. It is the `BehaviourEngine` interface and the `PolicyController` (section 12.9) is a simple GUI based client for this interface. To allow your own behaviour sets to be controlled via this interface, the policy controller implementation has to be instantiated within your executable. This can be seen in the example at:

`$MIRO_ROOT/examples/behaviours/engine/SimpleBehaviourEngine.cpp`

12.8 The Policy Editor

Editing large XML files is tedious and error prone. Therefore *Miro* offers an GUI based editor, with which you can build and edit your policies. You can also edit the parameters defined within the Parameters classes of your behaviours.

The policy editor can be started with no parameters or with the policy file to edit as first parameter. It has a menu bar and displays the policy graph within the big scroll view area of the program.

12.8.1 The Menu

File

- New
  Create a new policy.
- Open ...
  Open a policy file. The file dialog will be displayed.
- Save
  Save the current policy to disc. If no file name is yet specified, Save as will be invoked instead.
- Save as ...
  Save the current policy under a new file name. The file dialog will be displayed.
- Send to ...
  Send the current policy to a robot. A dialog box will be displayed to enter the robots name (the naming context, its behaviour engine is registered at). The behaviours and arbiters within the policy have to be present within the behaviour and arbiter repositories in the behaviour engine running on the robot. - Otherwise the parsing will fail.
- Quit
  Quit the policy editor.

Options

- Behaviour Descriptions
  Display the behaviour descriptions dialog. The dialog consist of a listbox with the currently loaded behaviour parameter description files. (See the chapter 7 for more details on parameter files.) Buttons for adding and deleting files are

provided. On pressing the button add, the file dialog will be displayed. Note that the selected file has to contain a valid behaviour description. - Verifying, whether the load was successful is currently only done on leaving the dialog. On the positive side the currently loaded description is memorized in the file .PolicyEditorConfig.xml file in the users home directory for further runs of the policy editor.

Help

Currently this menu only contains two about dialogs.

12.8.2 Editing the Policy Graph

Clicking with the right mouse button into the editing area produces a popup menu with the option to place a new action pattern. Enter the name of the new pattern in the dialog box.

Clicking with the right mouse button into the name of an action pattern produces a popup menu with the following options.

- Start pattern
  Select the pattern to be the start pattern. That is the pattern, the policy will begin with by default. Every policy has to have exactly one start pattern. The start pattern is marked with two asterix around the pattern name.
- Add behaviour
  Add a behaviour to the action pattern. A list of available behaviour will pop up. Every behaviour can be instantiated only once per action pattern.
- Add transition
  Click on another action pattern to link the two pattern with a transition. A dialog box will appear to enter the transition name.
- Rename pattern
  A dialog box will appear to enter the new name of the action pattern.
- Rename transition
  A dialog box will appear with the name and the target of the transitions. You can also delete a transition by deleting the name or the target of the transition. - Be careful, the new entry will only be accepted, if you left the entry field by clicking on another entry before hitting the okay button.
- Delete pattern
  Deletes the pattern.

Clicking with the right mouse button into the green area of an action pattern produces a popup menu with the following options.

- Set arbiter
  Select the arbiter for to the action pattern. A list of available arbiters will pop up. Every action pattern has to have an arbiter.
- Delete arbiter
  Deletes the arbiter from the action pattern.

Clicking with the right mouse button onto a behaviour within an action pattern produces a popup menu with the following options.

- Up
  Move the behaviour up one position in the list of behaviours. The position in the behaviour list represents the priority of the behaviour at the arbiter. - The higher, the more important.
- Down
  Move the behaviour down one position in the list of behaviours.
- Set parameters
  The parameters dialog will be displayed. It has three columns. In the left one, the parameter names are displayed as label. The middle column contains a single line entry field for the parameter. In the right column the measure of the parameter is displayed as label, if no measure is available the parameters type is displayed. The default value is displayed as bubble help associated with each entry field, if available. The parameters type is displayed as bubble help associated with each measure label. The dialog is strictly typed. Therefore the okay button is inactive if any of the entry fields are containing invalid input, for instance a value bigger than 180 for an Angle type.
- Delete behaviour
  Deletes the behaviour from the action pattern.

12.8.3 Describing the Available Behaviours

To be able to use your own behaviours within the PolicyEditor, you have to describe their properties (name and parameters) within a parameters description file. The parameter framework is explained in more detail in the previous chapter. Anyhow there are some minor restrictions for behaviour parameter description files, that are explained below:

<!–MiroParametersConfigDocument–>

config config_group Attributes:

- *name* The name of the config group. Within the behaviour parameters framework the valid names are **behaviour** and **arbiter**. Within each group only config items of the specified type are allowed to occur. Other groups are processed by the parameter auto code generation tool `MakeParams` but ignored by the PolicyEditor.

config_item Attributes:

- *name* The name of the behaviour or the arbiter.
- *parent* The name of the super class. That has to correspond to the corresponding super class of the behaviour.
- *namespace*

config_parameter Attributes:

- *name*
- *type* No nested types are allowed!
- *default*
- *measure*
- *inherited*

12.8.4 Example Behaviour Description File

12.8.5 Auto-generating Parameter Class Code

Having written the description of your behaviours parameters within XML, it is possible to auto-generate the necessary code for your behaviours associated Parameters class. See the previous chapter for details.

12.9 Policy Controller

The policy controller is some kind of remote control for the behaviour engine. It consists of a panel with four buttons and a menu bar. The buttons allow to start, stop, suspend and resume the behaviour engine.

12.9.1 The Menu

File

- Connect robot
  Connect the policy controller with a behaviour engine of a robot. A dialog box will be displayed to enter the robots name (the naming context, its behaviour engine is registered at).
- Load policy
  Load a policy file into the behaviour engine. A dialog box will be displayed to enter the file name. The file has to be locally accessible by the robots behaviour engine.
- Send policy
  Send a policy to the robot. The file dialog will be displayed to select the policy. The behaviours and arbiters within the policy have to be present within the behaviour and arbiter repositories in the behaviour engine running on the robot. - Otherwise the parsing will fail.
- Quit
  Quit the policy editor.

Edit

- Send transition
  Send a transition to the behaviour engine. A dialog box will be displayed to enter the transition name. This is for debug purposes only, as the transition will only succeed if it is registered at the currently active action pattern - which in turn can change any time. There will be also no feedback whether the transition succeeded or not.
- Send global transition
  Send a global transition to the behaviour engine. A dialog box will be displayed to enter the action pattern to activate. The action pattern has to be present within the current policy, otherwise the transition will fail.

Chapter 13

Writing a *Miro* Service

The programming of a *Miro* service is divided into two step: Accessing the low level hardware device (such as the Can bus) and writing the high level interfaces. Note that many hardware devices incorporate multiple sensor modalities.

13.1 High-Level Server Programming

In order to write a server you have to do the following steps:

1. Describe your interface in IDL.
2. Translate the IDL description to C++ files.
3. Implement your methods.

13.1.1 Copy the *Miro* server template directory

- Which files are in there

13.1.2 Describing an interface in IDL

- edit interface description (.idl)
- idl/Makefile

13.1.3 Translating the IDL description to C++

- call TAO_idl (in `$(TAO_ROOT)/TAO_IDL`)

13.1.4 Implementing your own methods

- in ...Impl
- don't forget to document !!

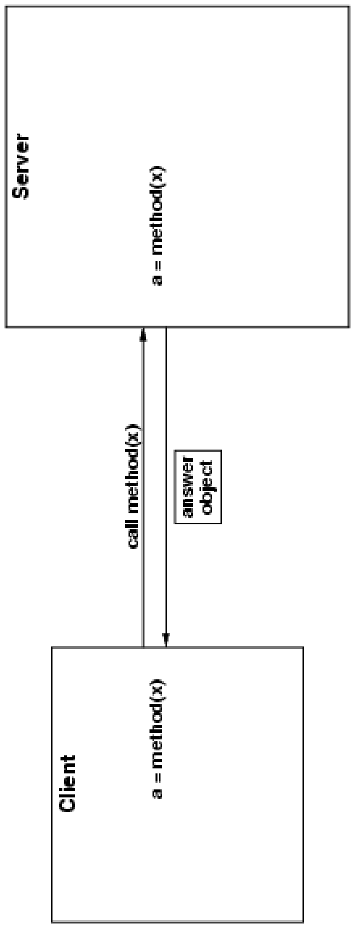


Figure 13.1: Client-Server Architecture Overview

13.2 Low-Level Server Programming

Low-Level Servers are Servers that talk to hardware devices. They usually use a device driver that is compiled into the kernel or loaded as a module. They provide handling for special protocols which have to be followed and provide the services of the hardware device in an abstracted way, so that they are useful.

Hardware devices in a UNIX system are represented by file descriptors. There are two groups, character devices and block devices, character devices are stream based and provide for sequential reading and writing, while block based devices provide random access to block addresses. At the moment all hardware devices used by miro use the character device abstraction.

A useful *Miro* low-level server must provide two functionalities: interaction on the CORBA level with (potential multiple) clients that use the service concurrently and handling of all communication with the device, which may be asynchronous to the requests. This requires a multithreaded design. Fortunately the ACE framework provides design patterns that fulfill these needs. For a schematic view of a *Miro* low-level server see figure 13.2.

13.2.1 The Device Framework

Connection

EventHandler

Consumer

13.2.2 The Configuration Framework

13.2.3 Parameter

13.2.4 XML parsing

13.2.5 The Reactor and Events

The ACE reactor class provides an abstraction of the popular `select` method in common UNIX programming environments. It is able to invoke an event handler when data is readable from the file descriptor[1]. It provides low latency and the thread does not consume any CPU power while waiting for data to come. To keep the responsiveness of this architectural element high[2], only few computation should be done inside of the event handler, usually this event handler only assembles full packets and passes these on to a separate task which accounts for higher level processing. It is important to note that the reactor and the event handler share a single thread of execution, so don't expect to get feedback from the reactor within the event handler.

13.2.6 Using Tasks

Tasks are the second important design pattern provided by ACE that we use. A task is an assembly of one or more threads, that share a common message queue, which can be used to pass requests to the threads. We usually use a task for handling of complete packets assembled by the event handler, which puts them on the message queue. According to the type of the packet the tasks takes appropriate action. This usually consists in retrieving data from the packet and signaling it on

[1] it can also handle timer events and asynchronous writes, see [8] for details
[2] which is essential to avoid flooded input queues

a condition variable. Condition variables are the main mechanism of the (CORBA) server implementation methods to wait for data requests or events that are provided by the hardware. The ACE task class provides an easy scaling for multiple working threads that concurrently work on the different requests.

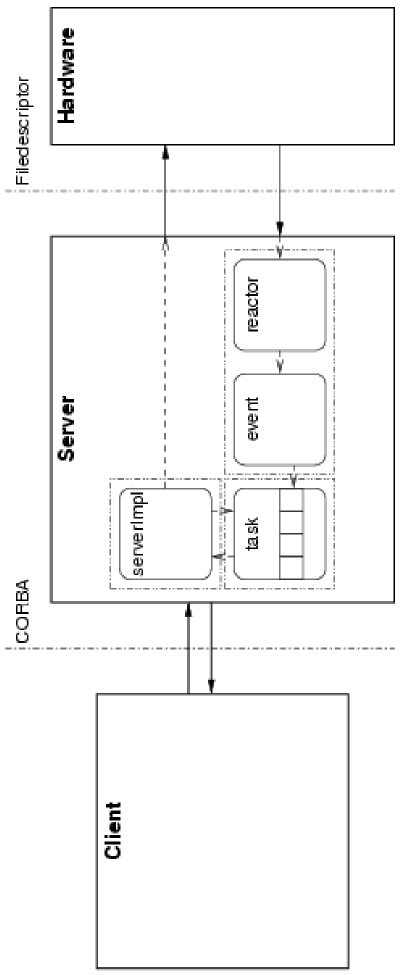


Figure 13.2: Client-Server Architecture

13.2.7 Thread/Task Synchronization

13.2.8 ACE logging

13.2.9 A Simple Example

Appendix A

Miro installation

This section is an extended version of the INSTALL file contained within *Miro*. If you have problems installing *Miro*, read on. Ask your system administrator if there exists a central installation, or whether it would be more convenient to prepare a central installation.

A.0.10 Requirements

You need the following packages in order to compile *Miro*:

- ACE (>= 5.2.2): Please refer to the installation instruction in B for building and installing ACE. The source tarball and CVS access is available at:
  `http://www.cs.wustl.edu/~schmidt/ACE.html`.
- TAO (>= 1.2.2): Building and installation is also described in section B. TAO downloads are available on:
  `http://www.theaceorb.com/`.
- Qt (>= 2.2.4): Installation should be quite easy, as Qt uses a configure script for automatic system checks. Qt is installed on most modern Unix/Linux systems anyway, since KDE relies on it. Recent versions of Qt can be found at:
  `http://www.trolltech.com/`.

If you want to compile the wrapper classes for the speech detection system, you also need:

- Speech tools: The Edinburgh Speech Tools Library is a collection of C++ class, functions and related programs for manipulating the sorts of objects used in speech processing. It is to provide the underlying classes in the Festival Speech System. We use version 1.2.3 that can be retrieved from:
  `http://www.cstr.ed.ac.uk/projects/speech_tools`
- Festival: The Festival Speech Synthesis System is a general multi-lingual speech synthesis system developed at CSTR. We successfully used version 1.4.2. See:
  `http://www.cstr.ed.ac.uk/projects/festival/`

- Sphinx: The CMU Sphinx Group Open Source Speech Recognition, a real-time, large vocabulary, speaker independent speech recognition system. We use version Sphinx2-0.4, downloaded from:
  `http://www.speech.cs.cmu.edu/sphinx`

Depending on your camera system, you may need libraries for the IEEE 1394 (aka Firewire) support:

- libraw1394: This library provides direct access to the IEEE 1394 bus through the Linux 1394 subsystem's raw1394 user space interface. It can be found one it's homepage:
  `http://www.linux1394.org/`
  or on SourceForge:
  `http://sourceforge.net/projects/libraw1394/`
- libdc1394: It is a library that is intended to provide a high level programming interface for application developers who wish to control IEEE 1394 based cameras that conform to the 1394-based Digital Camera Specification. Available via:
  `http://sourceforge.net/projects/libdc1394/`

Both libraries are available as compiled packages for the most modern Linux systems too.

To build the documentation, your system should provide the following tools:

- Doxygen: It is a JavaDoc like documentation system for C++, C, Java and IDL. It can generate an on-line HTML documentation extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code. You can download from the homepage: `http://www.stack.nl/d̃imitri/doxygen/`
  or again from SourceForge:
  `http://sourceforge.net/projects/doxygen/`
- The LaTeX package (either pdflatex or latex and dvips, bibtex and makeindex – available via one of the CTAN server, e.g. `http://dante.ctan.org`) and the convert image convertsion utility from the ImageMagick package (`http://imagemagick.sourceforge.net/`).

A.0.11 Download

As if you read this text, you already have a version of *Miro*, but if you want to update this version, or need some additional information, please have a look at `http://smart.informatik.uni-ulm.de/MIRO/index.html`

A.0.12 Compilation

Miro is now shipped with a configure script, so the package can be installed with the usual steps:

```
./ configure
make
(make install )
```

The configure script looks automatically for the necessary and optional software packages, and will stop or print a warning if some of them are not found. If you install e.g. ACE in an unusual place, you can pass the configure script some additional options like for example `--with-ACE=<ACE-root-dir>`. For Qt, their are more flexible command-line parameters, because Qt is often distributed over several directories (like e.g. in a standard Debian installation). In contrast to older version, the configure script now searches automatically for libqt or libqt-mt (the thread-save version). A complete list of command-line options are accessible via `./configure --help`.

Another way of telling the configure script where the packages are installed is the use of environment variables. For the required packages, these are `ACE_ROOT`, `TAO_ROOT` and `QTDIR` point to the base directory of each package. Because ACE and TAO need this environment variables anyway, this is the most comfortable way to point to the directories. For the speech detection wrappers classes, `SPEECH_TOOLS_ROOT`, `SPHINX_ROOT` and `FESTIVAL_ROOT` are possible. It can be necessary, to let the `LD_LIBRARY_PATH` point to all the libraries anyway.

If the configure script did not find the different packages anyhow, or if your system require unusual options for compilation or linking that the configure script does not know about, you can give configure initial values for variables by setting them in the environment. Using a Bourne-compatible shell, you can do that on the command line like this:

```
CFLAGS=-O2 LIBS=-lposix ./configure
```

Or on systems that have the 'env' program, you can do it like this:

```
env CPPFLAGS=-I/usr/local/include LDFLAGS=-s ./configure
```

On runtime, some applications rely on two environment variables, namely `MIRO_ROOT` (the base directory) and `MIRO_LOG` (directory, where logging data is stored), so please set these variables to the appropriate values too.

Additionally, you can enable or disable several features, like the support for different robot platforms and for different video devices.

Supported robot platforms:

- The B21 port is the oldest one and should therefore be the most stable, regarding the interfaces. All of the robots hardware components are supported by *Miro*.
- The Sparrow99 and Sparrow2003 robots are our soccer robots. They are our testbed for multi-robot programming. Therefore most new technology is first tested on this platform. So even as you do not have a Sparrow99 robot (we built them ourselfs), it might be interesting to look at the sources for this robot if you are looking for group communication technology etc.
- The Pioneer1 platform is a complete port for the ActiveMedia robot series, based on the PSOS, P2OS and AROS protocols. But we could only test and extend it to the platforms available to us.
  - The old Pioneer1 is supported, but lacks maintenance, as our model is not used at the lab anymore.

  – The Performance PeopleBot is in active development and already mostly complete. Motors, Sonars, Bumpers and Video are working. PanTilt and Gripper are experimental, Zoom is to come next.

Supported video devices:

- Bttv Frame Grabbers: These frame grabber cards are supported via video for Linux. This is the standard way of connecting standard analog video cameras to the computer.
- Firewire Digital Cameras: *Miro* supports the fire wire digital camera protocol using libraw1394 and libdc1394.
- Matrox Meteor Frame Grabbers: These rather old frame grabber cards are also supported by *Miro*. This device however, is mostly unmaintained.

Finally, you can choose if the documentation should be build or not. Even if you decide not to build it, but the configure script found all the necessary tools, the Makefiles are prepared. So you can go to the `doc/tex` and `doc/html` directory later on and build the documentation there with a simple make.

After running the configure script, *Miro* show up a summary of what will be compiled and which features will not. If this is not what you desired, please check the messages coming up during the configure run for packages, *Miro* did not find.

A.0.13 Installation

Miro can be installed with the a simple:

```
make install
```

By default, the package's files will be installed in `/usr/local/bin`, `/usr/local/lib` etc. You can specify an installation prefix other than `/usr/local` by giving configure the option `--prefix=PATH` (note: if you use the prefix-option, you should install *Miro* actually, otherwise libtool may look for the library in the installation directory and therefore do not find it).

Beside that, *Miro* can be used already without installation. Therefore, a make run install the libraries and the binaries during the compilation into the `lib/` and `bin/` directory. This enlarges the *Miro* directory, but can be quite useful, if you work on *Miro* itself and a derived application at the same time.

Moreover with this trick it is possible, to use different *Miro* versions in parallel at the same time. Therefore the build process allow to compile the package in a different directory. Assuming you have extracted *Miro* in a directory called `Miro`, you can build it in two different directories with miscellaneous options:

```
mkdir Miro.B21
cd Miro.B21
../Miro/configure --enable-B21
make
cd ..
mkdir Miro.Sparrow99
cd Miro.Sparrow99
../Miro/configure --enable-Sparrow99
make
```

A.0.14 Additional make targets

Beside the already described `install` target and an solely `make` (which in fact means `make all`), there are a couple of other targets, that may be of interest:

clean: Removes object files, libraries, binaries and automatically generated code, like the parameter classes generated from the xml files and the client and server stubs generated from the idl files.

distclean: Removes all of the above files and the automatically generated `Makfiles` generated from the autotools. So you have to reconfigure the packet after this.

dist: Generates a gzipped tar-file containing a snapshot of the actual *Miro* directory, including all the necessary source files to configure and build *Miro*.

distcheck: Some as above plus a complete make of the packed tar.gz file and finally make another tarfile to ensure the distribution is self-contained. You can change the configure options that are used for this make run within the toplevel `Makefile.am` by changing the `DISTCHECK_CONFIGURE_FLAGS` variable.

A.0.15 Developer information

The build-process within *Miro* is done using automake, autoconf and libtool. Normally, there is no need to use these programs, as of running the configure script is the only thing a user have to do.

The configure shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a Makefile in each directory of the package. It also create one config.h file containing system-dependent definitions that can be used in the source file to allow appropriate compilation:

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
 ...
#ifdef MIRO_HAS_1394
 ...
#endif
```

If you want to change something within *Miro* like for example adding additional files or directories, you have to change the proper `Makefile.am`. Do **not** change the `Makefile.in` or even the final `Makefile`, because changes there are getting lost at the next automake run. The same applies, if you want to change the behaviour of the configure script itself, for example to detect new software packages or special version information, you have to change the `configure.ac` and **not** the `configure` file itself.

Assuming that you have written appropriate `Makefile.am` and `configure.ac` files, you should be able to build your project by entering the following commands:

```
aclocal -I config
autoheader
automake
autoconf
./configure
```

For every consecutive step, you can simply call the resulting `config.status` script, that rerun the configure procedure with the used configuration values.

Depending on your versions of automake, autoconf and libtool you may get some warning messages about improper usage of `LDFLAGS`. Ignore this, since it is without any influence on the final result. We tested the whole procedure successfully with at least the following versions:

- automake 1.5
- autoconf 2.52
- libtool 1.4.2

Normally, it should be enough, to use the available examples within *Miro* (see section 6.1) to add some small additions to the build process, but sometimes a deeper understanding of the used tools might be necessary. Please have a look at the manuals, available e.g. directly from the GNU homepage: `http://www.gnu.org/manual/` or read the freely available book on `http://sources.redhat.com/autobook/`.

Appendix B

ACE and TAO Installation

ACE/TAO is a large software package with many configuration options affecting the build process as well as its runtime features. Therefore we summarize here our experience with the ACE/TAO installation.

At the time of this writing we suggest to use the latest BFO (bug fix only) version ACE/TAO — currently this is 5.3.2/1.3.2. There are knon problems with the notification service in TAO.

The options that should be set additionally in the `platform_macros.GNU` file (to be found under <ACE-directory>/include/makeinclude/) are the following:

```
debug=0
```

Debug information has a huge impact on the footprint of the libraries, so as long as you do not really need it, disable it.

Additionally, for further footprint reduction you might also add the following:

```
qt_reactor =0
xt_reactor =0
TAO_ORBSVCS= Naming Notify
DEFFLAGS= -DACE_USE_RCSID=0
ACE_COMPONENTS= FOR_TAO
```

The two reactors are not needed by ACE/TAO/Miro. So as long as you do not use them yourself, they might just be left out. Also Miro only uses the CORBA Naming Service and the CORBA Notification Service. Therefore the other services of TAO need not to be built.

We also recommend to read the installation instructions provided by ACE carefully. We admit, the installation isn't entirely trivial.

To compile and run programs that use the ACE/TAO toolkit, also the environment variables `ACE_ROOT` and `TAO_ROOT` need to be set to the appropriate root directories of your ACE/TAO installation. For the bash shell the following lines in your local `.bashrc` file should do the job:

```
export ACE_ROOT=<path to the ACE directory>
export TAO_ROOT=$ACE_ROOT/TAO
export LD_LIBRARY_PATH=$ACE_ROOT/ace/:$LD_LIBRARY_PATH
```

Appendix C

Project setup with Automake and Autoconf

This chapter tries to give a short introduction on how to set up your own project that uses *Miro* by the help of automake and autoconf. This text was created using autoconf 2.52 and automake 1.5, so your local installation may differ slithly.

First describe the the mode of operation of automake and autoconf is explained by a real-world example from the *Miro* source base. Afterwards the creation of the root configuration file of the autoconf tool, `configure.ac` is discussed. Afterwards the different Makefile.ac templates, shipped with *Miro* are introduced. After a short survey on the actual buid directives, the chapter concludes with links to internet resources on the auto-tools.

C.1 Introductory Example

First a short introduction into the whole structure of automake and autoconf, that uses a two-stage process.

- With automake you can (normaly) easily describe, how the files within a folder should be processed. The commands for this are written in a file called Makefile.am. As a minimal example look for example in `$MIRO_ROOT/src/-can/Makefile.am`:

  ```
  lib_LIBRARIES = libcan.a
  libcan_a_SOURCES = Parameters.xml CanConnection.cpp CanMessage.cpp ...
  ```

  This generates a (static) library named `libcan.a`, which is build from the assigned sources `Parameters.xml, CanConnection`... All other stuff within the Makefile.am is e.g. which files are installed into which directory, if the user applies a *"'make install"'* (`can_include_HEADERS`, note the prefix of the macro), or which files are generated automatically during the build-process and can therefor removed if a *"'make clean"'* is desired (`BUILT_SOURCES` and `CLEANFILES`). These Makefile.am are converted into `Makefile.ac` using the *automake* program.

- *autoconf* generates the real `Makefile` for the `Makefile.ac` files. Therefor it has to know the paths and options necessary for the build process and which files, libraries and programs it has to compile at all. Exactly that is, what the configure-script tries to detect and guess automatically. Of course no one wants to write a configure-script directly (take a look at it, it looks really, really ugly. Moreover a program should compile on a couple of different systems, each having their own quirks and peculiarities). Therefor one writes a `configure.ac` files, that is finally converted into the configure-script. The whole autoconf stuff is based on so called macros, from which are a lot of them allready included:

  ```
  AC_CHECK_LIB(raw1394, raw1394_get_libversion, ac_have_libraw1394=yes, [ac_have_libraw1
  AC_CHECK_HEADERS(errno.h fcntl.h stdlib.h)
  AC_CHECK_PROG(ac_has_dvips, dvips, yes, no)
  ```

  There are e.g. macros to test the existence of libraries, header files or executable programs. If you need a more complicated test, you can write them with a language called M4 (M4's syntax is quite close to a normal shell syntax). The results of all tests are write in a file called config.h additionally. This file can be included into the C-files and are treated as normal c-defines (e.g. `#define MIRO_HAS_METEOR 1`).

C.2 Create `configure.ac`

The easiest way to start with is using the autoscan program (from the autoconf-package). This will create a file called configure.scan that you should rename to configure.ac. (*autoscan* is also a good tool to check your existing configure.ac for missing tests). Now look into your new configure.ac file. The first line (`AC_INIT`) describe your project, so fill it with useful values, a name for your package, your version and an email address for bug-reports. The next two lines (`AC_CONFIG_SCRDIR`, `AC_CONFIG_HEADER`) should be fine.

The remainder of the file is group into different parts. First part is the check for programs. *autoconf* offers several ready to use checks for this, like e.g. `AC_PROG_CC` that checks several behaviours of your installed C-compiler or `AC_PROG_LN_S` that looks how to create a symlink. The automatically included macros should be fine.

The next paragraph checks for libraries. Note, that the philosophy of *autoconf* is to check for the possibility of creating and compiling a program with a certain function using this library, not the pure existence of this library. So look for a function that should be within the different libraries and insert them in the second field of the `AC_CHECK_LIB` macros (the first name is the library name itself). Possibly you have to remove lines for libraries, that are in fact builded by your project itself (Note, that it may be difficult to use C++ specific stuff here).

Third, header files are checked. Beside some prefabed macros for standard headers (e.g. `AC_HEADER_STC`) you can insert additional files into the `AC_CHECK_HEADERS` macros. Insert them as a space separated list.

Next two paragraph contains checks for typedef and structures and library functions. The automatically included macros should be fine, otherwise include additional functions into the `AC_CHECK_FUNCS` macro. The functions here refer to standart C functionality, not to mix up with special functions provided by additional libraries as described above.

The last, long macro `AC_CONFIG_FILES` contains a space separated list of all the Makefiles the configure script has to produce. This means, if you add a directory into your project later on, you have to insert the path here.

C.2.1 Tests for more complex packages (and facilities)

Most checks for the functionallity of your system should be covered with the above, autogenerated checks. But to build a project that relies on other, more complex packages (in our case *Miro*, ACE, TAO, Qt, etc), it is necessary to add additionally tests to the configure script.

Additionally checks are written in a language called M4. It is up to you, where to save these files, but I prefer to add an extra directory named config, so the following explanation will assume this.

First you have to check for the existence of e.g. ACE, TAO and Qt. Simply copy the necessary files `search_ace.m4`,`search_tao.m4` and so on from `$MIRO_ROOT/-config/` to your local config directory and add `AC_SEARCH_ACE`, `AC_SEARCH_TAO` and so on (the name of the function macros) to the `configure.ac` script (for example directly above the `AC_CONFIG_FILES` macro.

To be sure to add all the necessary libraries and options that are used to compile ACE/TAO, additional test are prepared within the file `feature_ace.m4` (again copy this file into your local folder and add the macro name into `configure.ac`).

To check for Miro and parts of the functionality, we prepared two files `$MIRO_ROOT/-templates/search_miro.m4` and `$MIRO_ROOT/templates/features_miro.m4`. Again use them the usual way, e.g. insert `AC_FEATURES_BTTV_MIRO` to be sure that Miro is compiled with support for $BTTV$ frame grabber cards.

If you need to write your own tests and checks, please feel free to use and modify these files. Additional macros that can be used out of the box or as examples for own tests, look at:

```
http://wwww.gnu.org/software/ac-archive/
http://ac-archive.sourceforge.net/
```

C.3 Create `Makefile.am` for directories, libraries and executables

The next step is to prepare a Makefile.am within each directory that contain files that need to be compiled. The easisest way is again to use the examples within `$MIRO_ROOT/templates/`. For example to create a Makefile.am that should call other Makefiles in other subdirs, take `$MIRO_ROOT/templates/Makefile.am.dir`, rename it to `Makefile.am` and add the names of the subdirectories. There are also examples for sources that should be compiled into a static library (Makefile.am.lib.a) into a shared library (Makefile.am.lib.so) or different executables (Makefile.am.bin). All examples contain some comments to guide you, what to fill in.

Remeber to be sure, that each directory with a `Makefile.am` is listed in the `AC_CONFIG_FILES` macro in the `configure.ac` script.

C.3.1 Makefile.am.dir

This Makefile starts subbuilds in the specified directories. It is an example for conditional compilation (depending on the configuration). If you don't have conditional branches, you can left the second variable empty or reduce the Makefile.am to a single line, like e.g. `SUBDIRS = firstDir secondDir`.

C.3.2 Makefile.am.bin

This Makefile builds a single or several binaries. The name of the binaries to be built are specified in the variable `bin_PROGRAMS`. There have to be an extra line for each binary to be build starting with the name of the binary itself and an additional `_SOURCES`-suffix. The sources are specified with their suffix (.cpp). Please also list all the header files here, otherwise they don't get distributed by an call to `make dist`. If the programs need to be linked against several libraries, there is again one line per binary containing the name of the binary plus an additional `_LDADD`-suffix.

C.3.3 Makefile.am.qt

This is a special Makefile.am template for Qt programs. For simplicity, the Makefile is designed for a single binary per directory, but beside that, most of the above section also applies here. The only exception is, that we have to distinguish between normal source files (variable `sources`) and files (variable `tomocsources`) that need to be compiled with Qt's Meta Object Compile (`moc`). Be sure to **not** include the header files into the `tomocsources` variable, but to the `sources` variable, otherwise they get "cleaned".

C.3.4 Makefile.am.lib.a

This Makefile builds a static library. The name of the library can be specified in the variable `lib_LIBRARIES`, the files building the library in the variable `sources` (with the .cpp suffix; or .xml in case of an xml-file for *Miro*'s parameter framework). Headers that are not automatically considered because they don't have a corresponding cpp-file should be specified in the variable `extraheader`. Don't forget to name the prefix for the `_SOURCES`-macro with the name of the resulting library. Therefore please substitued all occuring dots with underscores (e.g. the usual `.a` a the end of an static library). Additionally, replace the prefix for the macro `_include_HEADERS` with the name of the directory, to which the header files should be installed (see also the description of `Make-rules` below).

C.3.5 Makefile.am.lib.so

Makefiles for shared libraries look quite the same as for static libraries, to most of the above applies here too. The only difference from a users perspective is the name of the variable `lib_LTLIBRARIES` where you name your library (note the `LT` within the variable (and the targets at the end of the `Makefile.am`) which indicates that the library is build using the libtool package).

C.3.6 Conditionally Compiled Sub-projects

If you programs or libraries need to be compiled differently depending on configuration options, please have a look at an existing `Makefile.am`, e.g. in `$MIRO_ROOT/-examples/behaviours/engine/` or `$MIRO_ROOT/src/miro/`.

C.3.7 Create other `Makefile.am`

Because auto-conf/make themself are quite unflexible if you intend to do something, that the autotools are not designed to, you can also insert an "'all-local"' target into the `Makefile.am` that is treated as a normal Makefile part. Although this has to be necessarry sometimes, be careful with this, auto-conf/make may fail to calculate the correct dependencies which in turn may lead to incomplete and corrupt builds. Save places to write your own stuff are for example in the documentation directory. See `$MIRO_ROOT/doc/html` and `$MIRO_ROOT/doc/tex` for examples on building a doxygen or latex documentation.

C.3.8 Make-rules

Last you may have a look at the `Make-rules` file within the `$MIRO_ROOT/config/` directory. This file is included into each `Makefile.am`, so rules or variables that are needed globally, can be inserted here. If you want to define additional installation subdirectories for the header files, you can see examples here.

C.4 Build the beast

To finally build the new project, you have to call the different programs, which in turn convert the prepared files into a real, make-based project.

Assuming that you have written appropriate `Makefile.am` and `configure.ac` files, you should be able to build your project by entering the following commands:

autoheader Creates `config.h.in`.

aclocal -I config Adds `aclocal.m4` to directory. Defines some m4 macros used by the auto tools and the self-written tests from the config-directory.

autoconf Creates `configure` from `configure.ac`

automake Creates `Makefile.ac` from `Makefile.am`

./configure Creates `Makefile` from `Makefile.in`

make

Just repeat the last 5 steps to completely rebuild the project. Most projects have an `autogen.sh` script that runs everything up to the configure step.

C.5 Additional sources of help

In case of problems, first have a look at the offical manuals of both, automake and autoconf. You can find them on the GNU webpage:

`http://www.gnu.org/software/automake/manual/automake.html`
`http://www.gnu.org/software/autoconf/manual/autoconf-2.57/autoconf.html`

There is also a book available under the terms of the Open Publication License:

`http://sources.redhat.com/autobook/download.html`

Bibliography

[1] R. A. Brooks. Intelligence without representation. *Artificial Intelligence Journal*, 47:139–159, 1991.

[2] Pradeep Gore, Ron Cytron, Douglas C. Schmidt, and Carlos O'Ryan. Designing and optimizing a scalable CORBA notification service. In *LCTES/OM*, pages 196–204, 2001.

[3] Dave L. Mills. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. Network Working Group Request for Comments: 2030, October 1996.

[4] OMG. *Notification Service Specification*. Object Management Group, Inc., Needham, MA, 2000.

[5] OMG. *CORBA/IIOP Specification*. Object Management Group, Inc., Needham, MA, 2001.

[6] Douglas C. Schmidt. *Patterns and Performance of Real-time Object Request Brokers*. Distributed Object Computing Group, University of California, Irvine, 2000.

[7] Douglas C. Schmidt. *The ADAPTIVE Communication Environment. An Object-Oriented Network Programming Toolkit for Developing Communication Software*. Distributed Object Computing Group, University of California, Irvine, 2001.

[8] Umar Syyid. *An Introduction to the ADAPTIVE Communication Environment*. Distributed Object Computing Group, University of California, Irvine, 2000.

[9] Hans Utz. Quo vadis? Robuste hierarchische Navigation für autonome mobile Roboter. Diplomarbeit, University of Ulm, Ulm, Germany, October 2000. in german.

[10] Hans Utz, Gerd Mayer, and Gerhard Kraetzschmar. Middleware logging facilities for experimentation and evaluation in robotics. Workshop 27th German Conference on Artificial Intelligence (KI2004), University of Ulm, September 2004.

[11] Hans Utz, Freek Stulp, and Arndt Mühlenfeld. Sharing belief in teams of heterogeneous robots. In *RoboCup-VIII (to appear)*, Lecture Notes in Artificial Intelligence, Berlin, Heidelberg, Germany, 2004. Springer-Verlag.