

An Object-Oriented Realtime Framework for Distributed Control Systems

A. Traub*
R. D. Schraft

Fraunhofer Institute
Manufacturing Engineering and Automation
Nobelstraße 12, 70569 Stuttgart (Germany)

This paper presents the 'Realtime Framework', a package of software modules for building distributed real-time control systems in robotics and automation. The Realtime Framework covers the areas of client-server communication, control of program flow and modality through messaging and state machines, and low-level input/output. In addition, it contains a real-time utilities package and wrappers for operating system calls, which shield any operating system dependence from the application built on top of the Framework. Software design patterns were used to document solutions for recurring problems in control systems involving several related classes. The Realtime Framework is mostly aimed at increasing software modularity, portability, and re-usability, thereby reducing software development costs. This paper gives an overview of the major features of the Realtime Framework and summarizes practical experiences from applying it to a mobile robot control system.

1. Introduction

Modern control systems in robotics and automation usually include a large number of different sensors and actuators, and require a considerable amount of data processing capacity. "Intelligent" sensors with micro-controllers help to reduce data transmission rates by pre-processing sensory information. Likewise, "intelligent" actuators can be controlled by high-level commands. Distributed control systems integrate "intelligent" sensors and actuators together with data processing units like computers and micro-controllers, achieving a high level of hardware modularity.

However, the procedural software design techniques which are still widely used in industrial control systems are often unable to cope with the complexities arising from heterogeneous distributed computing. Software, which is not designed in a thoroughly modular way, is hard to develop with large teams of programmers and

difficult to adapt to new hardware platforms or new functional requirements. In addition, the robustness of procedural software suffers from the difficulty of testing individual software components independently of others. Besides procedural software design techniques, the second major problem in current control system development is that the communication protocols in distributed systems still vary widely depending on the manufacturer, the operating system and the particular network or bus system used. „Plug and Play“ is therefore next to impossible for manufacturers of control system components. All the above factors lead to excessive costs for software development and maintenance.

In order to reduce these costs while maintaining a high level of software quality, the design goals for control software therefore have to include:

- easy and efficient communication between different hardware platforms on bus systems and networks commonly used in automation industry
- re-usability of software components
- portability between operating systems and hardware platforms
- easy familiarization of new team members with the development process and existing software
- robustness towards internal software errors and unforeseen external conditions
- high run-time performance

This paper presents the *Realtime Framework*, a framework of object-oriented software components which specifically addresses the above requirements. The *Realtime Framework* allows an easy and efficient implementation of the communication processes typical for distributed control systems. The hardware components can be connected over a TCP/IP network or a CAN [1] fieldbus, which is widely used in industrial automation systems. The modular structure of the framework increases software re-usability and completely shields any operating system dependence from the application specific software which uses the framework. In addition to

* Corresponding author; e-mail: traub@ipa.fhg.de

software components, it also includes software design patterns [2] which reduce the initial period of new members in the development team. Throughout the framework development, the issues of software robustness and real-time performance have been considered.

In the section 2 of this paper, related approaches to this topic are reviewed. Section 3 describes the overall architecture of an application built with the *Realtime Framework*, and section 4 describes the *Realtime Framework* components in detail. After discussing some implementation details in section 5, we describe a sample application of the Framework for mobile robot control in Section 6. Conclusions are summarized in section 7.

2. Related Work

Several frameworks for control systems have been developed, among them NEXUS [3] and OSACA [4]. They are similar to the *Realtime Framework* presented in this paper in that they use client/server architectures for communication. NEXUS is a decentralized communication system with hierarchical error recovery for robustness. It was developed for mobile robotics but is generic enough for use in other control systems. OSACA is a communication framework for distributed control applications under Windows NT and VxWorks. Both NEXUS and OSACA are limited to TCP/IP protocols and cannot be directly used for communication over industrial fieldbus systems widely used in automation, like CAN, [1], Interbus [10] and Profibus [11].

Commercially available products are ControlShell™ [5] and Rhapsody™ [6]. They feature graphical editors for data flow and state diagrams, from which source code can be generated. Automatically generated source code however tends to be hard to read and understand during debugging. Therefore, this strategy was not adopted for the *Realtime Framework*. ControlShell™ is particular in that it does not use a client/server architecture for communication, but a dissemination system called NDDS. Like NEXUS and OSACA, ControlShell™ [5] and Rhapsody™ cannot be used for communication over industrial fieldbus systems.

ACE [7] is a general communication framework for client/server architectures, mainly focused on TCP/IP networks. It is freely available in source code for several different Windows and UNIX/POSIX platforms and well documented. Among the frameworks presented in this section, it is the only one which explicitly uses software design patterns. In contrast to the *Realtime Framework* however, it does not include convenient classes for the kind of high-level command and data exchange typical in distributed control systems.

Several organizations have developed communication standards for specific fieldbus systems, among them CANOpen [1], DeviceNET [8,9], Interbus [10] and Profibus [11]. A promising new standard is OLE* for Process Control (OPC) [12]. Commercial OPC client and server software is available for a variety of network and fieldbus systems. The main drawback of OPC is that it was specifically designed for Microsoft Windows operating systems. The Windows DCOM feature, on which OPC is based, is currently not even included in Microsoft's only real-time operating system, Windows CE [13].

Although not specifically designed for manufacturing environments, the Common Object Request Broker Architecture (CORBA) is gaining popularity in automation industry. A Real-time version of CORBA based on ACE [7] is freely available.

As long as these protocol standardization issues are not resolved, manufacturers of distributed control systems only have the choice of developing software for a specific communication standard or of introducing an extra layer in their software, which can be adapted to different protocols. The latter approach was adopted for the *Realtime Framework*. As shown in the following sections however, the functionality of the *Realtime Framework* goes far beyond that of protocols like OPC or CORBA in that it also provides mechanisms for very efficient client-server communication between processes and threads running on the same machine, design patterns for integrating re-useable components into control applications, and other things.

3. Architecture of a *Realtime Framework* Application

The *Realtime Framework* is designed as a layer of components, on top of which real-time applications and other libraries for more specific areas can be built (Fig. 1). At the time of writing, a library for mobile robotics based on the *Realtime Framework* was available, including components like self-localization and path planning. With this architecture, the amount of application specific code can be kept small by using the framework and other libraries.

Another advantage of this architecture is that all operating system dependent code is concentrated in a thin layer of the *Realtime Framework*, which greatly facilitates porting the application to a new operating system. The runtime overhead of this extra layer can be kept small by using C++ inline functions for operating system calls.

* Object Linking and Embedding

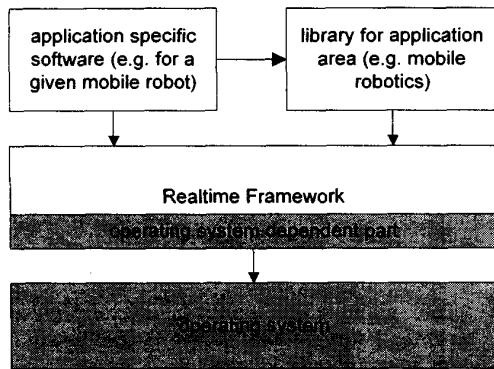


Fig. 1 Architecture of a typical *Realtime Framework* application. Arrows symbolize a 'uses'-relationship. Operating system dependent parts are shaded.

4. Components of the *Realtime Framework*

As shown in Fig. 2, the *Realtime Framework* is organized into several packages of classes which cover the areas common in most distributed control systems. The arrows indicate a "uses"-relationship, which is equivalent to "depends on". Note that circular dependencies have been avoided, so that for example the I/O package and the Utilities packages can be tested and used without the Client/Server and the Program Structure/Program Flow packages.

The I/O package mainly provides an interface to operating system dependent I/O operations for TCP/IP, serial communication and CAN fieldbus. Low-level functions like diagnostic output are concentrated in the Utilities package.

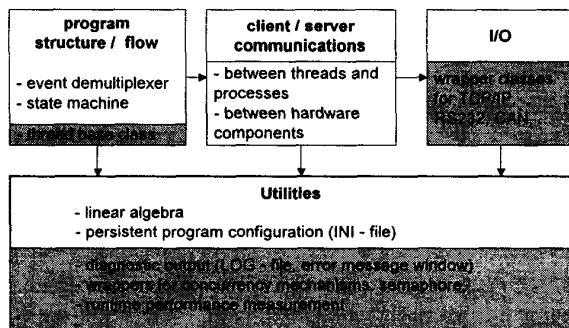


Fig. 2: Structure of the *Realtime Framework*. The Framework classes can be grouped into four packages. Arrows symbolize a 'uses'-relationship. Operating system dependent parts are shaded.

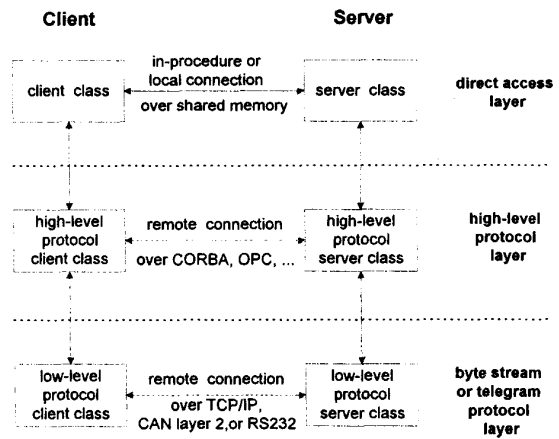


Fig. 3: Layers of client-server communication. If the client-server connection is between two threads of the same process (in-process) or between two processes running on the same machine (local), the communication can be handled in the topmost layer using shared memory. Otherwise, the data is passed on to the middle communication layer, which handles remote connections over high-level protocols like OPC or CANOpen, or to the lower layer, which implements communication over byte stream or telegram protocols.

As mentioned earlier, a major area of concern in current distributed control system development is communication between different hardware and software units. The client/server package implements the major types of client-server communication like data or command exchange and event notifications. In addition, a remote procedure call mechanism was developed. The communication can be either synchronous (blocking) or asynchronous (non-blocking). A priority can be assigned to each data package, which determines the order in which it will be handled by the recipient. In this way, emergency notifications can be handled very fast by assigning a high priority. Due to the layered structure of the client/server package (Fig. 3), the communication can take place between two threads of the same process (in-process), between two processes running on the same machine (local) or between different machines (remote) using several high-level or low-level protocols. In all cases, the same simple and flexible client/server class interface can be used to send or retrieve data. For example, the type of data transmitted between a client and a server can be conveniently specified by a C++ template parameter. Both simple data types like integer or floating point numbers and complex types (data structures) are allowed. Also, arrays of simple and complex data types can be declared. The client/server package thus allows separation of the communication content from the specific kind of communication protocol used, which is a major step towards software modularity and re-usability.

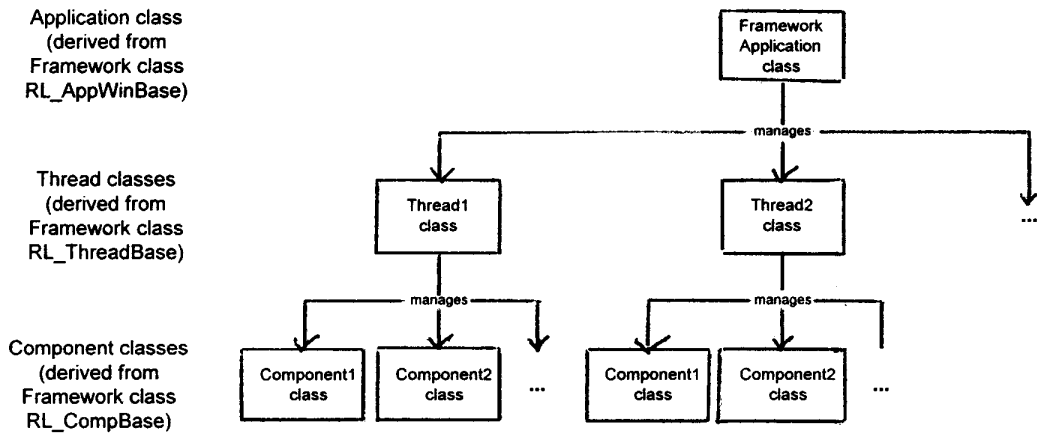


Fig. 4: *Management Tree* design pattern. Application specific classes (rectangles) are derived from corresponding Framework classes, which implement the functionality needed for management.

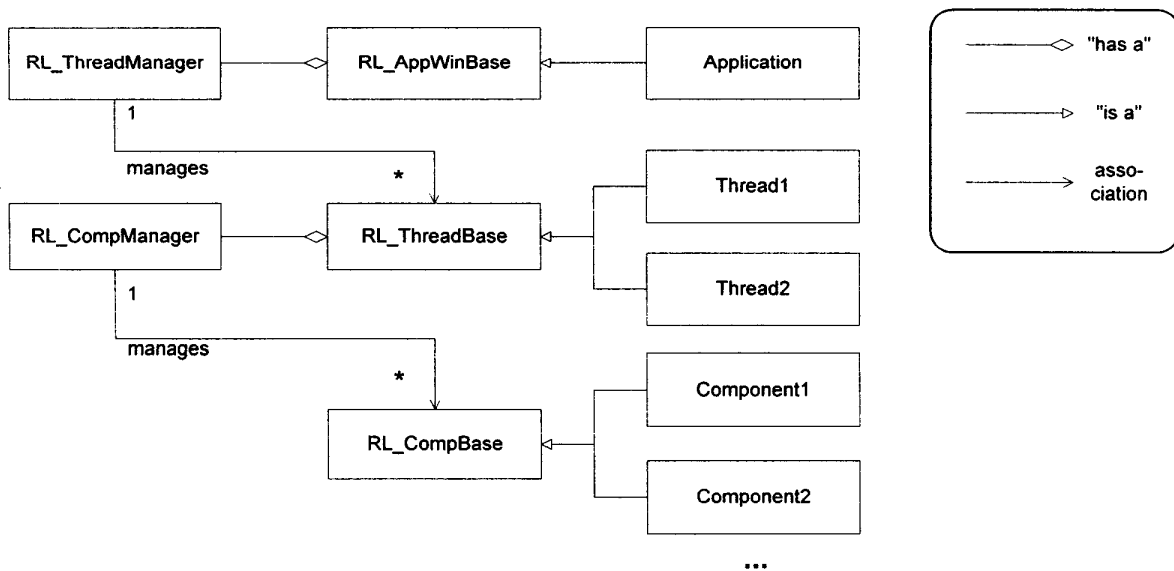


Fig. 5: UML diagram of the classes involved in the *Management Tree* design pattern (simplified). The left and center column of rectangles are *Realtime Framework* classes. Application specific classes are on the right. The application base class `RL_AppWinBase` contains an object of type `RL_ThreadManager`, which encapsulates the functionality needed to manage an arbitrary number of thread classes (derived from the thread base class `RL_ThreadBase`). Similarly, each thread class contains a `RL_CompManager` object to manage components.

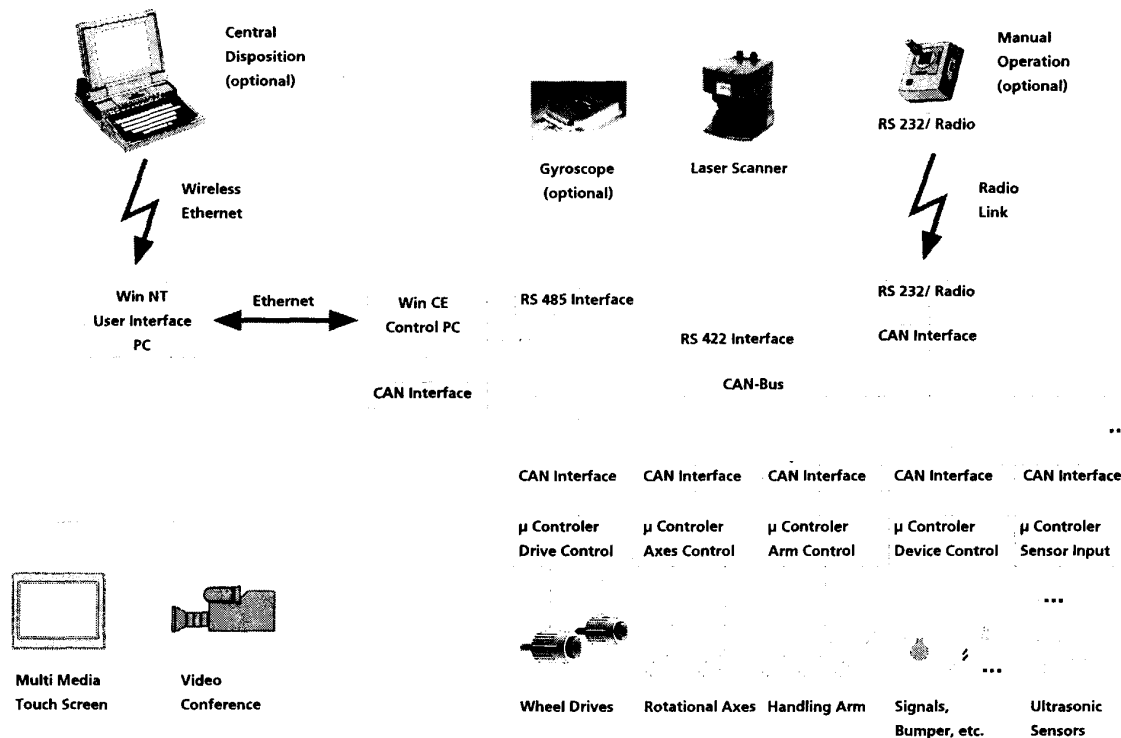


Fig. 6: Distributed control system of Care-O-bot™. Care-O-bot™ is a mobile home care system for elderly and disabled people with a two-wheel driving kinematics and a several axes for object manipulation. Its distributed control system consists of three Pentium PCs and several micro-controllers connected over TCP/IP and a CAN fieldbus.

Since in-process and local connections were implemented efficiently using shared memory, there is very little runtime overhead in using the standardized client and server class interfaces even within a process. In chapter 6, an example is given of how they can be used to structure the control and data flow in an application.

Another recurrent problem in control software development which is often underestimated is fluctuation in the development team. Modern control systems are so complex that it takes a considerable amount of work to understand modules and applications written by someone else. This leads to the commonly known phenomenon that software components are rather re-invented than re-used. Besides a good documentation, software design patterns [2] can help to alleviate this problem. Design patterns describe the organization and interaction of several related classes in an informal way. For example, they are useful for documenting recurrent micro-architectures of classes that have proven useful to build control software.

An important design pattern developed in conjunction with the *Realtime Framework* is the *Management Tree* (Fig. 4), which is used as a skeleton for most applications

built with the *Realtime Framework*. In this pattern, application specific classes, which are derived from corresponding *Realtime Framework* classes, are arranged in a tree-like hierarchical structure (all *Realtime Framework* classes start with *RL_* to avoid namespace pollution). Note that the arrows in Fig. 4 do not symbolize an inheritance relationship, but rather a “manages” association*, where “manages” refers primarily to initialization, start/execution, reset and de-initialization. The management associations are created during program start-up with function calls, which register the components with their threads and the threads with the application class. Fig. 5 shows how the design pattern is supported by the *Realtime Framework*. If, for example, one component detects an error that it cannot handle, it notifies the *RL_AppBase*-derived class, which then resets all other threads and components, allowing for a robust response to unforeseen conditions. The usage of the *Management Tree* design pattern greatly facilitates understanding the critical phases of program start-up,

* Association is used here in the sense defined by UML [15]

reset and shutdown in *Realtime Framework* applications developed by other people.

Besides the classes used in the *Management Tree* design pattern, the program structure/program flow package includes components like a configurable state machine class and the *Reactor* design pattern [14] with an event demultiplexer for flexible, priority-based handling of simultaneous events within a program thread.

5. Implementation Details

The *Realtime Framework* has been designed with the Unified Modeling Language (UML) [15] and was implemented in C++ on Windows NT and Windows CE. During its development, portability to POSIX-compliant operating systems like VxWorks has been ensured. For example, the Windows `WaitForMultipleObjects()` system call has been avoided, since it is not supported by POSIX.

Realtime Framework applications are created by statically or dynamically linking *Realtime Framework* libraries with the rest of the application and by calling an initialization function during program start-up. Most applications based on the *Realtime Framework* also derive several application specific classes from Framework classes according to the *Management Tree* and other design patterns (see section 4).

The real-time capabilities of operating systems like Windows CE are preserved in the *Realtime Framework* by guaranteeing deterministic response times for Framework function calls. In addition, the runtime performance and memory efficiency of the Framework were increased by avoiding excessive use of classes with automatic type conversion and memory allocation functions, like string classes.

6. Application of the *Realtime Framework* on a Mobile Robot

The *Realtime Framework* was used as basis for the control software of Care-O-bot™ (Fig. 7, [16]), a mobile home care system for elderly and disabled people which is being developed at the Fraunhofer Institute IPA. Care-O-bot™ was designed with a two-wheel driving kinematics and a multimedia touch screen as a mobile communications terminal and walking aid. A future model with a manipulator arm will also be able to perform simple household tasks like bringing medicine to a bed. The control system of Care-O-bot™ (Fig. 6) includes three PCs and several „intelligent“ sensors and actuators connected over a TCP/IP network and a CAN fieldbus. The *Realtime Framework* was used as the base layer of the control software architecture. It manages

communication between the different hardware units and provides other real-time control functionality for the applications running on the PCs.

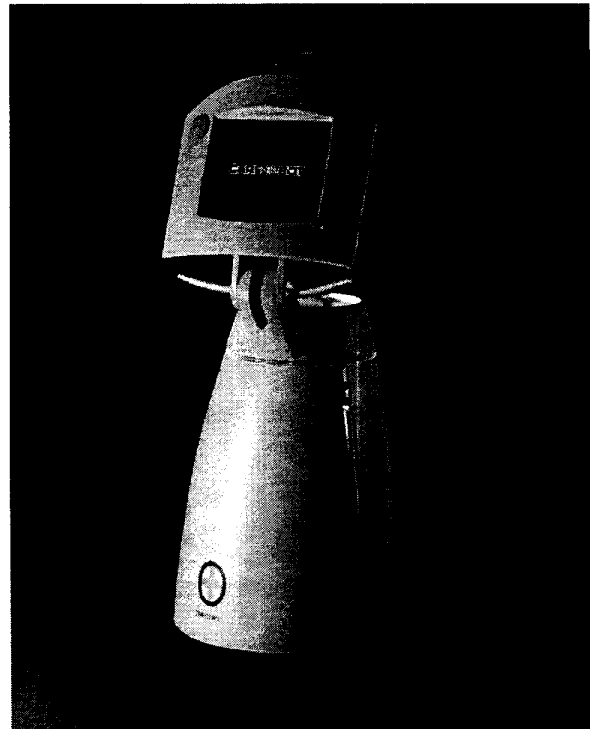


Fig. 7: Care-O-bot™. Below the moveable multimedia touch screen (top) are two handles for use as a walking aid.

The user interface PC accepts input from the user via touch screen or speech recognition and provides audiovisual feedback. It is connected to the control PC over a client-server connection, which is used to write user commands and to read robot status data (Fig. 8). Similarly, the control application is divided into several threads (tasks) which communicate over in-process client-server interfaces. This causes very little runtime overhead, since in-process and local connections are implemented very efficiently using shared memory.

Each thread contains several components according to the *Management Tree* design pattern (Fig. 4). The Motion Control Thread for example contains re-useable components for position interpolation, position control, inverse kinematics, and so on.

Besides the Control Process, a simple Watchdog Process is also running on the control PC, which supervises the correct operation of the safety-relevant part of the control application (motion control including obstacle detection) through a local client-server connection. Since these two processes are running in different address spaces under protected mode, the Watchdog Process cannot be corrupted even when there are serious programming flaws in the control application.

During the software development with up to six people working in parallel, the client-server interfaces of the *Realtime Framework* have proven very useful in integrating and re-using software components written by different people. Also, the *Management Tree* and *Reactor* design patterns were used to implement and document micro-architectures recurring in control software. They greatly facilitated the integration of developers familiar with other *Realtime Framework* applications into the team.

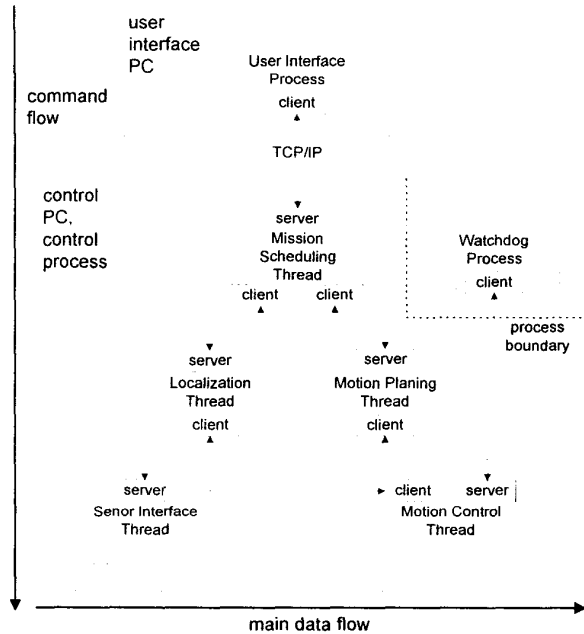


Fig. 8: Using client-server interfaces to structure the control and data flow between the processes and threads running on the user interface PC and the control PC of Care-O-bot™. The command flow is implemented through chains of client-server connections (top to bottom). The data flow is mostly directed from left to right. It can go from client to server or vice versa using either client write or client read operations.

7. Conclusions

The *Realtime Framework* presented in this paper addresses the need for new software architectures and design strategies in distributed industrial control systems. It provides a simple interface for client-server communication not only between different hardware units but also between processes and threads running on the same machine. In the latter case, the data exchange is extremely efficient, since it is implemented through shared memory. Applications based on the *Realtime Framework* are shielded from any operating system dependence, which enhances portability and re-use of

software components. Design patterns were developed in order to express and document common structures on a micro-architectural level between *Realtime Framework* applications.

Practical experiences with the *Realtime Framework* were collected during the development of a mobile robot with several data processing units connected over a TCP/IP network and a CAN fieldbus. It was found that object-oriented frameworks and design patterns can significantly reduce software development and maintenance costs while maintaining a high software quality.

Acknowledgments

This work is supported by the Bundesministerium für Forschung und Technologie, Germany, within the project IQ2000.

References

- [1] CAN in Automation e.V. Home Page, <http://can-cia.de>
- [2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, „Design Patterns: Elements of Reusable Object-Oriented Software“, Addison-Wesley, Reading, 1995
- [3] J. A. Fernandez, J. Gonzalez, „NEXUS: A Flexible, Efficient and Robust Framework for Integrating Software Components of a Robotic System“, Proceedings of the 1998 IEEE ICRA, Leuven, Belgium, 1998
- [4] OSACA Association, „The OSACA Project at a Glance“, <http://www.osaca.org/>
- [5] Real-Time Innovations Inc (RTI) and Stanford University, „Control Shell: Object-Oriented Framework for Real-Time System Software“, <http://128.102.240.17/products/cs.html>, 1996
- [6] I-Logix Inc., Rhapsody product overview, http://www.ilogix.com/fs_prod.htm
- [7] Schmidt, D. C., „ACE: an Object-Oriented Framework for Developing distributed Applications“, in Proceedings of the 6th USENIX C++ Technical Conference, Cambridge, Massachusetts, USENIX Association, April 1994; see also: <http://www.cs.wustl.edu/~schmidt/ACE.html>
- [8] The Open DeviceNET Vendors Association, <http://208.147.96.36/mrop/Organizations>
- [9] „interesting links and information to DeviceNET“, http://www.infoside.de/infida/wissen_devicenet.htm
- [10] INTERBUS Online, <http://www.ibsclub.com/index.htm>
- [11] Profibus Home Page, <http://www.profibus.com>
- [12] OPC Foundation Home Page, <http://www.opcfoundation.org/>
- [13] Microsoft Windows CE product information, <http://www.microsoft.com/windowsce/default.asp>
- [14] Schmidt, D. C., „The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2)“, C++ Report, vol. 5, Sept. 1993
- [15] G. Booch, J. Rumbaugh, I. Jacobsen, „Unified Modeling Language User Guide“, Addison Wesley, Longman, 1997
- [16] R. D. Schraft, C. Schaeffer, T. May, „The Concept of a System for Assisting Elderly or Disabled Persons in Home Environments“, Proceedings of the 24th IEEE IECON, Vol. 4 Aachen (Germany), 1998