Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2009/2010

Praca dyplomowa magisterska

Michał Skrzędziejewski

RPC for Embedded Systems

Opiekun pracy:
prof. nzw. dr hab. Cezary Zieliński

Ocena ....

....
Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Specjalność: Informatyka –
Inżynieria oprogramowania
i systemy informacyjne

Data urodzenia: 28 Lipiec 1983 r.

Data rozpoczęcia studiów: 1 października 2007 r.

Życiorys

Nazywam się Michał Skrzędziejewski. Urodziłem się 28 Lipca 1983 r. w Warszawie. Ukończyłem XIV Liceum Ogólnokształcące im. Stanisława Staszica. W roku 2007 uzyskałem dyplom inżyniera na kierunku "Electrical and Computer Engineering" prowadzonym na Wydziale Elektroniki i Technik Informacyjnych. Od 2008 roku pracuję w firmie GTECH Polska jako programista. Moje zainteresowania z dziedziny informatyki obejmują między innymi programowanie systemów biznesowych. Interesuję się również elektroniką, a w szczególności programowaniem mikrokontrolerów.

....
podpis studenta

Egzamin dyplomowy

Złożył egzamin dyplomowy w dn. ....

Z wynikiem ....

Ogólny wynik studiów ....

Dodatkowe wnioski i uwagi Komisji ....

Streszczenie

Tytuł: *RPC dla systemów wbudowanych*

Praca ta prezentuje szkielet KIPER realizujący mechanizm RPC (Remote Procedure Call - zdalne wywołanie procedury). KIPER zbudowany jest w oparciu o bibliotekę Protocol Buffers. Składa się on z dwóch modułów: KIPER-C (napisany w języku C, przeznaczony dla urządzeń wbudowanych) oraz KIPER-CPP (napisany w języku C++, przeznaczony dla komputerów klasy PC). Testy szkieletu KIPER odbyły się na istniejącym stanowisku sterującym z robotem przemysłowym. System MRROC++ (szkielet ramowy programowania robotów) został uruchomiony na dwóch komputerach klasy PC. Wykonywanym zadaniem było śledzenie konturu. Pomiary sił/momentów sił wymagane przez zadanie były uzykiwane poprzez zdalne wywołanie procedury przeprowadzone na sterowniku czujnika sił/momentów sił. Sterownik, zbudowany w oparciu o mikrokontroler z rdzeniem ARM7, pełnił rolę serwera RPC dostarczającego odczyty z czujnika. Wyniki testów pokazują, iż jest możliwe wykorzystanie mechanizmu RPC nawet w tak wymagającym scenariuszu.

Słowa kluczowe: *rpc, systemy wbudowane, mikrokontroler, robotyka, protocol buffers, mrroc++*

Abstract

This thesis describes KIPER, an RPC (Remote Procedure Call) framework based on the Protocol Buffers library. The framework consists of two modules: KIPER-C (written in in C, intended for embedded devices) and KIPER-CPP (written in C++, intended for PC class computers). An existing robot-control setup was employed as a benchmark for KIPER. Two networked computers were running MRROC++ (robot programming framework) which executed the contour-following task. The force/torque measurements required as an input for the task were obtained by means of a remote procedure call whose target was the force/torque sensor controller, an ARM7 microcontroller-based device interfacing with the force/torque sensor. The force/torque sensor controller acted as an RPC server which delivered readings from the sensor. The results of the tests show that it is feasible to use RPC as a means of communication even in such demanding scenario.

Key words: *rpc, embedded, microcontroller, robotics, protocol buffers, mrroc++*

Contents

1. Introduction . . . 1
1.1. Background . . . 1
1.2. Motivation . . . 3
1.3. Aim of the thesis . . . 4
1.4. Scope . . . 5
1.5. Conventions . . . 6
1.6. Outline . . . 6

2. Background . . . 8
2.1. Remote procedure call . . . 8
2.1.1. RPC in embedded systems . . . 12
2.1.2. Remarks . . . 13
2.2. Communication technologies . . . 13
2.2.1. RS-232 . . . 14
2.2.2. RS-485 . . . 14
2.2.3. CAN . . . 14
2.2.4. USB . . . 15
2.2.5. Ethernet . . . 16
2.2.6. EtherCAT . . . 17
2.2.7. Bus arbitration . . . 18

3. Requirements . . . 19
3.1. High level requirements . . . 19
3.2. Platform requirements . . . 20
3.3. Other requirements . . . 21

4. Design . . . 22
4.1. Packet encoding/decoding . . . 24
4.2. Message flow . . . 25
4.3. Choosing the transport mechanism . . . 26
4.4. Simple protocol for obtaining force/torque measurements . . . 27

5. Implementation . . . 29
5.1. Overview . . . 29
5.2. KIPER-CPP . . . 29
5.2.1. Employed technologies and standards . . . 29
5.2.2. KIPER-CPP API . . . 34

5.3. KIPER-C . . . 35
5.3.1. Employed technologies and standards . . . 35
5.3.2. Configuring KIPER-C . . . 38
5.3.3. KIPER-C API . . . 39
5.4. Integration with MRROC++ . . . 39
5.5. Force/torque sensor controller hardware . . . 41
5.6. Force/torque sensor controller firmware . . . 42
5.7. Components integration . . . 45
5.8. Source code remarks . . . 47

6. Using KIPER . . . 48
6.1. Using KIPER-CPP . . . 48
6.1.1. Prerequisites . . . 48
6.1.2. Building and installation . . . 48
6.1.3. Example usage . . . 50
6.1.4. RPC Server . . . 54
6.1.5. Using CMake to compile .proto files . . . 56
6.1.6. Transport mechanisms . . . 58
6.2. Using KIPER-C . . . 59
6.2.1. Building . . . 59
6.2.2. Example . . . 59
6.3. The LPC-P2378-SK firmware . . . 61
6.3.1. Prerequisites . . . 61
6.3.2. Building . . . 61
6.3.3. Logging . . . 62

7. Testing . . . 63
7.1. Initial tests . . . 63
7.2. Testing with force/torque sensor controller . . . 64
7.3. Testing with MRROC++ . . . 67
7.4. Results and analysis . . . 71

8. Summary and conclusions . . . 73
8.1. Requirements validation . . . 73
8.2. Limitations of the solution . . . 73
8.2.1. Real-time guarantees . . . 73
8.2.2. Location transparency . . . 74
8.2.3. Reliability . . . 74
8.2.4. Large messages . . . 75
8.2.5. High bandwidth . . . 75
8.2.6. Security . . . 75
8.3. Problems . . . 75
8.4. Further development . . . 75

A. Glossary . . . 77

Bibliography 79

1. Introduction

1.1. Background

In our daily lives we are, often unknowingly, surrounded by the ubiquitous embedded devices — tiny electronic appliances performing one or few dedicated functions. Indeed, not many people realise that their mobile phone or portable MP3 player contains a miniature, power-efficient, portable computer frequently running sophisticated software.

Most embedded devices exist in isolation as far as the communication with other systems is concerned — a modern home appliance such as dish washer does not need to communicate with any other system (the exception being some smart home automation systems). On the other hand, an embedded system that measures forces/torques acting on a robotic manipulator's gripper needs to send the measurements to the computer that controls the robot with a high frequency. Making the embedded devices communicate with outside world becomes more painless as contemporary embedded systems often contain abundance of various interfaces. Even relatively cheap microcontroller devices (basing on which many embedded systems are built) provide communication ports formerly included only in more expensive microcontrollers or available as separate integrated circuits. For example, many microcontrollers based on ARM7 or Cortex-M3 cores include 10/100 megabit Ethernet interface [1]. Even more interestingly, the number of additional components required to enable such interfaces is getting smaller. Consider the Ethernet controller, present in many modern microcontrollers. This controller usually requires an additional integrated circuit, the physical transceiver (PHY). However, recent advancements in interface integration eliminated the requirement for the separate PHY integrated circuit. The LM3S6618 microcontroller produced by Luminary Micro is a good illustration [2]. This microcontroller, based on ARM Cortex-M3 core, has an Ethernet controller with integrated PHY; at the time of writing the cost of such microcontroller is about 5$. Using such microcontroller it is both simple and cheap to build a device with Ethernet or even Internet capabilities.

Embedded systems often constitute the components of sensor networks — they act as sensor nodes, each having some set of sensors and actuators [3]. The

base stations (more powerful nodes) collect the aggregated information from sensor nodes and may act as gateways (e.g. to the Internet) [4]. Such or similar applications wishing to communicate over network must agree to the protocol that specifies how the communication is carried out. The application protocol decides how the application interprets the data received from another host on the network, or more generally, from another device. Some well-known protocols were designed solely to fulfil one specific purpose; a good illustration is POP3 (Post Office Protocol 3), the protocol used to retrieve e-mail from the remote server. Other application protocols, such as SOAP (Simple Object Access Protocol) are more general in the sense that they facilitate exchange of structured information, while not imposing any specific usage scenario. That is to say, we can image SOAP being used in a business system for retrieving some financial data, but also in other scenarios where it might be used to receive aggregated data from a sensor network or even to control a group of robots. Yet another class of applications utilize custom protocols; such approach gives more control over what exactly is sent over the wire (which may be important if, for example, performance is critical), but at the same time necessitates writing the protocol handling code. In such situation it's the developer who is responsible for deciding how the messages are encoded and decoded and must write the appropriate code that performs those operations.

Related to the application protocols is the RPC (Remote Procedure Call). It is a popular communication paradigm of client-server communication. Simply speaking, this mechanism is similar to calling local procedure, but when remote procedure is invoked, the input parameters are passed over the network to the remote execution environment, where the execution takes place [5]. The results of the execution are passed back to the caller. All of this happens (at least theoretically) as if the execution took place on a single machine.

There are numerous advantages coming from using RPC for communication:

— simple semantics (calling a remote procedure is similar to calling a local procedure)
— clearly defined interface exposed by the server (usually a list of procedures, with a description of input arguments and output values provided for each procedure)
— no need to write custom code that serializes (changes into stream of bytes) input arguments and deserializes (reads from the stream of bytes) output values since that is done automatically by the library

— lack of interoperability problems present when custom protocols are employed, for instance the endianness problem (order of the data bytes inside a larger data unit in memory)
— ease with which new types of messages may be added (by adding a new remote procedure)

Not surprisingly, using RPC comes at a cost; it introduces unavoidable overhead related to the serialization and deserialization of input parameters and output values. Such delay may be intolerable if low response time is required. Moreover, this communication mechanism is not appropriate for all scenarios, broadcast or multicast transmissions being good examples of where RPC should not be used. From the performance perspective it is also tempting to ignore the fact that client-server communication is not instantaneous; however, the call execution time much longer in comparison to locally called procedures. Apart from that, calling a remote procedure introduces a number of other problems that do not exist when a local procedure is called. For example, the execution of a local procedure succeeds most of the time, while execution of a remote procedure may fail in presence of network errors or remote machine crashes. Such exceptional conditions should be handled by the client program. Another problem of some RPC implementations is that the communication is, by design, synchronous, in the sense that the caller blocks until the calee replies with the response; for some situations it is undesirable, e.g. when the task is expected to take a lot of time. Asynchronous RPC systems solve this problem; in such systems, the caller does not wait until the method completes. Instead, the server sends the output value to the client as soon as the work performed by the procedure is done. Upon receiving the return value, a callback function executes on the client side, signalling the completion of the procedure. Still, the synchronous approach may be advantageous or even required in many cases, particularly for the system described here.

1.2. Motivation

A myriad of different RPC implementations for various hardware and software platforms exist today. Ease of use, expressiveness and interoperability are among the advantages that contribute to the popularity of this communication technology. Yet, most widely used frameworks are not intended to be run on constrained embedded devices, mostly due to the fact that they either require a specific operating system (or, more generally, some runtime platform), or are too large to fit in a small ROM an embedded system. Besides, many of the available RPC frameworks do not concern themselves with low latency, as the most common applications

of such frameworks rarely require response time of the order of, say, hundred microseconds. Existing solutions offering RPC for embedded systems are commercially available but expensive [6].

Considering the problems enumerated above, it was deemed reasonable to create KIPER — a low-latency RPC system based on open-source components, supporting constrained embedded systems. The name KIPER comes from the acronym of the early implementation, named, in Polish, KIerownik urzadzen Po EtheRnecie (Device manager over the Ethernet). The main assumption was that the system should facilitate calling remote procedures with PC acting as RPC client and a constrained embedded system acting as RPC server, while maintaining short response time (measured as the time difference between calling the remote procedure and obtaining the result). Specifically, KIPER was meant to replace an existing RS-232 link connecting the force/torque sensor controller (being a part of an existing robotic control system) to a PC (robot controller). Originally, it was assumed that the RS-232 link would be replaced by the Ethernet connection. Eventually it has been decided that in addition to that, that, KIPER will be used, with underlying communication over raw Ethernet or UDP. In the new setup (utilizing KIPER), PC (the client) calls the remote procedure on the force/torque controller (the server) to obtain force/torque measurements. The PC acts as a part of real-time robot controller (together with another networked PC), and for this reason the time between requesting a measurement from the force/torque sensor controller and delivering it, should be minimized. The force/torque sensor controller is based on LPC-P2378-SK development board including LPC2378 microcontroller (with ARM7TDMI core, 512KB flash ROM and 32KB RAM). This places constrains on the firmware that may be run on the board, exercising KIPER's ability to be run on constrained embedded devices.

1.3. Aim of the thesis

The primary goal of this thesis was to design, implement and assess an RPC system for exchanging information between general purpose computer (e.g. PC machine acting as a robot controller) and a constrained embedded device (e.g. sensor controller). It would also be desirable that KIPER shall be built to be independent of the underlying transport mechanism. The transport mechanism in this thesis is understood as any mechanism for carrying messages containing KIPER protocol between client and server; the terms such as „transport layer" are deliberately

avoided here to prevent confusion, since different transport mechanisms may operate on different layers, e.g. both Ethernet and UDP may be used as transport mechanisms, despite that the latter operates two layers above the former.

An important requirement was that the proposed solution should minimize the latency, i.e. the time between calling the remote procedure and obtaining the output value should be as small as possible. This is due to the real-time nature of the existing control system, that needs to retrieve the force/torque measurements within given time constraints. This is not to say that any attempt of formal proof will be made, rather a series of tests should decide the fitness of the system for such purpose.

It should be noted that often the term „embedded device" is stretched to the powerful devices (effectively computers) that can run „full featured" operating systems, as opposed to embedded devices, running dedicated operating systems such as TinyOS. As an illustration, consider the SAM9-L9261 development board (`http://olimex.com/dev/sam9-L9261.html`). This board is based on AT91SAM9261 microcontroller and runs Linux operating system. Many would argue, that this board may be considered an embedded system. Such system allows to take advantage of RPC libraries intended to run on PC's (CORBA or XML-based RPC libraries being the examples). However, this thesis is especially concerned with constrained embedded systems, i.e. based on microcontrollers with limited RAM and ROM, without operating system or with very simple operating system. One reason for this is that not always is the use of more complex embedded systems justified; simple microcontroller based devices, utilizing only built-in RAM and ROM, often suffice as networked sensor controllers, at a lower cost. Even more, lack of an operating system may be beneficial; without the overhead caused by e.g. scheduling, the response time to some events (e.g. an arrival of network packet) might be improved, in comparison with an embedded device with „full-fledged" operating system such as Linux.

1.4. Scope

The general outline of the most important steps completed while preparing this thesis, includes the following:

— analyzing existing solutions and choosing suitable RPC framework; Protocol Buffers and protobuf-c were chosen

— designing KIPER protocol that describes how the RPC client and RPC server communicate (that is, the protocol that encapsulates the Protocol Buffers messages while adding additional, required data)

— developing the PC part of the RPC system (KIPER-CPP) based on Protocol Buffers
— developing the Embedded device part of the RPC system (KIPER-C) based on protobuf-c
— writing firmware for the force/torque sensor controller, that enables to retrieve readings from the force/torque sensor via RPC;
— writing a variation of the said firmware, utilizing a simple, Ethernet-based protocol to provide the force/torque readings (this was required to compare with solution based on KIPER later in the testing stage)
— performing basic functional tests of KIPER;
— performing tests with force/torque sensor controller, but without MRROC++
— Integrating KIPER-CPP with MRROC++ (the robot control framework), so that MRROC++ could communicate with force/torque sensor controller using KIPER-CPP;
— Conducting the performance testing of the solution using various configurations of MRROC++

1.5. Conventions

Throughout the thesis there exist references to various files (e.g. source code or configuration files). The location of the files is specified by appending a relative path to a path prefix, such as `$KIPER_SRC_DIR`. The meaning of path prefixes is as follows:

$KIPER_SRC_DIR the directory containing the source code of KIPER (both KIPER-C and KIPER-CPP)

$FIRMWARE_SRC_DIR the directory containing the source code of the force/-torque sensor controller

$MRROCPP_SRC_DIR the directory containing the source code of MRROC++ framework

$KIPER_CPP_INSTALL_DIR the directory where KIPER-CPP has been installed

1.6. Outline

The rest of this thesis is organized as follows: Chapter 2 presents the overview of the RPC systems. Chapter 3 states the requirements for the project. Chapter 4 presents the KIPER protocol and high-level design of the system. Chapter 5 presents the implementation details of KIPER. Chapter 6 delivers a manual of KIPER's usage. Chapter 7 describes the test procedure and analysis of results.

Chapter 8 concludes. Additionaly, appendix A presents a glossary of terms and abbreviations commonly used in the thesis.

2. Background

2.1. Remote procedure call

Remote procedure call (RPC) is a construct that enables a client to request a service from a server that may be located on a different processor[1] [7]. It is a popular mechanism for structuring distributed systems, based on request/reply message exchange and (at least in its original form) having the semantics of a local procedure call [8]. The main virtue of RPC as a communication paradigm is that from the point of view of the client, calling a remote procedure is similar to calling a local procedure.

Most RPC systems provide a language that enables the definition of the remote procedures (e.g. for each procedure usually the name of that procedure, the input arguments and the output value). As an illustration, consider a fragment of source file written in Sun RPC language [9]:

```
1  program MESSAGEPROG {
       version PRINTMESSAGEVERS {
          int PRINTMESSAGE(string) = 1;
     } = 1;
5  } = 0x20000001;
```

Not going into unnecessary details, this fragment defines a remote procedure named `PRINTMESSAGE`, having one input argument of type `string` (array of characters) and an output value of type `int` (integer). Such source files are then translated using a stub compiler (e.g. `rpcgen` in the case of Sun RPC) to a set of source files (client and server stubs) that are used by the client to call a remote procedure, and by the server to provide the actual implementation of the remote procedure. Finally, the stubs are compiled (using the compiler for the language they were generated in, e.g. a C programming language compiler in the case of stubs generated by `rpcgen`) and linked with client and server programs. When the client wants to call a remote procedure, it calls a procedure provided by the client stub as it would call any other local procedure, and the RPC system takes care of all the steps required

[1] This is the most common, albeit not the only way to use RPC.

to execute the procedure on a remote machine. Figure 2.1 presents the overview

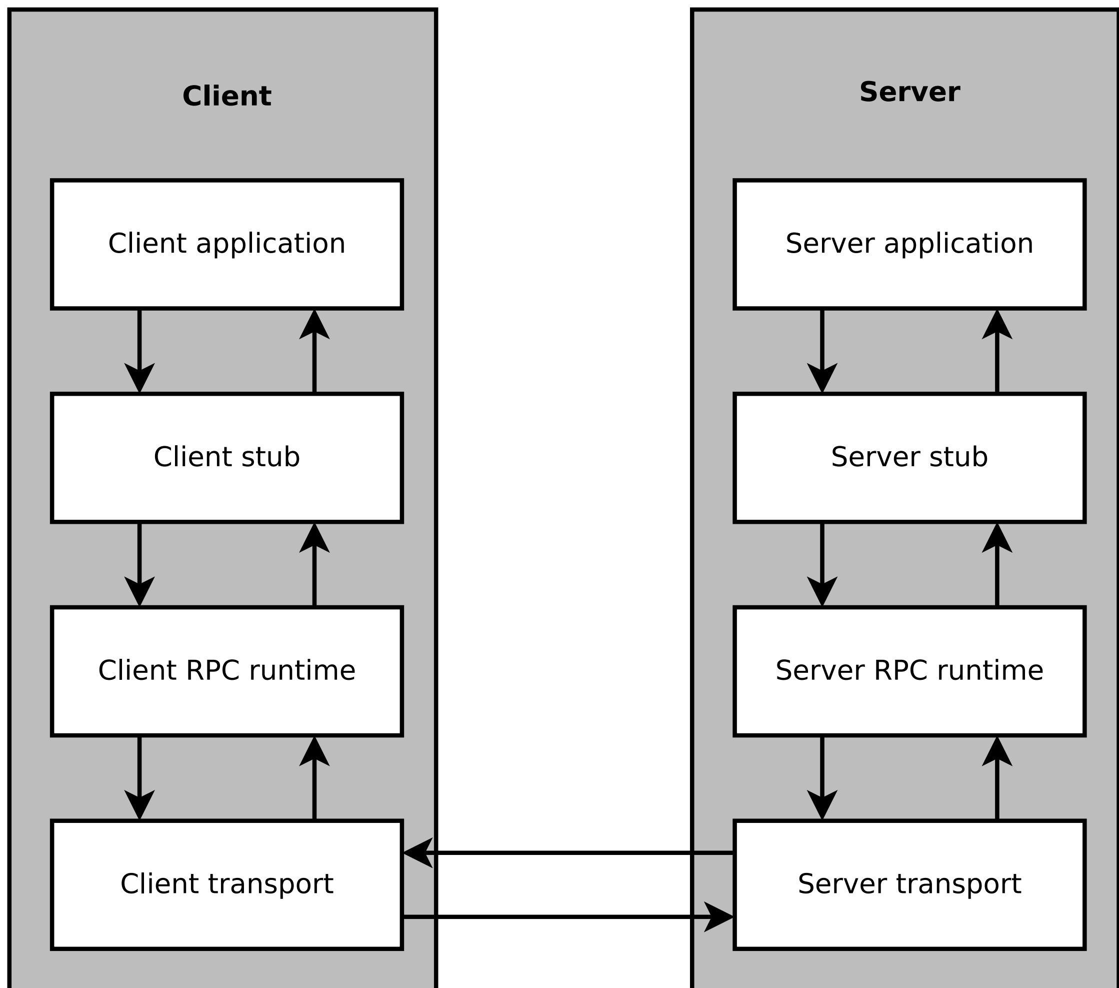

Figure 2.1. Remote procedure call flow

of how the remote procedure call is performed, taking Microsoft RPC model as en example [10]. (Other classic RPC implementations, such as IBM RPC [11] or Sun RPC [12] work in a similar fashion.) The RPC call takes place between a client process and a server process, having different address spaces and usually residing on different machines. The steps involved in calling a remote procedure are the following:

1. The client application issues a normal procedure call to a client stub; (This blocks the thread from which the call has been issued)
2. The client stub translates the input arguments given to the procedure to a standard format for transmission over network and calls the functions of the client RPC runtime to send the request to the server using given transport mechanism (e.g. over TCP/IP);

3. The server RPC runtime receives the request and calls the appropriate server stub procedure;
4. The server stub decodes the input arguments and calls the actual procedure on the server;
5. The remote procedure returns the output value to the server stub;
6. The server stub translates the output value to a standard format for transmission over network and calls the functions of server RPC runtime to send the response to the client using given transport;
7. The client RPC runtime receives the response, and passes it to the client stub;
8. The client stub decodes the response and passes it to the client application;
9. The client application resumes normal operation, as if the procedure has been called on a local computer;

Historically, the RPC concept dates back as at least 1976, when an RPC protocol was proposed as RFC 707 [13]. The highly cited paper by Birrell and Nelson [5] described in detail a working RPC implementation. The article laid the groundwork for future implementations and established much of the terminology related to RPC as it's used today. It also introduced a concept of service registry; previously the client had to know the exact location of the service (e.g. the IP address and the port number in the case of communication over TCP/IP or UDP) in order to call a remote procedure. With the registry, the client queries the third party machine to obtain the location of the service — any subsequent calls are made to the obtained location.

A classic, popular implementation of RPC is ONC RPC (also known as Sun RPC), described in RFC 1831 [12]. It serves as a basis of many UNIX services, especially NFS (Network File System) and operates over UDP or TCP. Created in the middle of the 1980s, it was one of the first commercial implementations of RPC.

In the early 1990s the object-oriented RPC systems began to appear [14]. Notable examples include Microsoft DCOM (Distributed Component Object Model), Java RMI (Remote Method Invocation) and CORBA (Common Object Requesting Broker Architecture). The object-oriented RPC systems share many similarities with earlier non-object oriented RPC systems, such as ONC RPC. One notable difference is that instead of accessing the interface composed purely of remote procedures, the object-oriented RPC systems define their interfaces in terms of classes and their methods; the client then obtains the handle to the remote object and calls its methods.

The beginning of the 2000s saw the advent of Web Services, which defined a standardized mechanism to describe, locate and communicate with online applications [15]. Being based on open standards, such as XML and SOAP, Web Services enjoy popularity up till today, thanks to a plethora of implementations for different platforms.

While the RPC paradigm may provide an illusion of transparency in the sense that calling a remote procedure is similar to calling a local procedure, one cannot underestimate the difference between those two situations. Firstly, calling a remote procedure may fail due to, for example, a network failure. Also in some cases (e.g. when retransmission of lost messages is used) it may not be possible to precisely determine if, or how many times, the procedure was executed. On top of that, with every remote procedure call comes a delay, mainly caused by the following factors [16]:

— marshalling (serialization) and unmarshalling (deserialization) of input arguments/output values
— data copying (across user-kernel boundary, across protocol layers, between network interface and kernel)
— packet initialization (initialization of protocol headers, etc.)
— thread scheduling and context switching
— network transmission time

Some problems related to RPC as a general communication paradigm were outlined in [17]. Many of the concerns raised by the author are still valid: for instance, the fact that RPC is not well suited for streaming communication. Another problem worth mentioning is that it is hard to guarantee exactly-once semantics (i.e. to ensure, that in the face of possible network or hardware failures, the remote procedure is executed exactly once). A more recent article [18] cites impedance mismatch between objects (e.g. in Java programming language) and their counterparts in XML as the main problem with current, XML-based RPC implementations (e.g. JAX-RPC).

As for contemporary RPC implementations, a multitude of free and commercial packages are available. Many business systems employ XML based RPC protocols (e.g. XML-RPC [19] or SOAP [20] being a basic building block of Web Services). These kind of systems are a popular tool for business systems integration. One of the reasons for using XML as the data exchange format for the remote call is that it is explicit and human-readable. However, this is at the expense of performance: parsing XML messages takes longer than parsing binary messages, hence XML based protocols are largely unusable when quick response times are called for.

Recently, two interesting RPC implementations were open-sourced. One is Thrift [21], being a basis for many applications at Facebook, a social networking site. Another is Protocol Buffers used by Google for storing and interchanging all kinds of communication. Both implementations offer an interesting alternative to XML, promising better performance, scalability and language independence. Additionally, they also elegantly solve the problem of versioning — dealing with addition or removal of some fields from the message types being exchanged (e.g. a situation when a client expects an older version of the protocol and the server already uses a new one).

2.1.1. RPC in embedded systems

The RPC communication paradigm is traditionally associated with general-purpose computers. Even so, there is nothing that prevents us from using RPC for communication with embedded systems. Such systems may be defined as „computing systems with tightly coupled software and hardware integration, that are designed to perform a dedicated function" [22]. The omnipresent embedded systems exist in consumer electronics (e.g. MP3 players, mobile phones, kitchen appliances), dependable devices (e.g. medical or automotive systems) and industrial control systems.

There is a thin line between what constitutes en embedded system, and what does not [23]. Nowadays it is common to encounter an x86-based computer running Linux or Windows operating system and performing some dedicated function. In this thesis, the focus was placed on more constrained devices: the tests were performed using a device built around ARM7-based microcontroller.

The idea of putting an RPC system on a constrained embedded device may sound unpractical. In fact, many RPC packages are are unfit for that purpose, due to their size, dependencies on other libraries or specific operating systems as well as processing power requirements which embedded systems do not satisfy. This limits the number of RPC packages to choose from, however does not make the task unimaginable, as proved by successful attempts to bring the benefits of RPC to the world of embedded systems. Authors of [24] examined the applicability of XML-RPC and SOAP as a communication technology for embedded systems and implemented communication with a remote instrument panel using XML-RPC. However, the evaluation kit used in the experiments had 1 MB RAM (in comparison to 32KB general purpose RAM of the force/torque sensor controller used in this thesis) therefore enabling the use of a more complex, larger library. [25] presents Marionette — a toolsuite that allows calling the functions and accessing

the variables of a program running on a wireless embedded device at run-time. The principal component of that toolsuite is Embedded Remote Procedure Call, an implementation of RPC for embedded systems. It has a couple of interesting features, most notably:

— there is no interface definition language — the user marks the appropriate functions with an `@rpc()` tag in the source code, and the server stub is automatically created at compile-time.

— the client code of Marionette consumes only 153 bytes of RAM and 4KB ROM.

2.1.2. Remarks

The above section discussed different topics related to RPC systems as described in the literature. Please note that not all of the terminology applies to all of the approaches to RPC. For instance the Protocol Buffers library does not require any special operating system processes to be run in order for the RPC to work. Even more, in the case of protobuf-c there is no need whatsoever for any process to be present; when executing as a part of a firmware for an embedded system we can hardly speak of a process, since there is no operating system. A term „remote machine" would also be difficult to apply here. Some purists would not consider such system to be RPC in its original meaning, but it's not the author's intention to dispute such problems.

2.2. Communication technologies

Many RPC packages provide default implementations for the transport mechanism. For example, ONC RPC by default may communicate over TCP or UDP. Conversely, the Protocol Buffers library leaves the implementation of the transport mechanism to the user. The choice of the transport mechanism may have implications on the performance of the RPC system (in terms of latency and throughput) as well as semantics (e.g. choosing TCP over UDP will guarantee packet delivery). This section presents an overview of selected communication standards that would be considered for the implementation of the transport mechanism for the designed RPC system. Note, that some of the presented protocols (e.g. RS-232, USB) are not traditionally associated with RPC. In spite of this, there is nothing that prevents us from implementing an RPC transport based on e.g. RS-232, hence the inclusion of the said interfaces in this section.

2.2.1. RS-232

RS-232 is one of the most common computer interfaces. Developed in the late 1960s, even as of today it is built into most PCs and microcontrollers. Notwithstanding the fact that many other interfaces emerged since the inception of RS-232, it still remains popular due to its ubiquity, simplicity and little cost [26].

The shortcomings of RS-232 are the following:

— Only two devices may be linked simultaneously;
— Maximum data rate is 20000 bits per second (although many devices support speeds of 115200 bits per second);
— Maximum cable length is about 30 meters;

Despite these drawbacks it is always worth to consider RS-232 as a viable alternative, due to its low cost and simplicity of implementation.

2.2.2. RS-485

RS-845 is a high-speed, long-distance serial interface, often used to reliably communicate with control systems [27]. This standard is described in [28].

In comparison with RS-232, RS-485 interface has several advantages [26]:

— fast transfer rate (up to 10Mbit/sec)
— ability to connect multiple drivers and receivers (effectively establishing a network)
— good noise immunity
— maximum line length of over 1 km

Those features make the RS-485 an interface especially well suited for communication in industrial environments. On the microcontroller side, RS-485 requires a dedicated integrated circuit, however, in contrast to RS-232, RS-485 is not a standard PC interface. Therefore a special (often costly) extension card is needed to use this interface on the PC side.

2.2.3. CAN

CAN (Controller Area Network) is a reliable digital network originally designed for automotive applications. The basic features of CAN are [29]:

— high-speed serial interface (up to 1Mbit/s)
— low-cost physical medium (operates over simple twisted wire pair)
— short data lengths when compared to other systems
— fast reaction times
— multi-master and peer-to-peer communications

— robust error detection and correction

Devices in CAN network share a single bus. Each node monitors the state of the bus and begins transmission when the bus becomes inactive. The first thing the node transmits after the start of frame (which is used to synchronize nodes on bus after being idle) is the message identifier. By examining the identifier, other nodes decide if they are interested in the message. In CAN, the logical 0 is the „dominant bit" and the logical 1 is the „recessive bit". It means, that if one node transmits the dominant bit (logical 0) and another node transmits the recessive bit (logical 1), the state of the bus will be logical 0. Each node monitors the state of its own transmissions; if a node transmits the recessive bit (logical 1) and sees that the state of the bus is dominant (logical 0), the collision has occurred and the node backs off. Hence, the node transmitting the message with the lowest identifier wins the arbitration after all other nodes have backed off (this is particularly useful in real-time systems for message prioritization). The bus arbitration is non-destructive – the node winning the arbitration just continues to transmit the message.

CAN achieves its robustness thanks to abundant error-checking procedures [30]. CAN includes five methods of error checking. At the message level:

— CRC field that contains the checksum of the application data
— ACK field that is set by the receiving node to acknowledge that the message has been received without errors
— form check that looks for fields in the message that always need to be set

At the bit level:

— Each bit transmitted by the node is monitored and error is generated if a bit is written into the bus and the opposite bit is read
— The bit stuffing rule states that after five consecutive bits of the same logic level, a bit of opposite level must be inserted. Therefore, if six consecutive bits of a same logic level are detected, it is an error condition.

CAN interface is built into many microcontrollers, but on the PC side it requires an extension card.

2.2.4. USB

USB (Universal Serial Bus) is a popular serial interface most commonly used to connect PC with various peripheral devices including: printers, disk drives, scanners or MP3 players [31]. USB standard defines different speeds and transfer types for different needs. For example, bulk transfers (used in external drives) have the highest throughput but give no latency guarantees. On the other hand, interrupt transfers are used when periodic, polled communication occurs. In USB,

the data is is carried using differential pair, which reduces noise. The advantages of USB include:

— ease of use for the end-user
— high transfer speeds
— ability to power the device directly from the bus (up to 500mA current per device can be drawn)
— ubiquity (USB is included in virtually all new PC's)
— low cost

The downside of USB is the implementation complexity, especially on the device side. The number of concepts the developer of the USB device's firmware has to grasp is huge and may be overwhelming at first. The USB protocol is complicated and much of it, (as opposed to other interfaces, e.g. CAN) needs to be implemented in software. Another problem is that all communication is initiated by the host; in standard USB, the devices cannot talk to each other directly, nor broadcasting messages is possible. Finally, the maximum cable length (without an intermediary hub) is only 5 meters.

2.2.5. Ethernet

Ethernet is a popular LAN (Local Area Network) standard defined by IEE802.3. It is the most common networking standard for short-range applications, with most implementations running at least 100Mbps [32, p. 94]. Ethernet consists of four basic elements [33, p. 24]:

frame : a standardized set of bits used to carry data

media access control protocol : a set of rules to allow multiple devices to communicate over the same Ethernet channel

signaling components : standardized electronic devices that send and receive signals over the channel

physical medium : cables and other hardware used to carry the Ethernet signals between computers attached to the network

The ubiquitous nature of Ethernet makes it an attractive option for establishing a low-latency, high-bandwidth communications link between an embedded device and PC. Most standard PCs now include a built-in network adapter. Moreover, many microcontrollers include Ethernet as a peripheral interface. The hosts may be connected directly using a twisted-pair cable, or via an intermediary network switch. In the former case, the latency is minimized, while in the latter case it is possible to form a network consisting of many devices. One great advantage of

Ethernet is that the cost of hardware used to create a small network is relatively insignificant.

2.2.6. EtherCAT

Since mid 80s, the word "fieldbus" has been functioning as an informal definition of a network connecting field devices, such as sensors, actuators, field controllers, regulators, drive controllers and man-machine interfaces [34]. The fieldbusses developed in the 90s dominate the market, but due to low quantities of components sold (compared with generic IT components) the costs are high [35]. In contrast, the costs of typical components required to create an Ethernet based LAN are quite low. This fact makes the Ethernet an interesting option for creating a fieldbus. Alas, there are some inherent properties of standard Ethernet that make it less than perfect for such an application, e.g.:

— high overhead for devices that frequently exchange small quantities of data,

— lack of real-time capability

EtherCAT (Ethernet for Control Automation Technology) was created to solve these problems while taking advantage of well established technology (Ethernet). In principle, from the point of view of the Ethernet, the EtherCAT bus is seen as a single large Ethernet device. EtherCAT relies on master-slave architecture; the master initiates the communication by sending the so-called telegram to the first slave device. (The telegram is carried in the standard Ethernet frame.) Starting with the first slave device, each node extracts the relevant data from the telegram or inserts the data into the telegram and passes the telegram to the next slave. The last slave sends the fully processed telegram to the master. Conceptually, the slave nodes form an open ring: at the open end, the master sends in the telegrams and receives them at the other end after they have been processed. (The actual physical connection of the nodes may be different, e.g. they may all be connected to the common network switch.) Thanks to the hardware processing, the telegrams are processed "on the fly" by the slave nodes, minimizing the delays and ensuring determinism.

EtherCAT is fully compatible with Ethernet in the sense that all the communication is carried out within a standard Ethernet frame, and using standard Ethernet hardware (the only exception being the slave devices which require dedicated hardware).

2.2.7. Bus arbitration

Many communication mechanisms allow to set up a network consisting of multiple devices. Often the devices within the network are connected using a single physical bus; in such situation the rules for accessing the bus must be specified. These rules may specify the order of transmission, algorithms for dealing with collisions (when two or more devices try to transmit simultaneously) and so on. Depending on the bus arbitration protocol used, we can make different assumptions about such properties as throughput and predictability of the communication:

— **RS-232** is a point-to-point protocol, so there is no problem of bus arbitration.

— in **RS-485**, the most common architecture is master-slave, with one master and multiple slaves. As it is the master who initiates all communication (usually by sequentially polling all slave devices), there is no problem of simultaneous transmission on the bus. It is possible to setup a network in which the nodes can transmit at any time, but it requires writing a collision detection algorithm in software, and is generally not recommended.

— in **CAN**, the devices start transmission on the bus at the same time, but eventually one node (transmitting the message with the lowest identifier) wins the arbitration while other nodes back off.

— **USB** by design is a master-slave protocol, with one master (the host) polling the devices sequentially, therefore there is no possibility of collisions.

— in **Ethernet** originally used a Carrier Sense Multiple Access With Collision Detection (CSMA/CD). In this access method, each node observes the state of the bus and when it becomes idle, the node begins the transmission. If a node detects a transmission by other node, it backs off and tries to retransmit a frame after a backoff period. Modern Ethernet networks, using switches and full-duplex transmission no longer require collision detection.

— **EtherCAT** is based on a token-ring topology, therefore only the holder of the token may transmit on medium.

3. Requirements

3.1. High level requirements

The main goal of this thesis was to create an RPC framework for communication between host (PC) computers and embedded devices. More specifically, the intended utilization (and a benchmark as well) of the framework was to facilitate the communication between the PC comprising the robot controller and the force/-torque sensor controller (based on an embedded board). The framework would replace the existing communication scheme based on RS-232.

As a rule, RPC communication frameworks consist of client and server: the server exposes the remote procedures, while the client calls them. In the case of this thesis, it has been decided that the force/torque sensor controller would run an RPC server, while the PC robot controller would use the RPC client to call the server and obtain force measurements (hence all communication is initiated by the PC).

Speaking in terms of high level requirements, the requirement for the RPC client part of the framework was to allow the robot control software running on the PC to retrieve force/torque measurements from the sensor controller via an RPC call. On the other hand, the RPC server part of the framework sends the force/torque measurements upon client's request.

To facilitate force control, the PC robot controller must obtain the force/torque measurements with a specified frequency. An RPC reply message carrying such measurement is relatively small in size (the payload consists of 3 force components measurements and 3 torque components measurements[1]). Hence the assumption about the designed system was that it would be used for high frequency, low-latency communication with relatively small payloads. The response for the measurement request shell arrive within a specific time, otherwise the result is useless. Therefore a timeout mechanism was required to be implemented — the user specifies the maximum time to wait for the reply; if that time is exceeded, the RPC call returns and the appropriate error flag is set.

[1] Represented as analog/digital converter readings

The most time consuming steps during the RPC call are:

— serialization and deserialization of the request/response messages (containing the arguments/return values of the RPC call respectively), and

— transferring the request packet from the client to the server and from the server to the client.

Therefore, in order to minimize the overall response time, it was of the importance to minimize both the time spent on serialization and deserialization of the messages as well as choose/implement such transport mechanism that minimizes the delay between sending and receiving the packet.

3.2. Platform requirements

In the experimental setup used to test the system, the PC robot controller retrieves the force/torque measurements from the sensor controller. The PC robot controller software consists of the MRROC++ (robot programming framework) running user tasks.

MRROC++ (Multi-Robot Research-Oriented Controller) is a library of modules that can be used to construct any multi-robot system controller [36]. Currently the MRROC++ framework runs under QNX Neutrino operating system [36]. At the time of this writing a Linux version was still a work in progress. Because of that, the PC part of KIPER was written with QNX as target platform, but allowing a painless transition to Linux operating systems if desired.

Taking into account that MRROC++ is written in C++ and C++ is a natural language for developing applications for QNX it was decided that C++ would be a language of choice for the PC part of the designed RPC framework.

The force/torque sensor controller is based on LPC-P2378 development board. The board is equipped with ARM7 microcontroller with 32 KB general purpose RAM and 512KB flash ROM. The firmware is written in C programming language and compiled using IAR Embedded Workbench for ARM. The idea was to augment the existing firmware with the RPC server functionality basing on a designed solution. Therefore, the constraint was that it should be able to compile the RPC client using the said compiler and fit it in the device's ROM. Obviously the runtime requirements of the running firmware should take into account the size of RAM (such variables as stack and heap size must be considered)

A „nice-to-have" but not required feature was to write KIPER in such way as to allow easy porting it to a different hardware or software platform by reimplementing those parts of the library which are platform dependent.

3.3. Other requirements

Apart from the requirements stated above there were two additional assumptions about the designed system. Firstly, as far as the transport mechanism for the RPC is concerned, the underlying physical connection should be implemented using cheap commodity hardware, i.e. no additional expensive extension cards would be required for the PC. Secondly, The designed RPC framework should, if possible, utilize existing free (preferably open source) libraries, both when considering the RPC library on which the framework would be based and also any other libraries employed. The extent to which an existing RPC could be employed would become apparent during the course of the project design.

4. Design

One of the first decisions to be made when planning for KIPER was a decision whether to create an RPC framework from scratch or to take advantage of existing software; in most cases, it is reasonable to use an existing solution and tailor it according to need, provided that it will be able to satisfy given requirements after modifications. It is widely acknowledged that using ready-made components usually results in shortened development times; it is not necessary to spend time on designing, coding, testing and documenting those components.

As far as the choice of the particular RPC framework was concerned, the main factors considered were:

— ability to run the framework both on a PC and on a constrained embedded device (force/torque sensor controller)

— small serlialization/deserialization overhead

— subjective ease of use, in terms of defining the services and using the library itself

For obvious reasons, the commercially available RPC frameworks could not have been used. However, what severely limited the choice was the target embedded device. Firstly, there is no operating system, so any libraries that depend on operating system services (such as Sun RPC) couldn't have been used. Secondly, the available RAM and ROM is limited. Thirdly, the available compiler compiles C code or Embedded C++ code (the latter being a heavily limited version of C++) so in fact what is left is a library that is written in C.

Considering the above, it has been decided to base the designed system on Protocol Buffers (`http://code.google.com/p/protobuf/`). In a nutshell, Protocol Buffers is a binary format for serializing structured data. The user defines custom structures (messages) in Protocol Buffers language. The protocol buffers compiler translates the `.proto` files with message definitions to the source code in multiple programming languages, such as C++/Java/Python. The user then uses the generated files to set the message contents, serialize or deserialize the messages (the generated files use the Protocol Buffers runtime library for serialization and deserialization of the messages). What is most interesting from the point of view of the thesis is the fact that Protocol Buffers includes a skeletal implementation of RPC

system (the user is required to implement the underlying transport mechanism). Another important feature of Protocol Buffers is that it uses an efficient, binary format, yielding much better peformance than in the case of e.g. XML-based RPC frameworks.

The Protocol Buffers library heavily utilizes the C++ standard library, which is impossible to fit in small flash ROM of a constrained embedded system. What is more, many C++ compilers targeted at embedded systems support only a subset of C++, called Embedded C++ [37]. As a matter of fact, it may be argued that the C programming language still prevails as the language of choice for developing embedded system firmware. Fortunately, a version of Protocol Buffers based on C programming language, protobuf-c, is freely available. Hence it has been decided to base the part of the system intended for embedded devices on protobuf-c. This library is in many respects similar to the original Protocol Buffers library, albeit much more compact. The protobuf-c compiler generates files in C programming language. Most importantly, the protobuf-c runtime library (which includes the serialization/deserialization code) is small and fits in the limited ROM of the force/torque sensor controller.

Both Protocol Buffers and protobuf-c libraries serialize the messages to the same binary format. However, those libraries include only skeletal RPC implementation, i.e. they generate method stubs, but there is no code for exchanging the messages between client and server. Essentially, the user is responsible for writing client/server code (which was achieved in this thesis). A part of this task was to create a protocol (independent of transport mechanism) that encapsulates the serialized request/response messages for exchanging information between client and server. Each request/response message is contained in a packet, which contains additional information, e.g. the id of the packet or the index of the method to be called. Normally, the client sends the request packet (which contains the request message) and the server answers with the response packet (which contains the response message). Below is the description of the request and response packets.

Request packet

Field	packet_id	method_index	data_type	data_size	data
Size	1 byte	1 byte	1 byte	2 bytes	variable

packet_id : the sequence number of the packet. The client assigns a consecutive packet_id to each request packet, starting from 0. The response packet sent by the server should contain the same packet_id. In other words, packet_id correlates request with the response. Having packet_id allows to detect late responses

that should be discarded by the client. This happens when the response for a given request arrives, but the request timeout has already expired.

method_index : the index of the method being called in the list of methods of the service. Each RPC method within a service has an index (starting from 0) in the order they were defined, so the first method has index 0, the second method has index 1 etc. The client automatically sets this field to the index of the method being called, so that the server knows which method to run.

data_type : indicates what is contained in the data field. When this field has value of 0, data_size will contain the size of the serialized request message, and data will contain the serialized request message. Field value of 1 means that the request message is a special empty message that will not be serialized (see 5). In that case the packet ends here; there are no data_size and data fields.

data_size : the size, in bytes, of the serialized request message, if present. The value is in network byte order.

data : the serialized request message, if present.

Response packet

Field	packet_id	data_type	data_size/error_code	data
Size	1 byte	1 byte	2 bytes	variable

packet_id : the sequence number of the packet. The value is copied from the incoming request packet.

data_type : indicates what is contained in the data field. When this field has value of 0, data_size will contain the size of the serialized response message, and data will contain the serialized response message. Value 1 is reserved. Value 2 means that the server returned an error; in that case error_code field is set instead of data_size (see chapter 5).

data_size : the size, in bytes, of the serialized response message (in network byte order) or error code if error was returned instead of regular response.

data : the serialized response message. This field is not present when error is returned.

4.1. Packet encoding/decoding

To call the RPC, the client creates a request packet. The client fills in the `packet_id` which is a sequence number. For each RPC service there is a number of RPC methods defined in the `.proto` file. As this file is used to generate stubs for client and server, both of them hold an array of the methods in the order they were defined in the `.proto` file. The client fills the `method_index` with the index of the

method being called. But how are the arguments of the method call serialized? Let's assume this is a regular RPC (`data_type = 0`, without `VoidMsg`). The arguments of the call are contained in the „request message", as defined in the `.proto` file. The client uses Protocol Buffers library to serialize this message automatically, to the Protocol Buffers binary format (description of which is located at `http://code.google.com/apis/protocolbuffers/docs/encoding.html`). For instance, if the request message is defined as

```
message Test2 {
  required string b = 2;
}
```

and, for the given call, the string `b` equal to „testing", after serialization the resulting stream of bytes is `12 07 74 65 73 74 69 6e 67` in hex. The first two bytes (`12 07`) denote the type (string) and the length (7) while the remaining bytes are the UTF-8 encoded string „testing". For other data types this is more complicated, e.g. the integer numbers may be encoded using variable length. The length of the serialized message is put into `data_size` while the resulting stream of bytes is put into `data` field of the packet; the packet is than sent to the server.

On the server side, upon receiving the request packet, the server reads the `method_index` field and locates the method to be called in the method array. Having the method definition, the server knows what is the expected request message type (the arguments of the call) for that method and uses Protocol Buffers to unpack the stream of bytes into a structure which is passed to the method handling code.

Similar steps are performed when the server encodes the reply packet and when the client receives this response packet.

4.2. Message flow

This section describes the message flow between RPC client and RPC server. This simple request-response protocol is independent from the underlying transport mechanism that is responsible for delivering the messages.

The RPC client always initiates the message exchange by sending the request packet to the server (the format of the request packet is described in the previous section). The request packet contains most importantly the index of the remote method to call as well as the serialized request message (carrying the arguments of the remote method). For each RPC call, the client sets up a timer; if the timer expires before response arrives, the call results in a timeout (the Protocol Buffers API

includes the `RpcController` class which may be used to check whether timeout has occurred.)

The RPC server will parse the received request packet and perform basic error checking: it will verify whether the method with the given index exists and if the type of the message the request carries is the same as the method's request message, as defined in method signature. If so, the server will try to parse the message. If parsing is unsuccessful (which should not happen and indicates a programmer error), the server will not send anything back to the client; instead an appropriate error message should be logged on the server side. One could argue that the server should send the error back to the client in such an event, but it has been decided that it would unnecessarily complicate the protocol (there is already a possibility to send „error" response, but it is reserved for logical errors).

Upon successful parsing the request message, the server will run the actual implementation of the method. When it completes, the server will either:

— send the regular response message; the implementation of the method prepares the response which is serialized, wrapped in the response packet and sent back to the client.

— send the „error" response message; the implementation of the method sets the error code, which is sent back to the client in a response packet. No response message is prepared and serialized.

— send no reply; this happens when the response message in the signature of the method is `VoidMsg` (a special message type defined in KIPER). In such a case the client does not expect the server to send the reply message at all.

When the client receives a response from the server, it first checks if the request of the same packet id as the response that arrived, is waiting. If so, the response message is parsed (again, any parse errors at this point are logged only) and the result passed to the caller. The caller receives the output value and the program proceeds further.

4.3. Choosing the transport mechanism

The Protocol Buffers (as well as protobuf-c) leave the implementation of the transport mechanism to the user. In the case of Protocol Buffers, the user has to write a concrete implementation of the `RpcChannel` interface, which represents a communication line to a service, as well as (on the server side) write an RPC server that will receive the packets and call the real methods of the service. In the case of protobuf-c, the user has to write an „implementation" of ProtobufCService pseudo-class (see 5.3.1), and an RPC server.

The choice of the communication medium was dictated by the availability of the interfaces on the force/torque sensor controller board, the need to provide low-latency communication, and simplicity. Among the interfaces available, the most interesting from the point of view of the project were: Ethernet, USB, RS-232 and CAN. Weighing different pros and cons, Ethernet was chosen as communication medium. The choice was motivated by simplicity of setup, low latency and speed. In comparison, RS-232 is relatively slow, CAN requires an additional PC card while USB is difficult to implement on the embedded device side.

With Ethernet chosen as communication medium, there were a few options as to what protocol to use. The first obvious choice was „raw Ethernet", where the application data is sent in Ethernet frame's payload field, without any additional protocol (e.g. IP). On the PC side, this requires „raw sockets" api which differs between operating systems. On the force/torque sensor controller side the implementation is simple as it involves reading and writing to the Ethernet controller buffer using predefined functions.

For comparison, it was decided to implement a transport mechanism using UDP atop Ethernet. This variant allows for simplification and unification of the implementation on the PC side. On the force/torque sensor controller board side it requires an additional library (uIP - TCP/IP implementation for embedded systems). On the PC, Boost ASIO library was chosen to enable communication over UDP (this portable library also runs on other platforms such as Linux or Windows). From the application point of view, sending UDP datagrams is not much different from sending raw Ethernet packets. Instead of MAC address, the destination host is identified by IP address and port. As with raw Ethernet, there is no guarantee of delivery. UDP introduces additional overhead (the impact of which shall be examined) but on the other hand it allows routing packets between different networks, which is impossible with raw Ethernet.

4.4. Simple protocol for obtaining force/torque measurements

For comparison purposes, a simple, raw Ethernet-based protocol for obtaining force/torque measurements from the force/torque sensor controller, was devised. Basing on this protocol a variation of the software was prepared. The required changes had to made both in force/torque sensor controller firmware and the PC part of the software. In the testing phase, the two variations were compared: one utilizing KIPER and another utilizing the simple protocol described in this section. This way it was possible to measure the overhead imposed by introducing the RPC libary.

The protocol works in the following way: The client sends the request as an Ethernet frame to the server (knowing server's MAC address). The request contains the number of the command to be executed (in our case there is only one command — „retrieve force/torque readings"'), the command data length, and command data (if any). The server sends the response containing response data length and response data (in our case, force-torque measurements) to the MAC address of the client, taken from the request Ethernet frame. The structure of the request and response packets is given below.

Request packet

Field	command_number	data_size	data
Size	1 byte	2 bytes	variable

command_number : the number of command to execute
data_size : the size of the command's arguments, may be zero
data : command data (if any)

Response packet

Field	data_size	data
Size	2 bytes	variable

data_size : The size of the reply data
data : The reply data

5. Implementation

5.1. Overview

As stated before, the requirements of this thesis dictated, that the following software modules should be created:

— KIPER-CPP: the an implementation of Protocol Buffers' RPC system, written in C++. Intended to be run on PC computers.

— KIPER-C: the implementation of protobuf-c RPC system, written in C. Intended to be run on embedded devices.

— Firmware for LPC-P2378-SK development board. Essentially, the firmware is responsible for getting the force/torque measurements from Ati6284 sensor and sending the measurements in response to the RPC requests. The firmware uses KIPER-C for communication, adding an implementation of transport mechanism using raw Ethernet and UDP. Note that a variation of the firmware, using simple protocol for communicating the force/torque measurements, has also been created.

5.2. KIPER-CPP

Essentially, KIPER-CPP is an implementation of Protocol Buffers' RPC system. Most importantly, it provides implementations of `RpcChannel` class (which is responsible for performing an actual RPC call). Available transport mechanisms are raw Ethernet and UDP.

5.2.1. Employed technologies and standards

Protocol Buffers

Protocol Buffers is described by its authors as „a flexible, efficient, automated mechanism for serializing structured data [. . .]" [38]. The structured data (called messages) are defined in `.proto` files using Protocol Buffers language. The Protocol Buffers compiler generates the C++/Java/Python code from the `.proto` files. The generated code allows to quickly serialize and deserialize the messages using an

efficient, well-described binary format. The messages defined in `.proto` files may be used in RPC systems as well as for permanent storage of arbitrary structured data.

The messages are defined in `.proto` files using Protocol Buffers language. As an illustration, imagine a system that tracks a number of mobile robots. A (very simplified) description of a robot in such system, written in a `.proto` file might look the following:

```
1
  message Robot {
      required uint32 robotId;
      optional float x;
5     optional float y;
      boolean gpsEnabled;
  };
```

This file defines a message named `Robot` and having four fields: `robotId` (32-bit unsigned integer field, identifier of a robot), `x` and `y` (floating point field, position of the robot) and `gpsEnabled` (boolean field, indicates whether robot has GPS unit enabled). Compiling this definition with Protocol Puffers compiler (`protoc`) results in a header and source file. The generated files contain the definition of `Robot` class as well as various methods for setting and retrieving value of fields as well as for serialization (e.g. to array or stream). The methods of the class use Protocol Buffers runtime library to perform serialization and deserialization. In our example, the serialized class could be used to exchange information about robots with another systems or perhaps, in a simulator, to save the simulation state. Another system requiring the information about a robot could possibly use an RPC to obtain such information. The definition of a method implementing this functionality would be the following:

```
message GetRobotInfoRequest {
    uint32 robotId;
};
service RobotService {
    rpc GetRobotInfo(GetRobotInfoRequest) returns (Robot);
};
```

This defines the `GetRobotInfo` method which takes `GetRobotInfoRequest` message (containing `robotId` — the id of the robot to be retrieved) as an argument and returns `Robot` of the given id.

Protocol buffers supports scalar fields (e.g. strings, integers, boolean values), as well as composite fields (enumerations and previously defined message types). Each field has to be marked as optional, required or repeated. Optional fields, as opposed to required fields, need not to be set before message is serialized. In addition, a required field, once introduced in the message definition, cannot be later removed if backward compatibility of the protocol needs to be maintained. Repeated field is basically a list of fields of the same type, repeated zero or more times (similar to an array).

The advantages of using Protocol Buffers, as advocated by the authors, include:

— speed and compactness
— simplicity
— generation of classes that are easy to use programatically

Protocol Buffers provides a skeletal RPC framework that generates interfaces and stubs from `.proto` files. Consider the following service definition:

```
service ExampleService {
    rpc exampleMethod (RequestMessage) returns (ResponseMessage);
}
```

This defines, within `ExampleService`, an RPC method named `exampleMethod` that takes a `RequestMessage` as an argument and returns `ResponseMessage`. For each service defined in `.proto` file, the Protocol Buffers compiler generates an interface and a stub. The stub implements the service interface and allows the client to call the RPC methods. Internally, the stub forwards the method call to the `RpcChannel`. There is no implementation of `RpcChannel` shipped with Protocol Buffers; typically the implementation will serialize the request message, pass it to the target of the call (e.g. a remote machine), wait for the reply, deserialize it call the closure method or function given at the call. One of the tasks for this thesis was to prepare two implementations of `RpcChannel`: one utilizing raw Ethernet, and another utilizing UDP.

The RPC framework provided by Protocol Buffers was developed with asynchronous communication in mind. Specifically, while calling a remote method, one has to provide a function object (a „closure” in Google's nomenclature) that will be called when the remote method completes. Inside the function or method enclosed by the function object, the caller should check for errors and if the call was successful, process the result. The implicit assumption here is that calling a remote method does not block the calling process; the method returns immediately. Only when the response arrives, or e.g when timeout occurs, is the closure called.

This model of communication was implemented in the first version of KIPER-CPP. However, it turned out that for the specific application of low-latency communication, the result is needed immediately, so the caller will wait for it anyway. The caller would typically use a condition variable, or any other applicable synchronization mechanism, to wait for the response to arrive. Therefore, it was decided to simplify the communication model by making the RPC calls synchronous: the calling thread is blocked until the response arrives or timeout occurs. The closures are no longer required — when the program passes the method call, either the method completed successfully or an error occurred. Instead of passing the closure, the client passes `NULL` value.

Boost

Boost (`http://www.boost.org/`) is a collection of free, peer-reviewed, portable C++ libraries. Most of the Boost libraries are header-only, i.e. they do not require linking with separate library binaries; to use the library one simply has to include the required header files. Those libraries which must be build separately compile on most Unix-like systems (including Linux and QNX) and Windows systems.

KIPER-CPP uses Boost ASIO for UDP transport. This allows to use the simplified, object-oriented, platform-independent interface for socket communication; the implemented transport mechanism may be reused on platforms that support Boost ASIO. KIPER-CPP also depends on other useful Boost libraries, including thread library, lexical cast and smart pointers.

CMake

CMake (`http://www.cmake.org`) is a cross-platform, open-source build system, designed to build, test and package software. Essentially, it generates standard build files from CMake configuration files. On Unix-like systems, by default, CMake generates Makefiles that are used by make. On Windows, CMake can generate, among others, Visual Studio project files and Eclipse CDT (C/C++ Development Tooling) project files.

The main advantage of CMake is that it relieves the programmer from the burden of writing Makefiles by hand, and allows to build software on different platform using the same set of configuration files.

CMake is used as a build system for KIPER-CPP, as well as for test version of KIPER-C. KIPER-CPP also ships with CMake modules for configuring Protocol Buffers and KIPER-CPP libraries (written for the purpose of this thesis). The Protocol Buffers module also allows to automatically compile `.proto` files.

BPF

One of the transport mechanisms that were required to be supported by KIPER was raw Ethernet. In this thesis, using raw Ethernet for communication is understood as sending application data as a payload of an Ethernet frame, without encapsulating it in any higher level protocol, such as IP or UDP.

Since version 6.4.0, the QNX network stack is based on NetBSD 4.0 network stack. The first approach to implement the raw Ethernet transport mechanism, was to use the pcap library, available with the QNX network stack. This library provides a simple interface for reading raw Ethernet frames from the network interface, as well as for sending such frames.

Unfortunately, the QNX version of pcap library appears to have a bug that effectively prevented the possibility of using the library in KIPER-CPP. In pcap, it is possible to setup a filter that discards the packets the user is not interested in. For instance, the following filter expression

```
ether proto 0x8000
```

will discard the frames that have ether type field set to value other than `0x8000` (`32768` decimal), i.e. the frames that do not encapsulate the IP protocol. Such filter has to be enabled to effectively select only the packets that are interesting from the application point of view, in the case of KIPER, the packets carrying messages. (These can be identified by the Ethernet protocol field, which in the case of KIPER is set to `0x8000`.)

Regretfully, enabling such filter under QNX, for reasons unknown, effectively filters all packets arriving at the interface, not only the ones not matched by the filter. However, knowing that under QNX the pcap library is only a wrapper around BSD Packet Filter (BPF), the solution is to use the BPF interface directly.

BPF is a low-level interface for „snooping" on incoming packets as well as for seding custom packets. All the operations supported by BPF are performed via BPF pseudo device, `/dev/bpf*`, e.g. `/dev/bpf0`. The functions to perform these operations are mainly `read()`, `write()` and `ioctl()`.

A huge performance increase is gained by filtering the packets the application is not interested in [39]. By employing the filter at the kernel level as opposed to application level (the application-level filter being realized by examining the contents of the packet after the packet is already passed to the application), the unnecessary copying of packets is avoided. BPF utilizes a simple filter pseudo-machine to implement its filtering model. Using instructions of the BPF pseudo-machine, it is

possible to examine packet's contents and decide if the packet should be passed or discarded (if the former case one also decides which portion of the packet should be retained).

The IEEE 802.3 standard defines a two-octet Type/Length field inside Ethernet frame. If the numerical value of the field is less than or equal 1500 (decimal), the field is used as length of the frame. If the value is greater than 1500, it defines the type of encapsulated protocol [33]. For example, if the Ethernet frame carries an IPv4 (Internet Protocol version 4) packet, the field will be set to `0x8000` (`32768` decimal). KIPER's implementation of raw Ethernet transport sets this field to `0xD903` (`55555` decimal) to mark KIPER packets. The BPF filter is set to capture only the packets with such value of type field. The array describing such filter is the following:

```
filterCommands[] =
    { BPF_STMT(BPF_LD + BPF_H + BPF_ABS, 12),
    BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, 55555, 0, 1),
    BPF_STMT(BPF_RET + BPF_K, -1),
    BPF_STMT(BPF_RET + BPF_K, 0) };
```

The filter is represented as an array created by composing C programming language macros. Each macro represents a single instruction. The first instruction loads the contents of the half-word (2 bytes) located at offset 12 starting from the beginning of the packet — the type field discussed above. The second instruction is a conditional jump: if the value of the accumulator is equal to `55555` (decimal), the jump offset (counting from the current instruction) is `0` (jump to the third instruction), otherwise it is `1` (jump to the fourth instruction). The third instruction returns from the program with value `-1` (pass the whole packet) while the fourth instruction returns from program with value `0` (discard the packet). The BPF manual contains detailed description of BPF pseudo-machine instructions.

5.2.2. KIPER-CPP API

Below is the short description of the main modules included in KIPER-CPP, listed by the name of the module header file.

RpcClient.hpp : Contains `RpcClient`, an abstract base class, subclassing `RpcChannel` from the Protocol Buffers library. `RpcClient` implements the `CallMethod` of the `RpcChannel` interface. The implementation transforms the request message (passed as an argument) into a byte array (a packet) which then is passed to a method that performs the actual delivery of the packet to the destination of the call. The method is implemented by the subclasses of

RpcClient, such as RawEthernetClient, which transmits the packet using raw Ethernet.

AsioUdpClient.hpp : An implementation of RpcClient, using UDP for communication. Based on Boost ASIO library.

RawEthernetClient.hpp : Contains RawEthernetClient, an implementation of RpcClient, using raw Ethernet for communication. Using BPF.

ClientRpcController.hpp : Contains ClientRpcController, an implementation of RpcController (an inteface from the Protocol Buffers library, required to be implemented) KIPER's implementation allows to set call timeout, check for any errors/timeout expiration after call completes and perform reset, in order to reuse the controller in the next call.

KiperException.hpp : Contains KiperException, a general exception class, thrown e.g. if class initialization fails. For instance, the constructor of AsioUdpClient throws this exception when connecting to remote host fails.

PacketUtil.hpp : Contains RequestPacket and ResponsePacket helper classes for creating packets

Log.hpp : Contains utility functions for writing log messages.

5.3. KIPER-C

KIPER-C is intended for deeply embedded devices hence it is written in C programming language. Generally speaking, it provides the implementation of protobuf-c RPC system. It may be treated as a library, but in reality it is not intended as a separate compilation unit to be linked against; rather the files are put into the project tree, customized according to the given platform and compiled as normal source files. (See the firmware of the force/torque sensor controller for an example.)

5.3.1. Employed technologies and standards

Object-oriented programming style in C

The C programming language does not support the object oriented programming paradigm. However, it is possible to emulate some aspects of object orientation by using structures and functions acting on those structures. Such object-oriented programming style is used in protobuf-c and KIPER-C, therefore it deserves an explanation.

As an example, consider an abstract C++ Shape class, with a pure virtual method that draws the shape on the screen, and a concrete Triangle class that derives from Shape class:

```
class Shape {
public:
    virtual void draw(Screen* screen)=0;
};

class Triangle : public Shape {
public:
    void draw(Screen* screen) {
        // implementation of draw()
    }
};
```

To develop these classes in C using object-oriented programming style, first we „forward-declare" the Shape structure (acting as Shape class), and define a new type, DrawFunPtr, a pointer to a draw function that will be implemented by the „subclasses".

```
struct Shape;
typedef void (*DrawFunPtr)(struct Shape*, struct Screen*);
```

Next, we declare the Shape class with DrawFunPtr as a field. The „subclasses" will set this field to the custom function.

```
struct Shape {
    DrawFunPtr draw;
};
```

The Triangle structure acts as a „subclass" of Shape. The first field of Triangle, named base needs to be of Shape type. This ensures that the pointer to the Triangle may be always safely cast to the pointer to Shape. Such cast is possible thanks to the way the C compiler places the fields of the structure in memory.

```
struct Triangle {
    struct Shape base;
    /* vertices */
    Point v1;
    Point v2;
    Point v3;
```

```
}
```

Then we create a `Triangle`-specific implementation of the `draw()` function. Note that since the signature of this function must conform to the previously defined `DrawFunPtr` type, the first argument of this function has to be `struct Shape*`. Therefore it is required to cast it to `struct Triangle*` first.

```
void Triangle_draw(struct Shape* shape, struct Screen* screen) {
    struct Triangle* triangle = (struct Triangle*)shape;
    /* draw the triangle on the screen */
    /* ...*/
}
```

We define the `initialize()` and `destroy()` function, acting as constructor and destructor respectively. The `initialize()` function should, at minimum, set the pointers to the functions this „subclass" overrides, and perform any other initialization steps. The `destroy()` function should perform any cleanup required, e.g. free the memory allocated inside `initialize()` function.

```
void Triangle_init(Triangle* triangle, Point v1, Point v2, Point v3) {
    triangle->base.draw = Triangle_draw;
    triangle->v1 = v1;
    triangle->v2 = v2;
    triangle->v3 = v3;
};

void Triangle_destroy() { }
```

It is now possible to use `Shape` in a polymorphic way:

```
struct Triangle triangle;
Triangle_init(&triangle, 1, 2, 3);
struct Shape* shape = (struct Shape*)triangle;
shape->draw(screen);
```

To simplify handling of the structures, it is often useful to use `typedef` statement. The convention in protobuf-c and KIPER-C is to first define a structure with _ suffix and than use a `typedef` to give this structure a new name, this time without trailing _.

```
struct Shape_ { (...) }
typedef Shape_ Shape;
```

Now it is possible to omit the `struct` keyword when referring to `Shape`, e.g.

```
Shape* shape = (Shape*) triangle;
```

Uip

Uip is „intended to make it possible to communicate using the TCP/IP protocol suite even on small 8-bit micro-controllers" [40]. While the author of Uip stresses its usefulness for very constrained, 8-bit microcontrollers, it is certainly possible to run it on more powerful, 32-bit microcontrollers. Uip focuses on TCP implementation, however communication over UDP is also possible.

Writing network applications using Uip differs from the traditional approach utilizing BSD sockets or Windows sockets (Winsock). Uip based applications run inside an event processing loop; typical events include appearance of a packet on a network interface or expiration of a periodic timer. The user is required to register a callback function that reacts to these events, using `UIP_APPCALL` or `UIP_UDP_APPCALL` for TCP and UDP respectively. Inside the callback function, the application will check for the source of the event and act accordingly (e.g. parse the incoming packet). All actions originating from the user (e.g. sending a packet) must also be performed inside the application callback function.

There were some modifications that had to be introduced in order to allow low-latency communication using Uip. Firstly, the periodic timer that calls `uip_udp_periodic()` function has been removed. (This function performs periodic servicing of UDP connections. User provided `UIP_UDP_APPCALL` function is called from within that function.) Instead, `uip_udp_periodic()` is always called in each iteration of the event loop (previously it was called only when the timer expired). This way the user's action (e.g. sending a packet) may be performed as soon as possible.

The original implementation of Uip did not support creating a listening UDP connection. Therefore it was necessary to implement a missing function (`uip_udp_listen()`) to allow KIPER-C to function as an RPC server with Uip's UDP implementation as transport mechanism.

5.3.2. Configuring KIPER-C

KIPER-C has many compile-time configuration parameters that need to be set by the users of the library. KIPER-C expects the file `<kiper-c/KiperDevConfig.h>`. The user has to create this file and place it in the directory that is added to the compiler's include path. (Note that this file needs to reside in `kiper-c` subdirectory as per include instruction). The easiest way to do this is by copying and customizing

the file `$KIPER_SRC_DIR/kiper-c/library-src/kiper-c/ConfigExample.h`. For an example on how to set various configuration parameters, refer to configuration header for the LPC-P2378-SK firmware, `$FIRMWARE_SRC_DIR/src/Server/kiper-c/KiperDevConfig.h` (in this case, the directory `$FIRMWARE_SRC_DIR/src/Server` is added to the compiler's include path..)

5.3.3. KIPER-C API

Below is the short description of the main modules created for KIPER-C, listed by the name of the module header file:

ServerTransport.h : contains base pseudo-class for exchanging packets with the client. Two „implementations" of this pseudo-class are present in LPC-P2378-SK firmware (for raw Ethernet and UDP communication).

RpcController.h : contains `RpcController` pseudo-class used by the server to signal any errors to the client.

RpcServer.h : Contains `RpcServer` pseudo-class, which receives packets from transport, decodes them, and passes them to the service implementation.

PacketUtil.h : Contains utilities for decoding/creating packets.

5.4. Integration with MRROC++

MRROC+ (Multi-Robot Research-Oriented Controller) [36] is a programming framework that may be used to create a multi-robot controller. MRROC++ comprises a set of libraries and processes:

— the operator interface, with User Interface Process (UI),
— the task dependent part, with MP (Master Process) and ECP (Effector Control Process), and
— the hardware dependent part with EDP (Effector Driver Proces) and VSP (Virtual Sensor Process).

The user is expected to customize the processes comprising the task dependent part and the hardware dependent part (the operator interface is constant). For each effector, an ECP process and an EDP process is created; EDP process is supervised by the ECP process. ECP is in charge of executing user's task dedicated to the effector, while EDP is responsible for the direct control of the effector. EDP is further divided into threads with different responsibilities. For the purpose of this thesis, the `EDP_FORCE` thread is the most interesting, because the test procedure was involved with retrieving the force/torque measurements, which is done inside

the said thread. This thread is responsible for force measurements and conversion, e.g. removing the effects of gravitation. The decision to make EDP (as opposed to VSP) responsible for force measurements was due to the dominating proprioceptive character of the force/torque sensors.

One of the steps of the thesis was replacing the existing mechanism for retrieving force/torque measurements with a new one, utilizing KIPER library. The assumption of the proposed solution was that the measurements would be retrieved by calling a remote method, and that the communication would take place on a local network, using raw Ethernet or UDP. To facilitate this, the code that originally retrieved the measurements was modified so that the measurements could be retrieved by calling a remote method via KIPER-CPP.

To retrieve the measurements via RPC, a `.proto` file with RPC service description had to be created. The file `$MRROCPP_SRC_DIR/src/edp/ati6284MS/ati6284.proto` defines the RPC service exposed by the force-torque sensor controller and also defines the messages used by the methods defined within the service:

```
package mrrocpp.edp.sensor;

import "Common.proto";

message GenForceReading {
    required sint32 fx = 1;
    required sint32 fy = 2;
    required sint32 fz = 3;
    required sint32 tx = 4;
    required sint32 ty = 5;
    required sint32 tz = 6;
}

service Ati6284 {
    rpc GetGenForceReading (kiper.VoidMsg) returns (GenForceReading);
}
```

The `Ati6284` service defines one method, `GetGenForceReading` which gets force/torque readings (returns `GenForceReading`). The `GenForceReading` message has six fields: `fx`, `fy`, `fz` (force measurements for axes x, y and z respectively) and `tx`, `ty`, `tz` (torque measurements for axes x, y and z respectively). Note that the measures are raw readings from the analog/digital converter; after retrieval from force/torque sensor controller they need to be additionally processed.

The Protocol Buffers compiler creates two classes for the service: `Ati6284` (the service interface, implemented by the server) and `Ati6284_Stub` (the stub used by the client). In MRROC++, the `Ati6284` class is used to retrieve the force/torque measurements. The constructor of this class takes `RpcChannel` as an argument. Depending on the transport mechanism to be used, either an instance of `AsioUdpChannel` or `RawEthernetChannel` is passed. To retrieve the measurements, the `GetGenForceReading` of `Ati6284_Stub` is called.

As mentioned before, in the testing phase the solution using KIPER was compared to the one utilizing a simple protocol. This latter version required changes in MRROC++ that are conceptually similar to those mentioned above, only the communication means is different.

5.5. Force/torque sensor controller hardware

In the robotic setup used in the testing stage of the project, the force/torque measurements are retrieved from the force/torque sensor via force/torque sensor controller, shown on figure 5.1.

Figure 5.1. Force/torque sensor controller

The complete description of the device is given in [50], here, only a brief description will be given. Figure 5.2 presents a simplified view of the force/torque sensor controller. The device is based on a LPC-P2378-SK development board; the

heart of it is the LPC2378 microcontroller (MCU) by NXP. The main features of this microcontroller are:
— 72MHz, 32-bit ARM7TDMI-S processor,
— 512kB on-chip flash
— 64kB SRAM
— serial interfaces support including Ethernet, UART and USB
This microcontroller is cost-effective, but powerful enough to run KIPER-C. The microcontroller is connected to an external Analog/Digital converter, MAX197 via GPIO (General Purpose Input Output) lines. This 12-bit ADC has 8 channels and is capable of converting the signal within ± 10V range, which is the voltage range of the force/torque sensor's output. Six of the eight available channels are used; three for force readings and three for torque readings.

The LPC2378 microcontroller supports the 100Mbit Ethernet with an addition of a PHY transceiver; the PC robot controller uses the Ethernet interface to retrieve the ADC readings (corresponding to the force/torque measurements) from the force/torque sensor controller. Apart from the Ethernet controller, there exists a JTAG interface which is used to program the microcontroller's on-chip flash. The debug messages logged by the firmware may be outputted using RS-232 connector.

5.6. Force/torque sensor controller firmware

One of the tasks of the thesis was to write a firmware for force/torque sensor controller. The firmware is responsible for reading measurements from Ati6284 force/torque sensor (via ADC) and exposing it via RPC service communicating over raw Ethernet or UDP. As basis for the transport mechanism (UDP, raw Ethernet) code, one of the example projects for the LPC-P2378-SK development board — Uip WebServer — was used. The example project is a HTTP web server running over TCP/IP using Uip library. The new firmware uses the code that initializes the Ethernet interface of the LPC-P2378-SK development board as well as allows sending and receiving Ethernet frames. Another part of the Uip Webserver project that was incorporated into the new firmware was the Uip code specifically configured for the LPC-P2378-SK Ethernet interface. The Uip was reconfigured in order to use UDP instead of TCP.

There are two KIPER-based versions of the firmware: first, using UDP (via Uip library) as transport mechanism, second using raw Ethernet as transport mechanism. These two versions exist as two different IAR Embedded Workbench projects within KiperDev workspace, server and server-raw. (IAR Embedded Workbench is a tool suite used to build LPC-P2378-SK firmware).

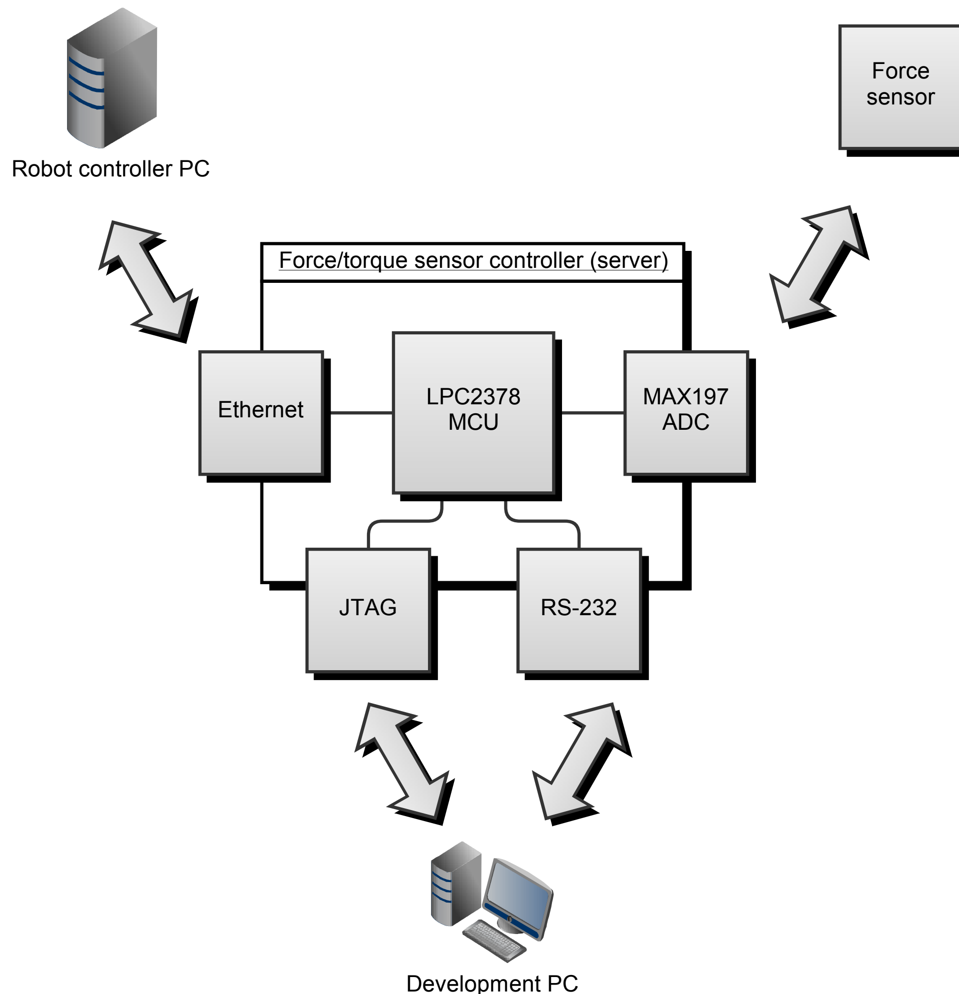

Figure 5.2. High level view of the force/torque sensor controller

The basic outline of the code is the same in both cases; after the initialization of the Ethernet interface and, in case of the UDP version, the Uip library, the software enters an infinite loop:

```
getMeasurements();
```

The function `getMeasurements()` obtains the force/torque measurements from the force/torque sensor. This functions has almost constant time of execution.

While the function `getMeasurements()` executes in an infinite loop, the processing of the incoming packets takes place in the interrupt service routine. The

interrupt is triggered when the packet arrives at the network interface. The interrupt service routine, `processNetworkEvents()` checks if the arriving packet is KIPER's packet and, if so:

— in UDP server version, passes the frame to the Uip, which parses the frame payload, and if it is an UDP packet sent to the port on which KIPER-C is listening, passes is to KIPER-C for further processing (KIPER-C will parse the payload as an RPC request), or

— in raw Ethernet server version, passes the payload of the frame directly to KIPER-C.

— KIPER-C, if required, sends the response message back to the client

For the moment the array containing the ADC measurements is updated, the Ethernet interrupt is disabled, to prevent the situation when the RPC reply is sent while in the middle of updating the ADC measurements. The pseudo-code of this fragment is the following:

```
tempAdcMeasurements = getMeasurementsFromAdc();
disableInterrupts();
adcMeasurements = tempAdcMeasurements;
enableInterrupts();
```

This way, the interrupts are disabled only for a very short period of time, i.e. the time needed to copy the 6-element array. The client gets the measurements from `adcMeasurements` array.

Note that since the packet processing is done inside an interrupt, the client's request may be processed as soon as possible. To further minimize the response time, the interrupt is set as FIQ (fast interrupt), which is a special type of interrupt in ARM processors. The processor utilizes special additional registers for FIQ, so that the time of context switch when the processor enters/exit FIQ mode is much smaller in comparison to regular interrupts.

The network-related configuration parameters (e.g. IP address, etc) may be changed by modifying the file `$FIRMWARE_SRC_DIR\KiperDev\src\Server\KiperDev`.

The RPC service exposed by the force/torque sensor controller is the `Ati6284` service discussed before. The implementation of the `GetGenForceReading` method simply places the most recent force/torque readings in the response message (`GenForceReading`).

For the purpose of the testing another version of the firmware has been created; this version does not use KIPER and instead uses a simple protocol (described in section 4.3).

5.7. Components integration

This section describes how different components were integrated in order to prepare for the testing phase.

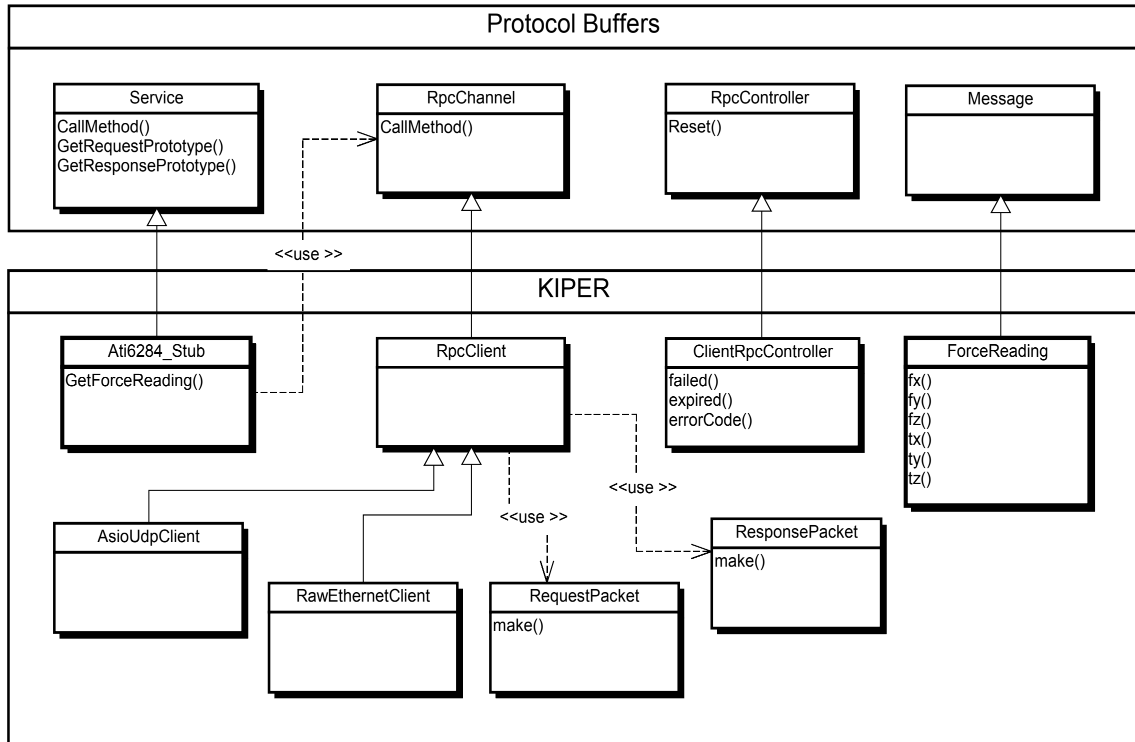

Figure 5.3. Client-side classes

Figure 5.3 shows the classes used by the client, which, in the test phase, is the standalone program (test stage 2) or MRROC++ (test stage 3, with KIPER used to retrieve force readings); also the modules to which the classes belong are shown. The client interacts with `Ati6284_Stub` class which is generated by the Protocol Buffers compiler. As an implementation of `Service`, it requires an instance of `RpcChannel`, it's possible to choose `AsioUdpClient` and `RawEthernetClient`. Those two implementations inherit from `RpcClient` which contains common methods dealing with packet processing, internally using `RequestPacket` and `ResponsePacket`. To call the `GetForceReading` method of the `Ati6284_Stub`, the client must create instances of `ClientRpcController` and `ForceReading`. The former is the implentation of the `RpcController` and enables setting/examining parameters of an RPC call. The latter is an automatically generated implementation of Protocol Buffers' `Message` and will be filled with force/torque readings after the call is complete.

Similarly, figure 5.4 shows the pseudo-classes (as described in section 5.3.1) used on the server when KIPER is used to retrieve force/torque measurements.

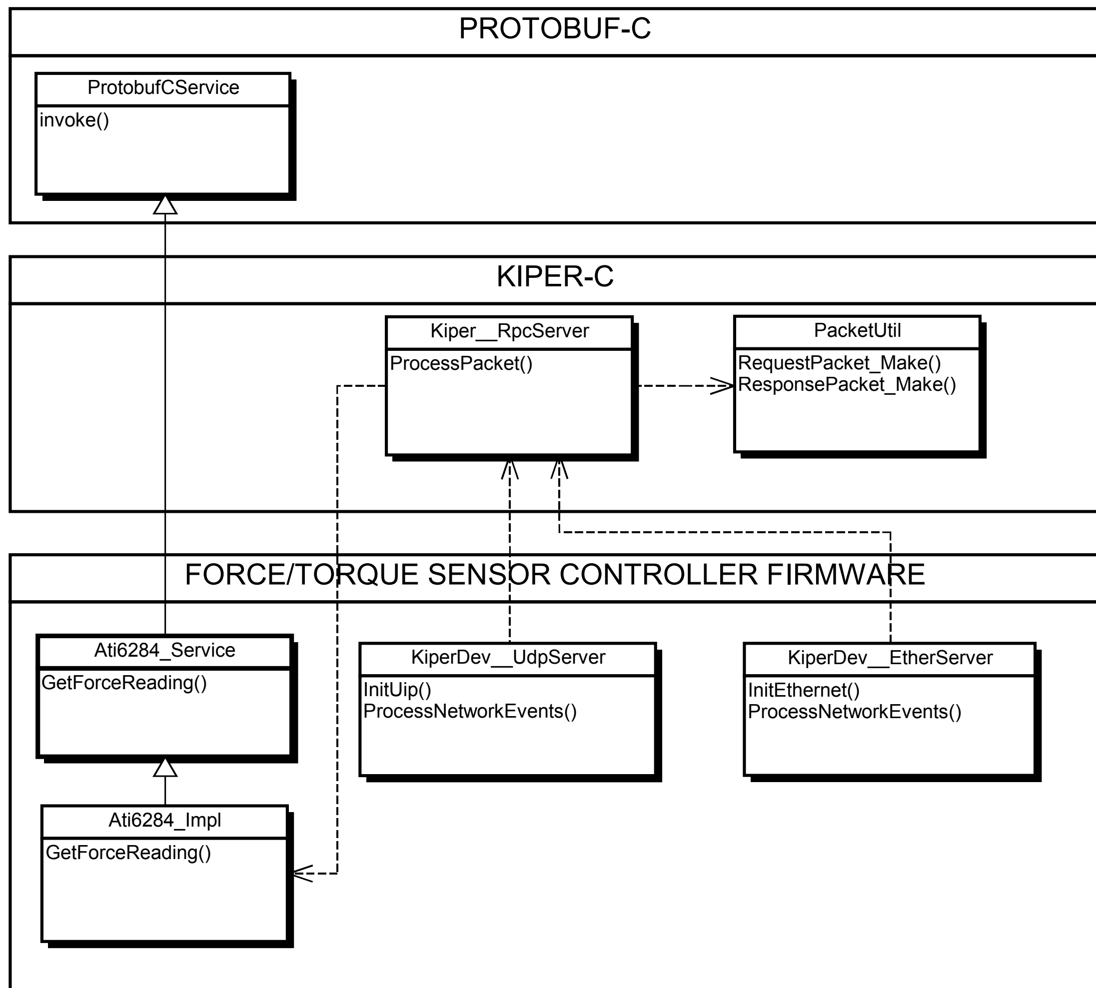

Figure 5.4. Server-side classes

When a network packet arrives it is handled by the `KiperDev__UdpServer` or `KiperDev__EtherServer` depending on which transport is used. The resulting Request packet (as described in chapter 4) with application data is passed to `Kiper__RpcServer` which decodes it and calls the method of the underlying `ProtobufCService`, in our case the `GetForceReading()` method of the `Ati6284_Impl`. Note that `Ati6284_Impl` is the „implementation" of the automatically generated `Ati6284_Service` pseudo-class. Again, `PacketUtil` contains functions facilitating packet processing.

For brevity, the diagrams describing the classes or used when the „simple protocol" (as in 4.3) is used are not presented here since the number of classes/pseudo-classes used on that occasion is small (on the client side there are two classes `ForceSensor` and `RawSocket` used internally by `Force Sensor`) and on the server side practically the whole work is done inside the interrupt service function.

5.8. Source code remarks

For the purpose of this thesis, different software modules were created: KIPER-CPP, KIPER-C and the firmware for force/torque sensor controller (basing on KIPER-C). Moreover, MRROC++ had to be modified to utilize KIPER. Some of the source directories contain „third-party" files or automatically generated files. First is the KIPER library, consisting of two modules, KIPER-CPP and KIPER-C. KIPER-CPP is based on Procol Buffers, in the sense that it links with the Protocol Buffers library and implements some of the Protocol Buffers interfaces; however there are no external source files in its source code apart from the ones generated by the Protocol Buffers compiler. KIPER is based on protobuf-c, but in the sense that it includes protobuf-c in its source code directory and implements some of protobuf-c interfaces. The `$KIPER_SRC_DIR/kiper-c/src/google` contains protobuf-c source files, modified by the author of this thesis to compile in IAR Embedded Workbench. Next is the firmware of the force/torque sensor controller. It was created basing on HTTP server example for LPC-P2378-SK development board which was included in the IAR Embedded Workbench suite. The source was stripped of unnecessary parts and modified according to the requirements. The files under `$FIRMWARE_SRC_DIR/src/Common/KiperDev/Ext/Ft6284.*` providing the code responsible for hardware interfacing with force/torque sensor were written by Kamil Tarkowski.

For the purpose of the thesis, additionally a modification of the EDP_FORCE process was required. As a base of a new version, the currently existing version by K. Tarkowski located under `$MRROCPP_SRC_DIR/src/edp/ati6284KT` was copied to `$MRROCPP_SRC_DIR/src/edp/ati6284MS` and became a base for a new version.

6. Using KIPER

6.1. Using KIPER-CPP

This section describes how to build, install and use KIPER-CPP under QNX 6.4.1.

6.1.1. Prerequisites

The requirements for building KIPER-CPP under QNX 6.4.1 are the following:
— CMake 2.6.0 or newer,
— Protocol Buffers 2.0.3 or newer, and
— Boost 1.38.0 or newer.
If Protocol Buffers or Boost are installed in non-standard locations, appropriate environmental variables need to be set when running CMake configuration.

Under QNX, KIPER-CPP has to be build using QCC. Even though QCC is a wrapper around gcc/g++, the default implementation of C++ standard library is set to Dinkumware. To build and link to Protocol Buffers library, the currently selected implemenation of the C++ standard library for QCC has to be set to GCC standard library. For example, for QNX 6.4.1 and gcc 4.3.3, in the file `/usr/qnx641/host/qnx6/x86/etc/qcc/gcc/4.3.3/default`, the line

```
CONF=gcc_ntox86
```

has to be replaced with

```
CONF=gcc_ntox86_gpp
```

6.1.2. Building and installation

To build KIPER under QNX 6.4.1 it is required to:
1. Unpack the source code or check it out from the repository, to e.g. `/home/user/kiper/src`;

2. Create a separate build directory, for example beside the source directory, e.g. /home/user/kiper/build;
3. Change the current directory to the build directory:

```
cd /home/usr/kiper/build
```

4. Configure CMake:

```
cmake ../src/kiper -DCMAKE_CXX_COMPILER=QCC
-DCMAKE_BUILD_TYPE=RelWithDebInfo
-DCMAKE_INSTALL_PREFIX=/opt
```

This instructs CMake to generate Makefiles for KIPER-CPP. The following CMake variables need to be set:
— CMAKE_CXX_COMPILER: specifies the C++ compiler to be used for compiling the sources. Setting this varbiable to QCC is mandatory on QNX, otherwise g++ compiler will be used and build will fail later.
— CMAKE_BUILD_TYPE: specifies the build type. This controls debug and optimization flags for compiler and linker. RelWithDebInfo (release with debug info) should be as good choice for most cases - it enables the optimizations while including the debug information in resulting binary files;
— CMAKE_INSTALL_PREFIX: specifies the installation directory.
For full discussion of CMake variables refer to [41].
If Boost or Protocol Buffers are installed in non-standard locations, it is required to set the appropriate environmental variables when configuring CMake. Let's assume that Boost is installed in /home/user/libs/boost and Protocol Buffers is installed in /home/user/libs/protobuf. To configure CMake with non-standard locations of libraries, execute

```
BOOST_ROOT=/home/user/libs/boost;
PROTOBUF_ROOT=/home/user/libs/boost
cmake ../src -DCMAKE_CXX_COMPILER=QCC
-DCMAKE_BUILD_TYPE=RelWithDebInfo
-DCMAKE_INSTALL_PREFIX=/opt
```

Additionally, it's possible to set the following environmental variables that give more fine-grained control over the location of headers and libraries:
— BOOST_INCLUDEDIR the location of Boost header files
— BOOST_LIBRARYDIR the location of Boost compiled libraries
— PROTOBUF_INCLUDE_DIR the location of Protocol Buffers header files

— `PROTOBUF_LIBRARY_DIR` the location of Protocol Buffers compiled libraries

5. Compile KIPER-CPP:

```
make
```

6. Install KIPER-CPP:

```
make install
```

The following files are installed:
— header files in `$KIPER_CPP_INSTALL_DIR/include`,
— library in `$KIPER_CPP_INSTALL_DIR/lib`,
— additional CMake modules in `$KIPER_CPP_INSTALL_DIR/shared/cmake`. These modules may be used to find and configure KIPER-CPP and Protocol Buffers libraries, and, in the case of Protocol Buffers, to automatically compile `.proto` files.
— common `.proto` files in `$KIPER_CPP_INSTALL_DIR/shared/proto`.

6.1.3. Example usage

This section explains how to:
— write a `.proto` file that defines an RPC service and the format of the exchanged messages,
— compile the `.proto` file,
— use the generated classes from within KIPER-CPP.

Writing .proto file

We will begin with the example `.proto` file. For the complete reference of the `.proto` files refer to [42]. Here is the complete listing of a `.proto` file, as used in the example:

```
1  package kiper.test;
   import "Common.proto";

   message TrimStringRequest {
5      required string str = 1;
       required uint32 length = 2;
   }

   message TrimStringResponse {
10     required string str = 1;
   }
```

```
message SendStringRequest {
    required string str = 1;
}

message GetStringRequest {
    required string str = 1;
}

service Test {
    rpc trimString (TrimStringRequest) returns (TrimStringResponse);
    rpc sendString (SendStringRequest) returns (VoidMsg);
    rpc getString (VoidMsg) returns GetStringResponse;
}
```

Below is an explanation of the individual sections of the `.proto` file.

```
package kiper.test;
import "Common.proto";
```

First, we set the package name to `kiper.test`. This dictates how the message names will be referred to from within the external `.proto` and sets the name of the namespace of the generated C++ classes. Next we import the file `Common.proto` which defines the special type of message, `VoidMsg`.

```
message TrimStringRequest {
    required string str = 1;
    required uint32 length = 2;
}

message TrimStringResponse {
    required string str = 1;
}

message SendStringRequest {
    required string str = 1;
}

message GetStringResponse {
    required string str = 1;
}
```

We define four message types that will be used in the methods. Each message defines a required string field (str). Additionally `TrimStringRequest` defines an `uint32` (32-bit integer) field named length.

```
rpc trimString (TrimStringRequest) returns (TrimStringResponse);
```

This line defines an RPC method named `trimString` that accepts `TrimStringRequest` as a request and returns `TrimStringResponse` as a response. The implementation of this simple method trims the string to the given length. `TrimStringRequest` defines two fields: `str` and `length`. The `str` field is the string to trim, while `length` is the trim length. After the method call completes successfully, the `str` field of `TrimStringResponse` message will contain the trimmed string.

```
    rpc sendString (SendStringRequest) returns (VoidMsg);
```

This line defines an RPC method named `sendString` that accepts `SendStringRequest` as a request. Setting `VoidMsg` as a response means, that the client does not expect any response for this RPC request.

```
    rpc getString (VoidMsg) returns GetStringResponse;
```

This line defines an RPC method named `getString` that returns `GetStringResponse` as a response. Setting `VoidMsg` as a request means that there are no arguments (input parameters) for this method. In this case there will be no serialization of request message. This optimization might be useful when the request needs to be sent as quickly as possible and there are no request parameters.

Compiling .proto file

To compile the defined `.proto` files use the `protoc` compiler. The syntax is `protoc [OPTION] PROTO_FILES`. Refer to the manual for a complete description.

Using generated classes from KIPER-CPP

This is the simplified description of how to use the generated classes to call remote methods. For the full source code refer to the examples in `$KIPER_SRC_DIR/kiper/example-src`.

To call a remote method, first we need an instance of RPC Client. In this example we will use the UDP client. The remote host in this example is `192.168.18.200` while the port is `55555`.

```
AsioUdpClient client("192.168.18.200", 55555);
```

Having an RPC client we may now create a stub for our service. We will use this stub to call remote methods.

```
Test_Stub service(&client);
```

For each RPC method call we need an RPC controller. Optionally we may change the default timeout, to, let's say, 1 millisecond.

```
ClientRpcController controller;
controller.setTimeout(boost::posix_time::milliseconds(1));
```

Each RPC method needs a request (input) message and response (output) message. We create `TrimStringRequest` and set the required fields. Then we create `TrimStringReply`. After the method call completes, the `str` field of `TrimStringReply` will contain the trimmed string.

```
TrimStringRequest trimStringRequest;
trimStringRequest.set_str("Hello,␣world!");
trimStringRequest.set_length(5);
TrimStringReply trimStringResponse;
```

Now we may call the RPC method:

```
service.trimString(&controller, &trimStringRequest,
  &trimStringResponse, NULL);
```

The fourth argument of each generated method is the callback to be run when the method call completes. The intention of the authors of Protocol Buffers was that this callback should be executed when the response arrives. However, due to the fact that the method calls are synchronous in KIPER-CPP (the calling thread blocks until response arrives or timeout expires), there is no need to pass a callback function to the method call. Therefore, the fourth argument to the `trimString` method is `NULL`.

After the call finishes, we may examine the RPC controller instance to check for any logical errors returned by the server or check if timeout occurred:

```
if (controller.failed()) {
  // server returned an error.
  // error code is controller.errorCode()
  // error message is controller.errorMessage()
} else if (controller.expired()) {
```

```
  // timeout occurred
} else {
  // method call was successful
}
```

If there were no errors, `trimStringResponse` will contain the result of the method call, obtained by calling `trimStringResponse.str()`.

RPC controller can be reused across call by calling `reset()`:

```
controller.reset();
```

This clears any „output" data of an RPC call (i.e. error, expiration) but leaves intact „input" data (i.e. timeout). So, if the timeout for the RPC controller was set to e.g. $500mS$, and `reset()` method was called, there is no need to set the timeout again. Thanks to this method it is not required to create a new instance of `RpcController` for each method call.

If a method defines `VoidMsg` as a response message, i.e. the server is not expected to send a response for the request, `NULL` should be passed as response message and as controller:

```
SendStringRequest sendStringRequest;
sendStringRequest.set_str("some string");
service.sendString(NULL, &sendStringRequest, NULL, NULL);
```

We don't need RPC controller in this case, because since we do not expect the server to send response, we don't need to set timeout nor do we need to examine the RPC controller for errors. Note, that in this situation the client cannot know if the method call succeeded or not.

If a method defines `VoidMsg` as a request message, i.e. it does not have input arguments and therefore does not require request message serialization, `NULL` should be passed as a request message:

```
GetStringResponse getStringResponse;
service.getString(&controller, NULL, &getStringResponse, NULL);
```

This instructs `KIPER-CPP` not to serialize the request message.

6.1.4. RPC Server

This section describes how to use KIPER-CPP as RPC server. In this scenario the machine running KIPER-CPP exposes an RPC service, and a remote device/computer calls the methods exposed by the service. Not that the current test setup did not take advantage of this functionality (the embedded device acted as RPC server).

KIPER-CPP includes a `RpcServer` class that processes RPC requests, forwards them to the implementation of the service and sends the reply, if required. To make this class work with a particular transport mechanism, another class will typically wrap the `RpcServer` class and call its `processPacket()` method upon receiving a packet. This is exactly how `AsioUdpServer` class works.

Compiling a `.proto` file containing the definition of an RPC service results in a header file with an abstract interface. The Protocol Buffers compiler automatically generates a stub implementation of this interface. The client uses this stub to call the remote methods. On the other hand, the server has to provide its own implementation of the service interface.

The service interface class generated by the Protocol Buffer compiler looks the following:

```
class Test : public ::google::protobuf::Service {
protected:
  inline Test() {};
public:
  virtual ~Test();
  virtual void trimString(google::protobuf::RpcController* controller,
    const ::kiper::TrimStringRequest* request,
    ::kiper::TrimStringReply* response,
    ::google::protobuf::Closure* done);
  virtual void sendString(google::protobuf::RpcController* controller,
    const ::kiper::SendStringRequest* request,
    ::kiper::VoidMsg* response,
    google::protobuf::Closure* done);
  virtual void getString(google::protobuf::RpcController* controller,
    const ::kiper::VoidMsg* request,
    ::kiper::GetStringResponse* response,
    ::google::protobuf::Closure* done);
  (...)
};
```

The RPC Server needs an implementation of this interface which we need to provide.

```
class TestImpl : public Test {
public:
    TestImpl();
    ~TestImpl();
    void trimString(google::protobuf::RpcController* controller,
       const kiper::TrimStringRequest* request,
```

```
        kiper::TrimStringReply* response,
        google::protobuf::Closure* done) {
      // the implementation of this method
    }

      void sendString(::google::protobuf::RpcController* controller,
        const kiper::SendStringRequest* request,
        kiper::VoidMsg* response,
        google::protobuf::Closure* done) {
      // the implementation of this method
    }
    (...)
protected:
private:
};
```

To create an RPC Server, first we create an instance of the service implementation:

```
TestImpl service;
```

Next, we will create an RPC Server using UDP as transport and listening on UDP port 55555.

```
AsioUdpServer rpcServer(&service, 55555);
```

Finally, we instruct the RPC server to start processing the requests.

```
rpcServer.run();
```

6.1.5. Using CMake to compile .proto files

The standard installation of CMake comes with many modules for configuring external libraries. For example, the `FindBoost.cmake` module searches for Boost headers and compiled libraries, and defines environmental variables with locations of headers and libraries that may be passed to the compiler.

KIPER-CPP is built using CMake and uses Protocol Buffers library, but at the time of this writing there was not official module for finding Protocol Buffers or calling protobuf compiler. Therefore it has been decided to write such a module. The module is located at `$KIPER_CPP_INSTALL_DIR/shared/cmake/FindProtobuf.cmake`. The module also allows to compile `.proto` files automatically.

To use the `FindProtobuf.cmake` its location needs to be added to CMake module path. Assuming that KIPER-CPP is installed in `/home/user/libs/kiper`, the following line should be added:

```
SET(CMAKE_MODULE_PATH /home/user/libs/kiper/shared/cmake)
```

To locate and configure the Protocol Buffers library add the following entry to the CMake configuration file:

```
FIND_PACKAGE(Protobuf REQUIRED)
```

The `REQUIRED` option forces CMake to stop processing when Protocol Buffers library is not found. If Protocol Buffers library was found, the module defines the following CMake variables:

— `Protobuf_INCLUDE_DIR`: the directory where the Protocol Buffers include files are located
— `Protobuf_LIBRARY`: the path to the Protocol Buffers compiled library
— `Protobuf_COMPILER`: the path to the Protocol Buffers compiler executable

Typically the above variables would be used in the following way:

```
INCLUDE_DIRECTORIES(${Protobuf_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(some_target ${Protobuf_LIBRARY})
```

which does the following:
— adds Protocol Buffers' include directory to the list of directories in which the compiler searches for header files;
— instructs the compiler to link `some_target` against the Protocol Buffers library. In general, `some_target` may be an executable or library built by CMake. If it is an executable, the Protocol Buffers library will be added to the list of libraries against which this executable is linked. If it is a library, it adds a transitive dependency on the Protocol Buffers library, i.e. if an executable is linked against `some_target` it will be automatically linked against Protocol Buffers library.

Using `FindProtobuf.cmake` module allows to automatically compile `.proto` files and add resulting source and header files to the build using `PROTOBUF_COMPILE` command. The `.proto` files are only recompiled whey they change. The syntax of the command is the following:

```
PROTOBUF_COMPILE(var proto_files output_dir [include_path])
```

where the meaning of the arguments is:

— `var`: The name of the variable that will contain the header and source files generated by the Protocol Buffers compiler
— `proto_files`: the list of the `.proto` files to compile
— `output_dir`: the name of the directory to put the generated files in
— `include_path` (optional): the include path passed to the Protocol Buffers compiler. This should be used to add the path to the `Common.proto` file which defines `VoidMsg`

The list of the generated files (placed in the variable supplied as the second argument to `PROTOBUF_COMPILE` command) must be added to the build, i.e. as an argument to the `ADD_EXECUTABLE` or `ADD_LIBRARY` command. Refer to `$KIPER_SRC_DIR/kiper/example.cmake` for an example on how to use `PROTOBUF_COMPILE` command.

6.1.6. Transport mechanisms

KIPER-CPP comes with two transport mechanisms: UDP (using Boost ASIO) and RAW Ethernet(using BPF BPF). The constructor of the `AsioUdpClient` is the following:

```
AsioUdpClient(const char* host, unsigned int port);
```

where `host` is the address of the remote computer (IP address or host name) and `port` is the UDP port to connect to.

Another type of client is `RawEthernetClient`. Its constructor has the following signature:

```
RawEthernetClient(const char* device, uint8_t mac[6]);
```

where `device` is the name of the network interface to use (e.g. `en0`) and `mac` is the host MAC address.

To implement a transport for an RPC client one has to subclass the `RpcClient` class. The subclass is at minimum required to implement the following methods:

```
virtual void sendRequest(std::size_t size);
virtual std::size_t sendRequestAndReceiveResponse(std::size_t size,
  boost::posix_time::time_duration timeout);
```

The `sendRequest()` method should send `size` bytes to the destination host. The data to be sent are obtained using `getSendBuffer()`. The `sendRequestAndReceiveResponse` should send `size` bytes to the destination host and wait for the reply for the maximum duration of `timeout`. The data to be sent

are obtained using `getSendBuffer()`, while the received reply needs to be placed in the buffer obtained using `getRecvBuffer()`.

6.2. Using KIPER-C

6.2.1. Building

When using KIPER-C on an embedded device, it is not required to build it separately, as an external library. However, the source tree includes some functional tests that run on a PC (see chapter 7). These tests require the protobuf-c library to be installed. The procedure to build KIPER-C (including tests) on a PC is similar to the one for building KIPER-CPP (see section 6.1.1). The main differences are:

— the source code for directory that needs to be provided as an argument to CMake is `$KIPER_SRC_DIR/kiper-c`;

— protobuf-c is required for building - the appropriate environmental variables are `$PROTOBUFC_ROOT`, `PROTOBUFC_INCLUDE_DIR`, `PROTOBUFC_LIBRARY_DIR` and `PROTOBUFC_COMPILER_DIR` (see the description of the respective variables that are used when building KIPER-CPP);

6.2.2. Example

RPC Client

Although not used in tests, KIPER-C may also be used as the RPC client. the steps are given below.

First, a callback function (that will process the result of the RPC call is required.

```
static void processTrimStringReply(const
Kiper__TrimStringReply* message,
  void* closureData) {
    Kiper__RpcController* rpcController = closureData;
    if (rpcController->timeout) {
    // timeout occurred.
    }
}
```

Next, the RPC client and the underlying transport are created.

```
static UdpClientTransport clientTransport;
static Kiper__RpcClient rpcClient;
UdpClientTransport_init(&clientTransport, "192.168.18.38", 55555);
Kiper__RpcClient_init(&rpcClient,
```

```
  (Kiper__ClientTransport*) &clientTransport,
  &kiper__test__descriptor);
```

Then, the RPC controller is initialized.

```
static Kiper__RpcController rpcController;
Kiper__RpcController_init(&rpcController);
```

Finally, the RPC call is performed.

```
kiper__test__trim_string((ProtobufCService*) &rpcClient, &request,
  processTrimStringReply, &rpcController);
```

After the response to the call arrives, KIPER-C will automatically the `processTrimStringReply` function.

RPC Server

The scenario used in the testing phase assumed that force/torque sensor controller is the RPC sever. Below is the description of the steps required to initialize such server using KIPER-C.

First, the implementation of the service interface is created and initialized:

```
Kiper__TestImpl testServiceImpl;
Kiper__TestImpl_init(&testServiceImpl);
```

Next, the transport mechanism is created and initialized. In this example, the transport is UDP running on port 55555.

```
UdpServerTransport transport;
UdpServerTransport_init(&transport, 55555);
```

Finally, an RPC server is created, using the transport and service created above:

```
Kiper__RpcServer rpcServer;
Kiper__RpcServer_init(&rpcServer, &transport.base, (ProtobufCService*)
  &testServiceImpl);
```

Upon receiving the raw packet data, the `Kiper__RpcServer_processPacket()` function is called to analyse it and correlate it with the pending request. If the data represent a correct response packet, a correct RPC method is called internally and the response is sent (if required).

```
Kiper__RpcServer_processPacket(&rpcServer, recvBuffer, recvLen,
  &clientAddr);
```

6.3. The LPC-P2378-SK firmware

6.3.1. Prerequisites

Te requirements for building and uploading the firmware to LPC-P2378-SK are the following:

— IAR Embedded Workbench for ARM version 5.40 or later
— H-JTAG, version 0.9.2 or later

IAR Embedded Workbench will be used to build the firmware and perform in-system debugging, while H-JTAG will be used to upload the firmware to the board's flash. Both programs run under Windows XP or later.

6.3.2. Building

To build LPC-P2378-SK firmware:

— Create a common directory for LPC-P2378-SK firmware source and KIPER source, e.g. `C:\LPC-FIRMWARE` – this directory will be referred to as `$FIRMWARE_SRC_DIR`;
— Unpack the KIPER source code, or check it out from the repository, to `$FIRMWARE_SRC_DIR/kiper`;
— Unpack the LPC-2378-SK firmware source code, or check it out from the repository, to `$FIRMWARE_SRC_DIR/KiperDev`;
— Open the IAR Embedded Workbench and load the workspace `$FIRMWARE_SRC_DIR/KiperDev/KiperDev.eww`. This workspace contains two projects: „server" (transport via Raw Ethernet) and „server-raw" (transport via UDP);
— Press F7 or choose Project->Make.

To upload the firmware to LPC-P2378-SK development board:

— Make sure that the board is properly connected for programming (usually via parallel port);
— Start H-JTAG;
— Start H-FLASHER;
— Under „Flash Selection" choose „NXP"->„LPC2378";
— Under „Configuration" enter 12 as Ext XTAL;
— Under „Programming" select „Intel Hex Format" as type and `$FIRMWARE_SRC_DIR/KiperDev/bin/server/Debug/Exe/server.hex` for UDP based server and `$FIRMWARE_SRC_DIR/KiperDev/bin/server-raw/Debug/Exe/server.hex` for raw Ethernet based server;

— Click 'Program'.

6.3.3. Logging

KIPER-C contains some helpful macros located in the header file `$KIPER_SRC_DIR/kiper-c/library-src/kiper-c/Log.h`. Those macros allow to log messages using syntax similar to that of `printf`, e.g.

```
KIPER_LOG_DEBUG("Number␣of␣requests␣=␣%d", num_requests);
```

The macros are used in force/torque sensor controller firmware to output debug messages to the serial port.

7. Testing

KIPER testing was divided into thee distinct phases:

1. Initial tests without force/torque sensor controller
2. Tests with force/torque sensor controller
3. Tests with MRROC++

Each test phase served a different purpose: the first phase validated core functionality of KIPER from the functional point of view; the second phase tested force/torque sensor controller while using KIPER to retrieve measurements and finally the third phase exercised a full-fledged robot-control setup with MRROC++. During phases two and three the solution using KIPER was compared to the one utilizing simple request-reply protocol (without serialization), as described in 4.3.

7.1. Initial tests

The purpose of the first stage of testing was the validation of the hardware-independent part of the KIPER framework, i.e. the KIPER-CPP and KIPER-C libraries. This was carried out by writing example code consisting of two parts: the client, utilizing KIPER-CPP, and the server, utilizing KIPER-C. (Similar configuration is also found in the second and the third stages.) These tests were devised to verify whether calling remote methods works correctly (e.g. whether requests and responses are properly serialized/deserialized) with the intention of being run on two PCs (one running the client and another one running the server), in contrast to second and third stages, where force/torque sensor controller was acting as RPC server. An RPC service definition was created for the purpose of such testing. This basic test was not concerned with performance, only logical correctness was verified. Any major mistakes in KIPER-CPP or KIPER-C (e.g. in serialization code) would be immediately revealed during such test. The source code for the test cases is located in `$KIPER_SRC_DIR/kiper/test-src` (client) and `$KIPER_SRC_DIR/kiper-c/test-src` (server). The test cases were prepared to cover different scenarios of remote procedure calls according to KIPER protocol capabilities. This includes regular, successful calls, calls in which the server returns an error, calls that end with timeout, calls with „void” request and response

messages, etc. To run the test cases, KIPER-CPP as well as KIPER-C need to be built on the PC (see sections 6.1.1 and 6.2.1). It is worth mentioning, that building KIPER-C on the PC server no purpose other than running the aforementioned test cases, because normally KIPER-C is built the same time the device's firmware is built. The tests use UDP as transport mechanism. The executable binaries required to perform the test are `$KIPER_BUILD_DIR/kiper-test-client` and `$KIPERC_BUILD_DIR/kiperc-test-server` for the client and server respectively. In our example we will setup the test to be carried out on a local machine, however the test may be as well run on two separate machines. To begin the test, first, the server needs to be run from the KIPER-C build directory:

```
kiperc-test-server 55555
```

where `55555` is the UDP port the server should listen on. If server was started without problems, the client may be run, from the KIPER build directory:

```
kipercpp-test-client localhost 55555
```

Where `localhost` is the host to connect to (in our case, the same local machine that the server was started on) and `55555` is the UDP port. This will execute the series of tests cases; The client calls the remote procedures in a fixed order and checks response is correct (e.g. the contents of the response message are valid, or if error was returned as expected), with the exception of one test case, when it expects the timeout to occur. On the other hand, the server expects that the messages come in the given order, verifies their contents and sends the response where required.

This first stage of testing also allowed to verify whether UDP transport for KIPER-CPP works correctly. As for KIPER-C, the UDP server that was written, was only used in this first stage, because it was written for the PC, while in the next stage, it is the force/torque sensor controller that acts as the RPC server.

7.2. Testing with force/torque sensor controller

After verifying that the test cases from the first stage execute without errors, the next step was to test with force/torque sensor controller. This second stage of the testing was, in a sense, a preparation for the third stage. The hardware configuration of the test was almost identical. (With the exception that the third

stage included an additional PC on which MRROC++ processes were run.) The purpose was to verify whether retrieving ADC readings from the force/torque sensor controller works correctly, without engaging MRROC++ yet.

A test setup, based on the equipment present in the robotics laboratory at the Institute of Control and Computation Engineering, was utilized. The experimental setup (see 7.1 for the purpose of the second and third stages of testing consisted of:

— IRp-6 — the 6-dof manipulator with a gripper and a force/torque sensor,
— Two networked PC's (robot2 and olin), running the QNX 6.4.1 operating system and, in the third stage, the processes of MRROC++ framework,
— the force/torque sensor controller based on a modified LPC-P2378-SK development board, augmented with the ADC and other necessary integrated circuits.

The hardware configuration of the PC's, was the following:

robot2 Intel Pentium III 1000MHz, 512MB RAM,

olin AMD Phenom II X3 710, 4096MB RAM (used only in third stage of testing)

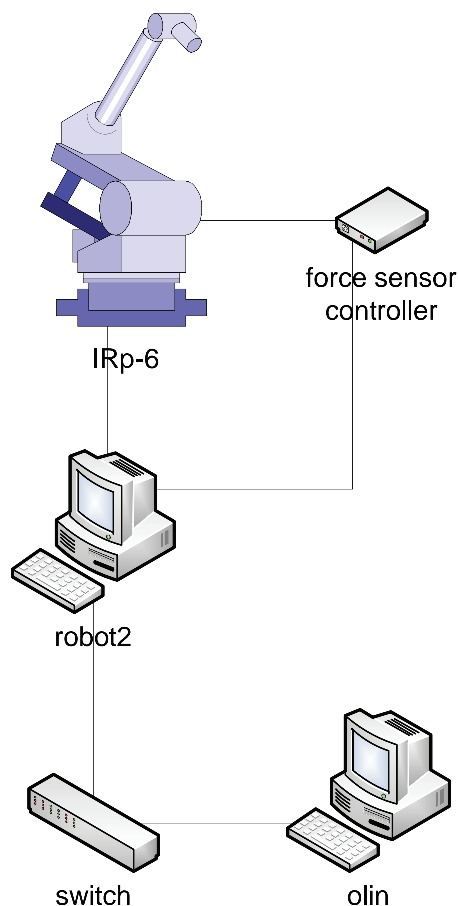

Figure 7.1. Test hardware setup

The force/torque sensor controller was loaded with firmware, as described in section 5.6. During the tests it was than connected to the dedicated Ethernet adapter of robot2 via standard Ethernet. Physically, the force/torque sensor controller and robot2 PC were connected using Ethernet cable. However, from the application point of view, the data were sent differently. One version of the firmware utilized a plain dedicated protocol with messages sent using raw Ethernet with a fixed message format; this version was used in the first test (see 4.3). The other version employed KIPER (measurements were obtained by calling a remote procedure with force/torque sensor controller acting as RPC server). There were two variants of this version: One, using raw Ethernet (used in the second test), and another, using UDP (used in the third test).

To measure the performance of such solution, and to compare the performance of different hardware and software configurations, the test code was instrumented with functions for measuring the latency (the time between requesting the force/-torque reading, and getting the reading). To accurately measure elapsed time, the `ClockCycles()` — A QNX specific function — was used. This function returns the current value of 64-bit CPU cycles counter (on x86 architecture it utilizes the RDTSC assembly instruction). In order to reliably use this function on multi-processor or multi-core systems, the thread which calls the function must be locked to a single core, using `ThreadCtl()` call:

```
ThreadCtl(_NTO_TCTL_RUNMASK, ...)
```

In our case the thread should be locked to the first core, as sometimes the application is run on a single-core machine.

Having the number of elapsed cycles between events and the CPU speed in Hertz, it is possible to compute the elapsed time (in microseconds):

$$time_{\mu s} = \frac{cycles}{\frac{CPU_CLOCK}{1000000}} \tag{7.1}$$

With the force/torque sensor controller loaded with new firmware, the test simply calls the service classes generated by protocol buffers using KIPER-CPP as the implementation of the RPC channel, to retrieve the ADC readings from the sensor. For each RPC call, the time between calling the method and receiving the response is measured. This is executed in a loop, a given number of times. At the end of the program execution, the latency measurements are written to a file. The first test used the UDP as the transport mechanism, while the second one used the raw Ethernet. For comparison purposes, a third test was also executed. This test received the ADC readings from the sensor without using protocol buffers (the

readings were encoded in a simple packet structure). Each test required a different firmware loaded into sensor controller. In each of the three tests the procedure was the same:

1. Load the force-torque sensor controller with appropriate firmware
2. Reset the force-torque sensor controller
3. Run the appropriate client binary on the `robot2` PC.
4. Collect the results (the file with latency measurements)

For each file with latency measurements, selected statistical properties (mean, standard deviation and 95th percentile) were calculated. Table 7.1 presents the results.

Table 7.1. Results of the second stage of testing

test no	mean (μS)	sd (μS)	95th (μS)
1	111.347500	37.931610	123.000000
2	170.070130	28.233760	173.000000
3	223.060000	49.000000	204.000000

There are couple of things to notice about the result of the initial tests. First, as expected, as far as the mean and 95th percentile values are concerned, the first test (not using KIPER) is the fastest, because there is no serialization/deserialization overhead. In this test, the force/torque measurements are retrieved using simple, custom protocol packed directly into Ethernet frames.

It is also not a surprise that the version utilizing KIPER and raw Ethernet as transport mechanism came up second and the version utilizing KIPER and UDP as transport mechanism the last one. These variations pack the data serialized by Protocol Buffers into raw Ethernet frames and UDP datagrams respectively.

While the KIPER based versions are slower, they offer a more than acceptable performance for the given task, as far as mean latency and stability (as measured by standard deviation) are concerned. Because of the slight advantage of the raw Ethernet version (test 1) over the UDP version (test 2), both using KIPER, the version using raw Ethernet was chosen for further tests.

7.3. Testing with MRROC++

In the third, final stage of testing, KIPER was employed in a real-time robot control scenario. Two networked PCs (robot2 and olin) were running MRROC++ programming framework; robot2 PC was directly interfaced with IRP-6, a robotic manipulator with a gripper. A force/torque sensor, attached to the gripper, was

connected to a force/torque sensor controller which, in turn, was connected to robot2 via Ethernet, as in previous stage. MRROC++ used the raw ADC readings obtained from the force/torque sensor controller to compute forces/torques acting on the gripper; these readings were used in the contour following task which employs force control (In force control, not only the motion of robot is controlled, but also the force that it exerts on the en vironment.) As in previous stage of testing, the ADC readings were retrieved either using simple dedicated protocol, or with the aid of KIPER; those variants were compared as far as the performance is concerned.

The robot control setup utilized in third stage is in fact a modified version of a previous, existing setup in which the force/torque sensor controller was connected to robot2 via ISA bus. In an effort to replace the aging interface with a contemporary solution, a new force/torque controller was devised. There were different options as to which interface to use to interface the new controller with the PC. Ultimately, Ethernet was chosen for reasons including performance, simplicity and price. In the basic version, the ADC measurements are retrieved using a dedicated request/response protocol over raw Ethernet. However, to validate if we may use RPC in a real-time, control system, the second version uses KIPER while maintaining the Ethernet as physical connection; the measurements are obtained by calling a remote procedure.

In addition the robot2 PC used in second stage of the testing, the third stage introduces another, more powerful PC - olin. This reflects the standard usage of MRROC++, as the framework is designed to be run in a distributed fashion — different processes may be executed on different machines as long as the machines can communicate within a network. The QNX operating system facilitates this by providing services for transparent communication between processes, irrespectively of the physical machine the process is running on. The ability to run different processes of MRROC++ on different machines allows, for instance, to offload some computations to a more powerful machine. As an example, consider the two computers used in the test; of these only robot2 is directly interfaced with IRp-6 robot — this computer is responsible for direct control of the robot's joints via ISA bus. This implies, that in real mode (when MRROC++ interacts with robot hardware), the EDP process needs to be run on robot2. By no means is this machine a „speed demon" by today's standards, however it has the ISA bus required to communicate with robot's hardware. Therefore it makes sense to run the EDP process on robot2 and the other processes (UI, ECP, MP) on a more powerful machine; in case of this

test — `olin` (A brief description of MRROC++ processes was presented in section 5.4)

As described in section 5.4, the EDP_FORCE process was modified to obtain the ADC readings from the force/torque sensor controller using the code similar to the one described in previous section. Inside the EDP process, the raw ADC readings obtained from the force/torque sensor controller were converted to force/torque readings and subsequently used in control law.

As in the previous testing stage, the latency measurements were performed. This time, however, the measurements were attached to the reader interface of MRROC++ (The reader is normally used for other purposes such as force measurements.) Starting the reader from the MRROC++ UI also starts the latency measurements. Each measured time is stored in an array. When the reader is stopped, the file with measurements is written to disk. (This is completely independent from the files produced by the reader — the latency measurements were attached to the reader interface purely as a convenience.)

The contour-following task used in the tests consists of two phases: in the first phase, the manipulator is compliant – the operator may reposition the manipulator by applying the force to the gripper (in which the force/torque sensor is installed). During this phase it should be evident, whether force measurements work correctly — the manipulator should smoothly follow operator's movements. Any „jerks" that occur during repositioning of the robot arm indicate problems with force measurements.

In the second phase, the robot follows the unknown contour according to the task configuration. The practical difference between the first and the second phase is that the generated trajectory is different (instead of following operator's movements the robot follows the unknown contour). However the basic principle of operation is the same. Therefore, for simplicity, only the first phase of the task was used. The testing covered six similar test cases. In each case, the MRROC++ UI was run on `olin` and then other MRROC++ processes were started. (The location of processes on the machines was set in the task configuration file.) After starting the EDP, ECP and MP processes the contour-following task was started and the reader enabled. The compliant robotic arm was then forced to move in different directions by applying force to the gripper, for about 30 seconds. After that time the reader was stopped, the MRROC++ processes unloaded and the measurements collected.

For each of the test cases, the configuration of MRROC++ was set by customizing the task configuration file. In our case, the file was

$MRROCPP_INSTALL_DIR/src/app/sk_postument.ini. (This is the configuration file for the contour following task.) This file allows, most importantly, to specify the machine on which each of the MRROC++ processes should be run. The exact steps in each test instance were the following:

1. start the MRROC++ UI;
2. load the EDP process;
3. load the MP process;
4. start the task;
5. start the reader;
6. wait 30 seconds;
7. stop the reader;

The processes were run on two computers: robot2 (slower, but required due to the hardware connection with the robot) and olin (more powerful, multi-core). robot2 had two network adapters: one dedicated adapter for communication with force-torque controller and another for communication with other PC's running QNX, including olin.

A parameter that changed between the cases was the location of the MRROC++ processes. The UI process was always run on olin, while the EDP process was always run on robot2. The possible combinations of other processes were the following:

— EDP, ECP and MP on robot2,
— EDP, ECP on robot2,
— EDP on robot2

Note that the MRROC++ was running in the test (simulated) mode; in this mode the principle of the operation (the processes created, the communication between processes) is the same but there is no direct interaction with robot's hardware.

Table 7.2. Results of the third stage of testing

test no.	protocol	EDP	ECP	MP	UI
1	simple protocol	robot2	robot2	robot2	olin
2	simple protocol	robot2	robot2	olin	olin
3	simple protocol	robot2	olin	olin	olin
4	KIPER	robot2	robot2	robot2	olin
5	KIPER	robot2	robot2	olin	olin
6	KIPER	robot2	olin	olin	olin

Table 7.2 summarizes the configuration of each the test cases. The columns of the table have the following meaning:

protocol — whether KIPER with raw Ethernet or simple protocol was used to communicate with the sensor controller

EDP — the machine on which EDP was run

ECP — the machine on which ECP was run

MP — the machine on which MP was run

UI — the machine on which UI was run

7.4. Results and analysis

Each test case produced a file with latency measurements. Additionally, the average CPU utilization on robot2 was measured. For each test case, the number of measurements was stripped to 30000 as to standardize the test cases. Table 7.4 presents the summary of the latency measurements. For each test case, the table gives the mean time of the arrival of the response, as well as the standard deviation.

test no.	mean (μS)	sd (μS)	95th percentile (μS)	CPU load
1	120.19	52.04	140.54	3%
2	118.75	54.56	137.96	3%
3	132.74	64.68	185.63	6%
4	181.73	57.92	205.51	5%
5	178.54	56.42	202.86	4%
6	195.53	64.59	249.55	7%

A brief look at the results allows us to make some interesting observations. First of all, an immediate consequence of retrieving the force measurements while executing MRROC++ increases the response time. For example, the mean time of retrieving the measurement when no other processes were running and KIPER (with raw Ethernet) was used for communication, was about 167ms, while for the similar communication setup, but with MRROC++ running it was 183.73 ms, 178.54 ms or 195.53 ms, depending on where the MRROC++ processes were executed. Moreover, the standard deviation of the response time is higher, which reflects the worse stability of measurements.

The first three test cases used a simple custom protocol to retrieve measurements from the force/torque sensor controller via raw Ethernet. The protocol used fixed length request and response messages; the measurement data were simply placed at a given offset of a response message. The results related to the response time are similar for those test cases. The third test case stands out as the one having the highest CPU load. This result may look counterintuitive in the sense that in the first test case (3% CPU load) EDP, ECP and MP processes were running

on robot2, while in the third case only EDP process was running on robot2. However, having MRROC++ processes running on different machines means they need to communicate over the network, and on the slower machine such as robot2, increased network traffic means increased CPU load. Because under QNX, the network stack is implemented as a user-space process, its CPU consumption may be observed as for any other regular process. In fact, or the test cases number 3 and 6 (which have the same configuration of the location of MRROC++ processes on the machines), the CPU usage for the `io-pkt-v4-hc` process (the network stack) observed was 1.5-2 times larger in comparison to the rest of test cases. We may observe that the first two test cases are similar as far as the response time and CPU load is concerned. Those two test cases differ in that the MP process runs either on `robot2` or on `olin` machine, which, as turns out, does not influence the test results much.

What happens if we replace the simple communication based on fixed messages sent over the Ethernet with KIPER? This is answered by the results of the test cases 4-6. Unavoidably, the CPU consumption increases as well as the mean response time. Interestingly, the fourth test case shows that when EDP, ECP processes are located on robot2, the CPU usage is the smallest. As again, this is caused by the lack of network communication between EDP, ECP and MP processes, but this time (as opposed to the first three test cases) the CPU load is even higher due to the runtime overhead imposed by KIPER.

The nature of the task requires the force control loop to be run with a frequency of 500 Hz. Increasing this frequency does not necessarily improve the quality of the control but may instead lead to increased CPU consumption. This is especially true when KIPER is used to retrieve the measurements, as for each measurement, the request message has to be serialized and the response message has to be deserialized, which costs CPU cycles. On a slower (for the year 2010) machine this is especially noticeable. From the results it can be seen that, in the worst case, the 95th percentile is at about $250\mu S$. This means that the response time for 95% of the requests is below that value, which is clearly sufficient for the given task. Consider test cases 1 and 4. The difference between mean times is about 60 μS, which is relatively large. However with the 95th percentile of the test case number 4 being equal about 205 μS, we may achieve the frequency of about 5000Hz, which is a pretty good result.

8. Summary and conclusions

8.1. Requirements validation

First and foremost, it has been confirmed that it is possible to implement an RPC library that runs on a microcontroller with limited resources (RAM, ROM) and exhibits low-latency characteristics. The library was tested in a real world scenario involving an embedded system (a development board with ARM7 microcontroller) and a PC running a QNX operating system. The development board gathered the measurements from the force/torque sensor and sent them to the PC running the robot control software (MRROC++) upon request. Obtained force/torque measurements were used to carry out the task of following the unknown contour by the robotic manipulator. All of this was achieved using commodity hardware and publicly available software.

One may argue, that for such simple setup using RPC for communication is an overkill. This might be true, but at the same time the proposed solution is more flexible and easier to expand if a system needs to be modified. Certainly RPC is not a „silver bullet" and its use should not be forced wherever possible — for each application one should weigh pros and cons of using RPC instead of some other form of communication.

It is not a surprise that a dedicated protocol performs much better than the designed RPC framework. With times with order of hundred microseconds, the overhead associated with serialization and deserialization of the messages becomes noticeable. Still the obtained results show that the application of RPC to embedded systems, even those involved in real-time robot-control, is feasible.

8.2. Limitations of the solution

8.2.1. Real-time guarantees

By no means the presented solution makes any hard guarantees about its low-latency properties. This is due to many factors, including nondeterminism of underlying connection methods. Yet the fact remains that it is enough to enable

real-time control of the robotic manipulator — missing a deadline once in a while does not degrade the performance of the system.

Although one factor contributing to the overall performance of the system is the time spent on serialization and deserialization of the input arguments and output values, the major cause of the latency is the connection method that is responsible for the actual data exchange between the PC end the embedded device. Fortunately, when connecting the force/torque sensor controller using a separate network adapter, we avoid the latency introduced by intermediary network switches as well as prevent any packet collisions.

8.2.2. Location transparency

Existing solution does not provide any location transparency — the exact address of the server (e.g. IP address and port number in case of UDP) has to be known in order to call remote methods. This may become a problem if the number of servers becomes large. In that case it would be more convenient to refer to the server by name instead of address. It would be possible to implement such „server locator" atop KIPER using broadcast messages to detect servers; however currently, KIPER does not have any explicit means for sending broadcast messages. It would be possible to create an RPC Client with a broadcast address (e.g. `ff:ff:ff:ff:ff:ff` for Ethernet or `255.255.255.0` for UDP) and send a non-reply message asking for device's address, so this might be the way to add this functionality to KIPER.

8.2.3. Reliability

There are no reliability mechanisms built in KIPER; if the server is running, the request packet is delivered at most once, but there is no built-in retransmission mechanism in case of failure. For many message types it is not a problem, e.g. if the reply for the force measurement request fails to arrive within a given time limit, another request would be sent anyway. However, if the time limit expires, the client has no way to tell if the server received the message or not. This is not a problem if the messages are idempotent, one can always call an RPC in a loop until a valid response arrives. The RPC related literature discusses different call semantics providing guarantees as to how many times the remote procedure will be called, e.g. at-least-once, exactly-once. For a simple and lightweight system such as the one considered in this thesis, implementing such semantics would be „over-engineering". Therefore KIPER implements the „at most once" semantic meaning that the remote procedure call will never occur more than once.

8.2.4. Large messages

KIPER is not suited for sending large messages. The documentation of the underlying Protocol Buffers library states: „Protocol buffers are not designed to handle large messages. As a general rule of thumb, if you are dealing in messages larger than a megabyte each, it may be time to consider an alternate strategy" [43]. Additionally the implemented transport mechanisms (UDP, raw Ethernet) do not support sending such large messages.

8.2.5. High bandwidth

KIPER is not suited for high bandwidth, stream transmissions. For instance, sending digital video would be handled much better by TCP/IP or other specific protocols.

8.2.6. Security

KIPER does not have any built-in security mechanisms. This need not be a problem in an isolated, controlled environment.

8.3. Problems

One problem that become apparent after KIPER-C library was finished is that implementing RPC services on the server side is not very convenient. For KIPER-CPP it is just the matter of subclassing the generated service class, while for KIPER-C it is not as straightforward; it involves creating a class-like structure with the base service as the first field writing functions (emulating methods in object-oriented programming) and setting the addresses of those functions in the base structure. This is not intuitive and may be considered as „hack" to circumvent the lack of object-orientation in C programming language.

8.4. Further development

KIPER comes with two transport mechanisms implemented: raw Ethernet and UDP. An interesting idea would be to develop another implementation and verify its suitability for real-time communication. USB (Universal Serial Bus) would be a good choice as it is well suited for low-latency, high frequency communication [31]. The LPC-P2378 development board that was used for testing includes a USB capable microcontroller and there are source code examples that demonstrate USB capabilities of the board. To avoid lengthy device driver development it

would be wise to make the board conform to the HID (Human Interface Device) class [44]. This class of USB devices, while originally meant for devices that require human interaction, may be used by any device as long as it conforms to HID specification. Most modern operating system already contain drivers for HID devices so all communication with HID device is carried out using appropriate system calls.

The current implementation of KIPER-CPP was tested under QNX operating system. In the nearest future the MRROC++ might be completely ported to Linux operating system, and in such event it would be desirable to have a working version of KIPER-CPP for Linux as well. Current implementation should work under Linux with the exception of raw Ethernet transport, which should be rewritten to use „raw sockets" instead of the BPF interface which is missing under Linux.

Given the description of the protocol and a reference implementation it should be fairly easy to port KIPER to a different language. As an example, the implementation in Java using UDP as a transport mechanism could be used to prepare a GUI (Graphical User Interface) for visualising the force/torque sensor reading.

A. Glossary

.proto file A file written in Protocol Buffers definition language containing definition of messages and RPC services.

ADC Analog to Digital Converter. A circuit that converts analog value (voltage) to electrical signals understood by digital circuits

Boost A collection of peer-reviewed C++ libraries, some of which are used in KIPER.

Request packet A packet encapsulating a request message.

Response packet A packet encapsulating a response message.

CMake A multi-platform build system. Used to build KIPER.

ECP Effector Control Process. In MRROC++ the process responsible for the execution of the user's task associated with the given effector.

EDP Effector Driver Process. In MRROC++ the process responsible for controlling the effector's hardware.

Deserialization Transformation of stream of bytes into class or structure

Embedded system A system with tightly integrated software/hardware performing a dedicated function.

Firmware Executable software image for the embedded device

KIPER An RPC system for communication between PC's and constrained embedded devices

KIPER-CPP A component of KIPER designed for PC's. Written in C++

KIPER-C Part of KIPER designed for constrained embedded devices. Written in C.

Microcontroller Small computer on a single integrated circuit.

MP Master Process. A single MRROC++ process responsible for coordination of all the effectors present in the system.

MRROC++ Multi-Robot Research Oriented Controller. An Object-oriented robot programming framework.

Protocol Buffers A serialization format with an interface description language, implemented as an open-source library on which KIPER is based

protobuf-c An implementation of Protocol Buffers in C programming language

ROM Read Only Memory.

Raw Ethernet A communication protocol utilizing Ethernet frame's data field to carry the application protocol, while ommiting higher protocol layers such as IP.

RPC Remote Procedure Call. A communication paradigm allowing to call a procedure of a remote process as if it was a local procedure.

Serialization Transformation of class or structure into stream of bytes

Stub A skeleton, in the form of source files, that allows the client to call remote procedures as a regular procedures

Transport mechanism In the context of this thesis, any communication mechanism allowing the RPC client and RPC server to communicate using the RPC protocol.

UI User Interface process. In MRROC++ the user interface process for the human operator.

Uip A TCP/IP library for embedded devices

UDP A connectionless protocol atop IP.

VSP Virtual Sensor Process. In MRROC++ a process performing a data aggregation on a sensor.

Bibliography

[1] NXP, "LPC23xx Leaflet," 2008. [Online]. Available: http://www.standardics.nxp.com/literature/leaflets/microcontrollers/pdf/lpc23xx.pdf

[2] Stellaris, "LM3S6100 Microcontroller Product Brief," 2009. [Online]. Available: http://www.luminarymicro.com/products/lm3s6100.html

[3] J. Heidemann and R. Govindan, "An overview of embedded sensor networks," *Handbook of Networked and Embedded Control Systems, D. Hristu-Varsakelis and WS Levine, editors, Springer-Verlag*, 2004.

[4] J. Stankovic, T. Abdelzaher, C. Lu, L. Sha, and J. Hou, "Real-time communication and coordination in embedded sensor networks," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1002–1022, 2003.

[5] A. Birrell and B. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.

[6] N. Murphy, "Introduction to CORBA for embedded systems."

[7] M. Ben, "Principles of concurrent and distributed programming," 2006.

[8] L. Peterson and B. Davie, *Computer networks: a systems approach*. Morgan Kaufmann, 2007.

[9] S. Microsystems, "ONC+ Developer's Guide," 2002. [Online]. Available: http://dlc.sun.com/pdf/816-1435/816-1435.pdf

[10] Microsoft, "Microsoft RPC Model: How RPC Works," 2009. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa373935(VS.85).aspx

[11] IBM, "IBM RPC model," 2009. [Online]. Available: http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.progcomm/doc/progcomc/rpc_mod.htm

[12] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2," 1995.

[13] J. White, "A high-level framework for network-based resource sharing," in *Proceedings of the June 7-10, 1976, national computer conference and exposition*. ACM New York, NY, USA, 1976, pp. 561–570.

[14] M. Holliday, J. Houston, and E. Jones, "From sockets and RMI to web services," in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*. ACM New York, NY, USA, 2008, pp. 236–240.

[15] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI," *IEEE Internet computing*, pp. 86–93, 2002.

[16] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. Addison-Wesley Longman, 2005.

[17] A. Tanenbaum and R. van Renesse, "A critique of the remote procedure call paradigm," in *Proc. Euteco'88*, 1988, pp. 775–783.

[18] S. Vinoski and I. Technologies, "RPC under fire," *IEEE Internet Computing*, vol. 9, no. 5, pp. 93–95, 2005.

[19] D. Winer, "XML-RPC Specification," 1999. [Online]. Available: http://www.xmlrpc.com/spec

[20] W3C, "SOAP Version 1.2 Part 1: Messaging Framework," 2007. [Online]. Available: http://www.xmlrpc.com/spec

[21] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," 2007.

[22] Q. Li and C. Yao, *Real-time concepts for embedded systems*. CMP, 2003.

[23] J. Ganssle, "What makes embedded different?" 2009.

[24] S. Dissanaike, P. Wijkman, and M. Wijkman, "Utilizing XML-RPC or SOAP on an embedded system," in *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*, 2004, pp. 438–440.

[25] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: using rpc for interactive development and debugging of wireless embedded networks," in *Proceedings of the 5th international conference on Information processing in sensor networks*. ACM, 2006, p. 423.

[26] J. Axelson, *Serial port complete: programming and circuits for RS-232 and RS-485 links and networks*. lakeview research llc, 1998.

[27] S. Venkateswaran, *Essential Linux device drivers*. Prentice Hall Press Upper Saddle River, NJ, USA, 2008.

[28] E. EIA, "485 (RS-485),-1983," *Standard for Electrical Characteristics of Generators and Receivers for use in a Balanced Digital Multipoint Systems*.

[29] M. Farsi, K. Ratcliff, and M. Barbosa, "An overview of controller area network," *Computing & Control Engineering Journal*, vol. 10, no. 3, pp. 113–120, 1999.

[30] S. Corrigan, "Introduction to the Controller Area Network (CAN)," *Application Report, Texas Instruments*, 2002.

[31] J. Axelson, *USB complete: everything you need to develop custom USB peripherals*. lakeview research llc, 2005.

[32] J. Ganssle and M. Barr, *Embedded systems dictionary*. Cmp, 2003.

[33] E. Spurgeon Charles, *Ethernet: The definitive guide*. O'Reilly & Associates, 2000.

[34] J. Thomesse, "Fieldbus technology in industrial automation," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1073–1101, 2005.

[35] D. Janssen, "Dirk, Buttner, Holger,EtherCAT–the Ethernet Fieldbus?" *PC Control Magazine*, vol. 3, 2003.

[36] C. Zielinski, "The MRROC++ system," in *Robot Motion and Control, 1999. RoMoCo'99. Proceedings of the First Workshop on*, 1999, pp. 147–152.

[37] E. C. T. Committe, "The Embedded C++ specification," 1999. [Online]. Available: http://www.caravan.net/ec2plus/spec.html

[38] Google, "Protocol Buffers Documentation - Developer Guide," 2008. [Online]. Available: http://code.google.com/apis/protocolbuffers/docs/overview.html

[39] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. Winter'93 USENIX Conference*, 1993.

[40] A. Dunkels, "The uIP 1.0 Reference Manual," 2006.

[41] KitWare, "CMake Documentation," 2009. [Online]. Available: http://www.cmake.org/cmake/help/cmake2.6docs.html

[42] Google, "Protocol Buffers Documentation - Language Guide," 2008. [Online]. Available: http://code.google.com/apis/protocolbuffers/docs/proto.html

[43] ——, "Protocol Buffers Documentation - Techniques," 2008. [Online]. Available: http://code.google.com/apis/protocolbuffers/docs/techniques.html

[44] U. I. Forum, "Device Class Definition for Human Interface Devices (HID)," 2001. [Online]. Available: http://www.usb.org/developers/devclass_docs/HID1_11.pdf

[45] P. Laplante, *REAL-TIME SYSTEMS DESIGN & ANALYSIS*. Wiley India Pvt. Ltd., 2006.

[46] C. Zieliński, T. Winiarski, K. Mianowski, A. Rydzewski, and W. Szynkiewicz, "End-effector sensors role in service robots," in *Robot Motion and Control 2007 (LNCiS) Lecture Notes in Control and Information Sciences*, K. Kozłowski, Ed. Springer Verlag London Limited, June, 11–13 2007, pp. 401–413. [Online]. Available: http://robotyka.ia.pw.edu.pl/twiki/pub/Publications/WebHome/ZieRoMoCo2007.pdf

[47] M. Kulesza, "Sprzeg czujnika siły ATI-IA F/T 3084 Gamma z komputerem PC (In Polish)," Master's thesis, WEiTI, Warsaw, 2009. [Online]. Available: http://robotyka.ia.pw.edu.pl/twiki/pub/Publications/WebHome/BScMaciejKulesza.pdf

[48] T. Winiarski and C. Zieliński, "Sterowanie interakcją manipulatora ze środowiskiem - część pierwsza," in *X Krajowa Konferencja Robotyki – Problemy Robotyki*, vol. 2. Oficyna Wydawnicza Politechniki Warszawskiej, 2008, pp. 473–482. [Online]. Available: http://robotyka.ia.pw.edu.pl/twiki/pub/Publications/WebHome/WiniarskiSimzs2008p1.pdf

[49] M. Henning, "The rise and fall of CORBA," 2008.

[50] K. Tarkowski. Morgan Kaufmann, 2010.