Using and Extending the Miro Middleware for Autonomous Mobile Robots

Daniel Krüger, Ingo van Lil, Niko Sünderhauf, Robert Baumgartl, Peter Protzel
Department of Electrical Engineering and Information Technology
Chemnitz University of Technology
09111 Chemnitz
Germany
{niko.suenderhauf, peter.protzel}@etit.tu-chemnitz.de,
robert.baumgartl@cs.tu-chemnitz.de

Abstract

Controlling and coordinating a heterogeneous and autonomous multi-robot system is still a challenging task. To reduce the required implementation effort, the middleware concept has been introduced recently. This software layer defines unified interfaces and communication services according to the individual robot capabilities. We were faced with the problem of identifying an appropriate middleware software architecture for our heterogeneous multi-robot system designated for outdoor search-and-rescue missions. After comparing different alternatives, we chose Miro as a basic architecture. This article describes our experiences with using and adapting Miro for our purposes. Special attention is payed to the open problem of communication security.

1. Introduction

One of the key research areas at the Institute for Automation at Chemnitz University of Technology are mobile autonomous outdoor robots. Our robots are designated for search-and-rescue (SAR) missions in dangerous environments or inaccessible terrain. This requires the cooperation of several heterogeneous robots with different capabilities, e.g. by dividing up the search space and sharing sensor information or local maps. To improve the fault-tolerance of the multi-robot system, a graceful performance degradation must be ensured by a task reallocation in case of a partial or total robot failure.

Missions are planned and coordinated by human operators who can interact with the robots but do not teleoperate them. Instead, the robots act as autonomously as possible and the operators only intervene in exceptional situations. Figure 1 gives an impression of the robots currently being used. All robots are equipped with the typical range of sensors for outdoor applications (laser scanner, mono and stereo cameras, infrared camera, differential GPS, tilt sensor etc.). One robot has an arm for manipulating objects and moving a camera.

The very heterogeneous and distributed architecture consisting of multiple processing units on every robot, a central control computer and several workstations used by the human operators imposes a major challenge. Thus, the main design objective is to ensure a flexible, fault-tolerant communication infrastructure between all active system components. The amount of communication data varies considerably depending on the current situation. Individual failures of system components, temporal overload conditions and stress-induced errors in human decision-making must be considered and properly handled. On every processing node real-time and off-line activities co-exist.

The rest of the paper is structured as follows. In section 2 we support the selection of a middleware architecture. Additionally, we describe the underlying hardware very briefly. Our experiences with using and extending the chosen middleware Miro are discussed in section 3. Section 4 contains a short analysis of relevant aspects of robot communication security. Finally, the main contributions of the paper are summarized and the next project stages are sketched in section 5.

2. Platform Considerations

2.1 Middleware

A central term in this paper is 'middleware'. In general, a middleware is a software layer that provides services for the communication of distributed applications by transparently defining standardized interfaces and protocols. It provides an infrastructure for integration of applications and data in a heterogeneous and distributed domain. In our domain of mobile robotics the middleware layer should provide interfaces to the different kinds of actuators and sensors that are used on the robots and

Figure 1: Impression of our heterogeneous Multi-Robot System for Outdoor Applications

encapsulate them in a way that high-level software can be easily ported from one hardware (robot) to another. In this way, middleware ensures that high-level code is reusable and can be deployed on different hardware architectures. The middleware layer should also provide communication methods between software modules within one robot or computer but also between modules running on different robots or computers and thus enable a distributed control architecture, inter-robot communication and collaboration.

2.2 Selecting a Suitable Software Architecture

Previous work in our group was based on the ARIA robot API (ActivMedia, 2005). It is released under the GNU Public License (GPL) and well-documented. Unfortunately, ARIA has some limitations, e.g. it is especially designed for the robots offered by ActivMedia with differential drive and a near-zero turn radius. Aria does not provide suitable mechanisms for inter-module communication or even inter-robot communication. As we were heading for a team of heterogeneous robots (including robots with Ackermann steering) and a distributed control architecture, we had to look for an alternative software architecture. Further, the ARIA API changed several times. The resulting porting overhead for our software components was significant.

Several middleware and software architectures and frameworks for the robotics domain had been developed by the community during the past years and many of them would meet our needs. We had to identify the one that would suit our needs best, considering a number of criteria:

What kind of hardware platform is required? Is the software compatible to the existing hardware? Could it be ported to future hardware easily? What language is it written in? What languages can be used in the higher level software modules that have to call the middleware routines? How long will it take new users to familiarize with the middleware? How flexible, in terms of the used actuators and sensors, is it? Does it already support the existing sensors, actuators and robot platforms? How easy is it to extend the support for new sensors or actuators or changing their and the robot's configuration? What license conditions have to be respected? Is the software in use by other groups and well supported by the developers? Could it be maintained locally in case the developers stop working on it? How much effort would it take to switch to the new software?

Developing our own middleware would give us full control and full flexibility, but would require a great effort and would take a long time to define all interfaces from scratch and to implement the support for our heterogeneous robot hardware. So implementing a complete new middleware on our own was never a real option.

Instead, we looked at the available middleware and identified two good candidates that could be used: Player (Gerkey et al., 2003) and Miro (Utz et al., 2002). ORCA 1 (Brooks et al., 2005) and ORCA 2 (Orebäck, 2005) which both emerged from the ORCOS (Orebäck, 2004) project were no real alternatives at this time, as ORCA 1 uses a mixture of inter-module communication mechanisms (CORBA, CRUD and a self-made socket based approach) that seemed too complex to us. ORCA 2 had just started its development and the process of porting the old ORCOS@KTH framework from CORBA to ICE was not finished and we could not foresee whether the outcome would fit our needs. However, if we were faced with the decision today, ORCA 2 would be a good candidate to consider. Some other middleware from the robotics domain came to our attention later on, namely MCA2 (Scholl, 2003) and MARIE (Cote et al., 2006). But these were not considered during the decision process.

Player is maybe the best known middleware architecture available. It is in use by many research groups worldwide and supports a vast variety of sensors and robot platforms including our Pioneer2AT robot. Player is designed in a client-server structure where the robot is represented as a collection of devices that can be controlled via plain TCP sockets. The clients (i.e. the controllers) can be written in any language that can control a socket. Player defines its own message protocol that is used during the communication between the clients and the server (devices). However, the communication between the different clients is completely up to the clients themselves, Player does not provide any methods for inter-client communication. A huge collection of software is available for Player, e.g the client library that offers many standard algorithms used in the robotics community and two simulators, Stage (2D) and

Gazebo (3D).

Miro (the Middleware for Robots) is developed at the University of Ulm, Germany. Unlike Player, which leaves much of the communication between the clients themselves and clients and servers to be implemented by the user, Miro uses CORBA as a powerful communication tool between the different modules on the robot and between several robots. CORBA is widely used in industry and fulfills our needs of flexible inter-module and inter-robot communication. Due to the CORBA interface, it is possible to exchange data and commands between modules written in different languages easily. Miro itself was written in C++ and already supported many sensors that were in use in our group as well as the Pioneer1 platform. Miro is used by the RoboCup team of the University of Ulm and at several other universities in Europe and beyond, but it is not really well known in the community.

The stability of the API was a major concern for our work. We felt a well-established standard as CORBA would be a clear advantage in comparison to a rapidly-evolving project as Player. From a software architect's point of view the CORBA-based middleware is much more elegant and powerful as the more rudimentary Player functionality. Of course, it remains to be analyzed how much overhead is introduced by CORBA but that is beyond the scope of this paper.

The code-base and user community of Miro are not as big as for Player, but Miro provides an interface to Player, so that it is possible to test software written with Miro in the simulators Stage and Gazebo which would otherwise be a clear advantage to Player.

After carefully evaluating the pros and cons of each alternative we selected Miro. This decision was made somewhat arbitrarily, because we did not have any prior experience with Miro or Player. It was decided to add support for all not yet integrated sensors and actuators of our Pioneer 2-AT robot. A second task was the integration of all features of our self-developed mobile platform into the Miro framework.

2.3 Robot Hardware

So far Miro has been adapted to two different robots. The first one is a Pioneer 2 AT from ActivMedia Robotics. The robot has a differential drive with four wheels. It is equipped with sonar range finders, one LIDAR scanner, a pantilt camera, differential GPS sensor, a compass and an inclination sensor. A gripper can be installed as an actuator.

The second robot is self-developed and based on a four wheel electro-scooter that can transport one person. In contrast to the Pioneer 2 AT, this robot has a car steering also known as Ackermann steering. The sensor equipment is comparable to the Pioneer robot, but we use an additional stereo camera system, an acceleration sensor and two LIDAR scanner with one mounted on a tilt unit to provide three-dimensional scans. For maximum flexibility in terms of processing capacity, additional computer modules are very easy to integrate into the system.

3. Experiences with Miro

3.1 Porting Miro

Miro is a CORBA-based framework for programming mobile robots. Its development started in 1999 at the computer science department of the university of Ulm, Germany, and it is being used extensively by the university's RoboCup team "Ulm Sparrows". Its basic structure is depicted in figure 2.

The prospective effort of integrating new hardware devices depends heavily on the type of device. Four basic cases can be distinguished:

1. The hardware is already fully supported by Miro. In this case the associated service can be used with little or no additional effort.

2. Miro already provides support for devices with equal functionality. Then, it is probably possible to reuse the existing interfaces and base classes, leaving only the hardware-specific parts to be implemented.

3. Miro provides support for similar devices. In this case the programmer must derive his own interfaces and implementation classes from the existing ones, adding the missing functionality.

4. There is simply no support of the device or similar ones in Miro. This means that new services must be implemented more or less from scratch.

Examples for all four cases were encountered during the port of Miro to our Pioneer 2-AT robot: Miro did already include support for the robot's predecessor, the ActivMedia Pioneer 1 platform, the Sick laser scanner and the video4linux interface which is used to access the camera image. Apart from creating a Miro service configuration file, which among other things defines the serial interface devices to use, describes the robot's sensor configuration and contains a few constants for the differential drive, there was hardly anything to do to use all these parts of the robot.

Getting the camera controls for panning, tilting and zooming to work proved more difficult: Miro did already define interfaces for PTZ cameras and the Pioneer base service did even implement those interfaces, but the service unfortunately required a Canon camera model to communicate with while our robot was equipped with a Sony camera. There were several changes necessary in order to make our camera work with Miro without breaking the existing implementation:

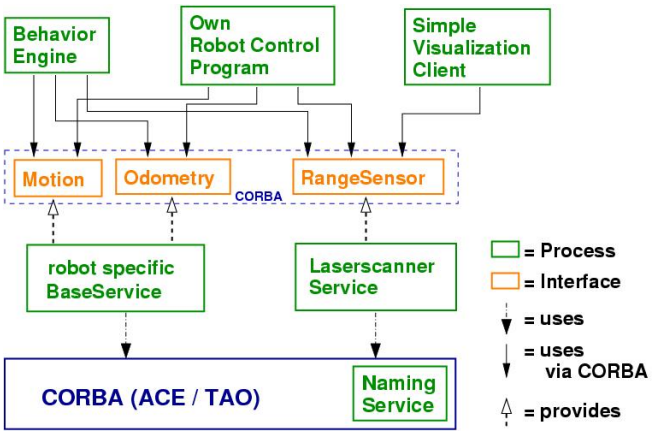

Figure 2: Basic Miro Architecture

- Two new parameters, "Vendor" and "Model" were added to the service's configuration file. This added the required flexibility to the service and even prepared it for future integration of further camera models.
- A handler for the Sony camera's serial protocol had to be implemented.
- The existing "PanTilt" interface met our requirements pretty well, it could be reused, including the basic implementation.
- The existing "CanonCamera" interface on the other hand was, as the name suggests, pretty closely tailored to the Canon camera's set of functions. Apart from that the interface had a few more deficiencies, e.g. the zoom factor was specified as a percentage instead of a device-independent physical size. As a consequence, a generic "CameraControl" interface was designed that combines the (presumed) shared functionality of all PTZ cameras. Specialized interfaces for the Canon and Sony camera models were derived from this generic interface, adding some more methods for each particular model.

The interface redesign described above required some serious changes in the existing Miro service (and existing clients as well). The Miro developer team was very helpful and did a lot of work to support the task.

The remaining sensors (a compass module and a GPS receiver) were completely new to Miro, mainly because most supported robot models were designed for indoor use only. All interfaces had to be designed from scratch. Implementing the services was greatly eased by existing Miro classes for certain tasks, e.g. for serial device communication.

Analyzing the existing Miro source code and implementing all extensions described above were achieved in the course of a six-month diploma project. Not counting comments, the patches sent back to the developer team amount to about 6,000 to 7,000 lines of C++ source code. All changes were accepted by the developers and integrated into the official Miro source tree.

To port Miro to our second, self-constructed mobile platform, which was not supported in any way by Miro, it was necessary to create a new interface for the Ackermann steering. This interface was derived from the existing "Motion" interface. The "Motion" interface only provides operations to set the translational and the rotational velocity which is generic enough to convert these to a steering angle. The second part of porting Miro to the mobile platform was the "Odometry" interface. The user or robot control algorithm can query through this interface the current velocity and position in a virtual world coordinate system. The readings from the incremental position encoders in each front wheel and the actual steering angel are used to calculate the current position in the coordinate system.

3.2 pyMiro - A Python Binding for Miro

pyMiro is a Python binding for Miro that is developed in our group. It enables us to use Python for rapid-prototyping algorithms for our robots. Code development in Python is faster, more bug-free (Prechelt, 2000) and simply more fun and less exhausting than implementing everything in C++. In the current version it offers simple calls to get references to all Miro-objects like sensors, actuators and motion control and can handle the events created by the different modules. pyMiro can be seen as a Python wrapper around the Miro functionality written in C++. Wrapping this functionality is very easy due to the integrated CORBA communication mechanisms in Miro. In our experience, it is much easier (especially for students from other fields than computer science) to start programming the robots in an interactive way, as it is offered by the Python shell. pyMiro especially hides much of the internal workflow that would otherwise be needed to take care of when writing clients to control the robots. So with pyMiro our students get a good start into robotics and are not discouraged by writing complicated C++ code. A simple example should demonstrate the ease of rapid-prototyping algorithms with Miro and pyMiro:

```
import pyMiro

sonar=pyMiro.getSonar()
loko=pyMiro.getMotion()

while(1):
        scan=sonar.getFullScan()
        front_scan=scan.range[0]

        if min(front_scan)<300:
                loko.limp()
```

```
else:
        loko.setLRVelocity(50,50)
```

This simple script just moves the robot forward until the closest obstacle detected by the sonar sensors is closer than 300mm, in which case the robot would simply stop. We only need these few lines of code in pyMiro. The same functionality written in C++ takes many more lines of code.

4. Security Aspects

The robots and operators communicate via an IEEE 801.11 b/g wireless network which is part of the university campus network. Strictly using secure services (e.g. SSH for remote access to computers) ensures privacy and the integrity of user data. The usage of insecure services as telnet or ftp is strongly discouraged but not forbidden.

Unfortunately, client-server communication via CORBA is not encrypted nor protected in any way. CORBA is an open standard, the interface specifications (IDL files) for Miro and even binary client programs are freely available.

As a consequence, all Miro services are accessible from within the university campus network. Thus, everyone in the university network could send a request to get the sensor readings (e.g. ranges, GPS coordinates, camera images). This is not a problem as long as the robots are used solely in research contexts. However, a potential risk arises when access to actuators (e.g. the robot arm, gripper or drive motor) is granted to unauthorized users. Attackers familiar with the system could damage equipment and even harm people. One of the robots weighs more than 100kg and reaches a velocity of 20 km/h.

Therefore, mechanisms are required to restrict access to the system internals (at least the actuators) to trusted persons. Users communicating with the robot must be authenticated. Further, the integrity of transmitted commands has to be guaranteed. Confidentiality is not necessary, because the transmitted information is not secret. Therefore, access control for sensors is not required.

4.1 Virtual Private Network

A simple solution could be to setup a virtual private network (VPN) and connect all robots, workstation computers and the master computer to it. Access to that VPN is granted to authorized persons and robots only. All network traffic is encrypted, which assures also confidentiality. Software for VPNs is available and stable. Apart from client and server software installation and configuration, no additional maintenance efforts are needed. Furthermore, this approach prevents man-in-the-middle attacks.

One potential disadvantage is the need for a so-called VPN access concentrator node when more than two nodes constitute the VPN. This effectively reduces the available communication bandwidth by a factor of two in shared media as WLANs. Considering the huge amount, different nature (sensor data, communication between robots, robot control) and potential real-time requirements of communication data in critical situations, this reduction seems somewhat problematic.

Another potential drawback is the significant processing power required for data encryption and decryption. The computing capacity of robots is limited and must be preserved for robot control algorithms. Encrypting all network traffic would impose a significant load onto the robot's CPUs (Ravi et al., 2004). A potential solution could be offloading security processing to a dedicated processor (e.g. a DSP) or to a security processing unit within the CPU (e.g. the VIA Eden's padlock engine).

4.2 Use of Credentials

Because Miro is based on CORBA, another viable approach is to add an additional parameter to all security-relevant functions. This parameter contains a credential, e.g. a password. This credential from the client needs to be validated at each function call. Furthermore a database to manage the credentials must be implemented. The credential must be transmitted in encrypted form. Additionally, the integrity of the remaining function parameters must be kept intact.

This approach requires much more implementation effort, but imposes justifiable extra cost at run time in terms of processing power and memory space.

4.3 CORBA Security Service

A third interesting possibility is the usage of the CORBA Security Service. Miro uses "The ACE ORB" (TAO) as CORBA implementation (Gokhale and Schmidt, 1999). Unfortunately, TAO only provides basic security services so far. We consider it an easy task to replace the commonly used Internet Inter ORB Protocol (IIOP) by the IIOP over SSL implementation (SSLIOP). This ensures confidentiality for all remote method invocations between two ORBs. Also authentication is done via client and server X.509 certificates but this is an all-or-nothing approach; all or no services in a process are secured. There is no simple way yet to distinguish between several levels of security, or to encrypt only selected services. This results in the same drawback as for VPNs: encrypting and decrypting all data requires significant processing power which is precious in mobile environments. However, CORBA defines the concept of Portable Interceptors which solves this problem (Group, 2001). This could be implemented in a way that is totally transparent to the clients and servers. Only the base class of clients and servers from Miro would have

to be modified. This approach is currently under investigation.

5. Summary & Outlook

Integrating new hardware into Miro did not impose major problems so far. Miro proved to be a flexible and extensible architecture. Newly-developed sensors and actuators can be very rapidly integrated into the framework.

The only drawback from our point of view is the weak security support of Miro. Either significant processing overhead or a major programming effort is inevitable. The use of Portable Interceptors could be a possible solution.

In the near future we will concentrate on porting algorithms for sensor fusion to Miro and the design of an appropriate robot control architecture. One possible approach could be the use of the behaviour engine already available in Miro.

References

ActivMedia, I. (2005). ARIA ActivMedia Robotics Interface for Applications. Available from: http://www.activrobots.com/SOFTWARE/aria.html. Accessed: 2005-07-12.

Brooks, A., Kaupp, T., Makarenko, A., Orebäck, A., and Williams, S. (2005). Towards component-based robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005)*.

Cote, C., Brosseau, Y., Letourneau, D., Raievsky, C., and Michaud, F. (2006). Robotic software integration using marie. *International Journal of Advanced Robotic Systems - Special Issue on Software Development and Integration in Robotics*, 3(1):55–60.

Gerkey, B. P., Vaughan, R. T., and Howard, A. (2003). The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, Coimbra, Portugal.

Gokhale, A. and Schmidt, D. (1999). Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM '99)*, New York.

Group, O. M. (2001). Interceptors Published Draft with CORBA 2.4+ Core Chapters. Available from http://www.omg.org/docs/ptc/01-03-04.pdf. Accessed: 2005-07-15.

Orebäck, A. (2004). *A Component Framework for Autonomous Mobile Robots*. PhD thesis, KTH, Stockholm.

Orebäck, A. (2005). ORCA2 - Components for Robotics. `http://orca-robotics.sourceforge.net/`.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *Computer*, 33(10):23–29.

Ravi, S., Raghunathan, A., Kocher, P., and Hattangady, S. (2004). Security in Embedded Systems: Design Challenges. *ACM Transactions on Embedded Computing Systems*, 3(3):461–491.

Scholl, K.-U. (2003). MCA2 - Modular Controller Architecture. `http://mca2.sourceforge.net`.

Utz, H., Sablatng, S., Enderle, S., and Kraetzschmar, G. (2002). Miro – Middleware for Mobile Robot Applications. *IEEE Transactions on Robotics and Automation*, 18(4):493–497.