Diplomarbeit

Development of an IDL Compiler for Micro-Kernel based Components

Ronald Aigner
<ra3@inf.tu-dresden.de>
Dresden University of Technology
Operating Systems Group

September 25, 2001

Contents

List of Figures

Chapter 1

Introduction

The development of software evolved over the last few years from a procedural and modular approach towards object-oriented and component-based concepts. These concepts allow the optimization of the software development process by replacing monolithic systems with a composition of *components*, which provide appropriate functionality for applications. The clean separation of components makes it very easy to exchange or separately test them. Applications can be extended by modifying single components without the need to change the whole code which uses the component.

A component has to provide an *interface* by which other components can communicate with it. An interface specifies the functions a component is able to execute. The component can be thoroughly tested, because its interface and the supposed behaviour is known. When a component uses the functionality of another component it can also be called the *client* of this particular component.

If the functions to be executed are in a different thread, address space, or even in a different node than the calling component, the parameters of the function have to be made available to the called component. This propagation of data is also called a *message*. The transfer of a message consists of multiple steps. First the data to be transmitted has to be packed into the *message buffer* – *marshal data* –, then the message buffer is transferred from the sender to the receiver and last, the receiver has to unpack the data from the message buffer – *unmarshal data*. The message is transported to the called component by communication mechanisms of the underlying platform. Thus we can say an interface specifies the messages a component will accept. Such an interface can be described in multiple ways, reaching from natural language in a product description to a rather abstract description in a separate, computer readable language – an *interface description language* (IDL).

If a developer intends to use a component, he would have to write communication code, to send a message, which has a format supported by the communication mechanism. This results in writing similarly structured code over and over again. A computer readable description of the interface can be translated by a software tool – the *IDL compiler* – into communication code and can, to some extend, be verified. The IDL compiler simplifies the development of components, because it relieves the developer of writing the communication code by hand. This repetitive task may easily introduce errors to the communication code, which are hard to find. Because the compiler can be trusted to produce code without typing errors and volatile errors, a developer can focus on the development of the component rather than looking for errors, which might have slipped into his communication code.

An IDL compiler simplifies the development of components in a significant way, by "hiding" the communication barrier between components. The compiler can make it almost transparent for the developer, whether he is using the component across a network, address space boundaries or within the same thread. Thus a changing communication interface of the underlying platform has no effect on the generated interface, which is used by other code. This also introduces portability of the components across changing communication interfaces, because the interface is independent of the communication interface.

There are several IDL compilers available for different communication APIs, such as the Mach Interface Generator (MIG) [WT89], Flick [EFF+97], `rpcgen` [Mic88], and many others. All these compilers implement the traditional semantic of procedure calls. This means, a caller uses the functionality of a component by calling one of its functions. This involves a message sent from the caller to the component and a message back to the caller.

This is often more than the caller wants to do. Sometimes the caller simply likes to signal the component to start working on something. The caller is not interested in any outcome of the call. To fit this concept into the remote procedure call (RPC) mechanism, some IDLs have a keyword, such as OMG IDL's `oneway`, to tell the compiler that no answer is expected. Another approach to generalize the RPC mechanisms is to use *message passing* instead and include RPC as a special case implementation. Message passing does not imply a function execution at the component's side, but simply says: here is a message of a specific type for you, you will know what to do. RPC semantic can easily be replicated using message passing. It also allows the client to receive messages from the component and therefore the client has to be ready to receive the message from the component. The mechanism of message passing is well suited to provide a more general approach to communication between components, which only implement signaling messages.

The IDL compiler of this work is embedded in the Dresden Real-Time Operating System (DROPS) project, which focuses on building components for real-time applications [BBH+98]. This work includes implementing the functionality of a monolithic operation system as separate servers running on top of the L4 μ-kernel [HBB$^+$98, GJP$^+$00]. These servers, which provide operation system functionality to application, I call *system components*. Because these system components use each other's functionality, they have to communicate with each other across address space boundaries. The generation of the necessary communication code is a typical application area for an IDL compiler. It not only eases the separation of the system components but also provides a level of portability to these components. Because the communication code is represented by the interface description, a component can be moved to another platform by recompiling the interface description.

When writing applications, more than one system component can be used frequently. This entails a lot of communication between components, which means a lot of copy operations. Because copy operations are expensive it is a goal of the IDL compiler to minimize copy operations in the generated code. Some of the parameters can be excluded from packing, for instance if it is a constant value, known at compile time of the code. It is also possible to optimize the performance of copy operations when using appropriate copy mechanisms for different data types.

One strategy of the L4 μ-kernel is to "tolerate [a concept] inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality." [Lie96]. E.g. does L4 leave "memory management and paging outside the μ-kernel; only the grant, map and flush operation are retained inside the kernel." [Lie96]. These operations can be initiated by sending a specially formatted inter-process communication (IPC) message.

In order to build communication code for system components, it is necessary for the IDL compiler to know about these special semantics and to provide some way for developers to use them. The compiler also needs to know how to make the μ-kernel interpreted normal data as specially formatted data – *kernel data* – which influences the semantics of an IPC. Such kernel data is e.g. a flexpage, which describes a memory region. It can be used with a specially formatted IPC to grant or map the memory region into another address space.

This work continues early research of the DICE project [Aig01]. DICE supports the simple client/server structure of communication. It knows about the flexpage data type and optimizes some of the messages using indirect strings. This work concentrates on the implementation of message passing, special kernel semantics and enhanced optimization into DICE – the IDL compiler of the DROPS project.

Structure of this Work

The paper contains the following chapters. Chapter 2 – Related Work – introduces the reasons to implement an IDL compiler for μ-kernels at all. It also introduces some of the topics and related work concerning the paper. Chapter 3 discusses the design decisions made when incorporating the ideas of message passing and kernel-data structures as well as optimization into the IDL compiler. Following some details on implementation in Chapter 4, such as the IDL with included support of the mentioned ideas. Chapter 5 contains the measurements of the generated communication code to show the overhead of the communication across address space boundaries. There is a comparisons to hand-written code as well as to code produced by other IDL compilers, such as Flick. Which kind of future work has to be done and a summary and conclusion of the work done up to now can be found in Chapter 6 and 7, respectively.

Acknowledgements

I like to thank everybody, who supported my work on this IDL compiler. Especially my coaches Prof. Dr. H. Härtig and Lars Reuther. I like to thank Volkmar Uhlig for the endless discussion and his advise. I also like to thank the staff of the Operating Systems Group at the Dresden University of Technology and the System Architecture Group at the University of Karlsruhe. Thank you for the discussions and incitations.

I also like to thank the users of DICE. Without their adventurous spirit the compiler would lack important parts of its functionality.

I like to thank my parent for supporting me all the time. Last but certainly not least I like to thank my wife Zilly. Thank you for your patience and love.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

Date: September 25, 2001

Author: ____________________________

Ronald Aigner

Chapter 2

Related Work

This work has the title “IDL compiler for μ-kernels.” Thus we have to find out what is so special about an IDL compiler designed for *μ-kernels*. Therefore, I looked at several other IDL compilers which produce code for μ-kernels. I also looked at IDL compilers tailored to low-level tasks, such as device driver support.

2.1 Terminology

2.1.1 Communication Scenarios

To simplify the understanding of the following sections, a short introduction of some communication scenarios is necessary. Figure 2.1 shows three kinds of scenarios. The classic, synchronous RPC scenario is shown under `a)`. It involves a sender (or caller), which sends a message to the receiver – another component –, which executes the called function and returns a message. Until the sender receives the reply its execution is blocked.

The configuration under `b)` represents an asynchronous RPC. The caller sends a message to the other component, but instead of blocking, continues its execution. Later it enters a wait state and blocks until it receives the reply from the called component. The two components may execute in parallel.

The picture under `c)` shows a simple send message scenario. The caller is not interested in any outcome and thus does not wait for a reply message from the called component.

2.1.2 Compiler Chain

Whenever an IDL file is translated into communication code and the generated code is integrated and translated by another compiler, the various compilers form a compiler chain. This compiler chain is described in Figure 2.2. At first the IDL file is translated using the IDL compiler into the target language of the IDL compiler. The generated communication code is then

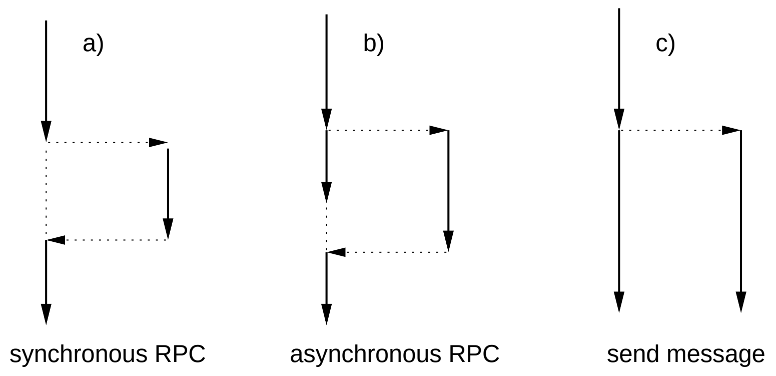

Figure 2.1: Communication Scenarios

translated by the target language compiler into its target language, which is usually executable code. Each compiler consists of multiple stages. These include at least a reading and a writing stage – see Figure 2.3. The reading stage usually performs some validation and eliminates unnecessary input, such as comments. This is also called *pre-processing*. When writing the target code, each compiler does some optimization to generate reasonable target code.

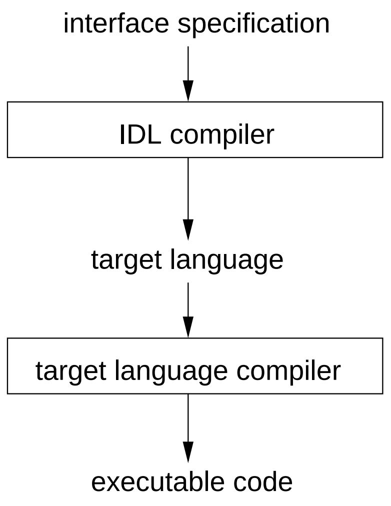

Figure 2.2: An Example Compiler Chain

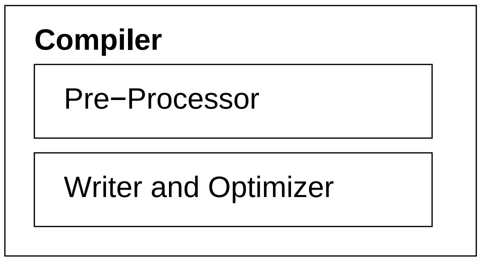

Figure 2.3: The minimal Parts of a Compiler

2.2 IDL Compilers

2.2.1 Mach Interface Generator - MIG

The Mach Interface Generator is provided to “produce a remote procedure call interface to IPC message passing” [WT89] for communication in a multi-tasking system. The interface definition language, MIG uses, is specific to MIG. It defines an interface of a server in much the same way as other IDLs do. An interface description specifies a component with functions and some user-defined types. The close interaction with the Mach kernel is based on the fact that “the first unspecified parameter in any operation statement is assumed be the `RequestPort` unless a `RequestPort` parameter was already specified.” [DJT89]. The MIG interface description language has, besides other types, a special type for Mach ports (`RequestPort`), the Mach communication identifier.

The port concept entails a lot of information specific to Mach, which has to be represented in the IDL. This restricts MIG to be used with Mach only. No port to other kernels, which do not follow the Mach specification, or even μ-kernels, is possible. Communication on Mach has another implicit semantic: because Mach buffers messages inside the kernel, communication, which interacts directly with the kernel, is always asynchronous. Of course exist algorithms, which can simulate synchronous behaviour using asynchronous communication, but the natural communication mechanism of Mach is asynchronous. There also exists a type representing access right for ports – the basic security mechanism of Mach –, which MIG must understand and translate appropriately.

This deep interaction shows that the Mach ports have such a special semantic that they cannot be mapped straight forward onto simple data types. Instead, these features with special semantics were integrated into the IDL and its compiler to provide simple access to these features for the users of the interface.

Nevertheless, the specification of a communication partner in the IDL is arguable. If it is done in the same way as with MIG, it is hard to port the compiler and the IDL to another platform using different specifications. But the integration of the communication partner into the IDL showed practical for Mach, because the transfer of rights is handled using ports and port rights and thus it is essential to implement system components.

2.2.2 Devil

To find abstractions for kernel specific data types I looked at Devil, a description language for hardware, especially for devices [MRC+00, RM01, MM01]. Until now we used the term communication as exchange of message between two software components. Because Devil is a description language for devices the terminus communication describes the access of a driver to the hardware device. Thus one of the components is the device driver and the other one the device itself. A device description in Devil can be translated using the *taz* compiler into C code, which accesses the device. This C code can be understood as the communication code.

Because the description of hardware devices is different to a software component description, a new IDL was created. The compiler has to generate the "communication code" unique for the specified device. To be able to create the correct communication code the compiler needs to know how the component can be accessed. This is usually done reading or writing bits or bytes from registers of the device or its memory.

Therefore a description of the registers and memory available on the device has to be included in the interface description. Also does the description of the functional interface has to contain the number and sequence of manipulated registers and memory in order to execute the expected functionality. The IDL compiler generates "communication code", which performs the specified manipulations.

Devil Data Types

Devil's most basic data type is a **port**. "A port hides the fact that, depending on how the device is mapped, it can be operated via either I/O or memory operations." [MRC+00, Sc. 2]. All ports are addressed relative to a base address, which is specified in the device's constructor. The specification also contains valid offsets from the base address to enable verification of ports. When using a port in register definition, the port is specified using the base address and a constant or range offset. E.g. `register reg = base @ 0`, which specifies the register `reg` to use the port with the offset 0 from the base address.

The next level of abstraction is the **register**. "A register defines the granularity of interaction with a device. Thus the size of a register (number of bits) must be explicitly specified." [MM01, pg. 7]. There are usually two ports associated with one register. One port to read from and one to write to. Or only one port if the read and write operation share the same port. To specify the bits of the port, which are interesting for the access, a register can be declared with a bit mask. The bit mask is a string, which has as many characters as the port has bits. The character '.' denotes a used bit. The characters '0' or '1' denote an unused bit, which is set to the zero or one respectively if written. '*' denotes an unused and undefined bit. This way every single bit can be defined as used, not used or preset when reading or writing it.

The top level of abstraction are device **variables**. "In order to minimize the number of I/O operations required for communicating with a device, hardware designers often group several independent values into a single register. Accessing these values requires bitwise operations." [MM01, pg. 7] Therefore Devil abstracts values as device variable, which are a sequence of register bits. These variables are of simple types, like boolean, signed or unsigned integers. When accessing the device, this is done by using these variables.

These are not all of the features of Devil, but the most interesting abstractions of basic data types. There exists no abstractions for interrupts in Devil.

2.2.3 Flick

When talking about IDL compilers for the L4 IPC interface the Flick [EFF$^+$97] compiler has to be mentioned. The concepts behind Flick and its L4 adaptation by Volkmar Uhlig are explained in [Aig01] and [Uhl99].

Flick generates communication stubs consisting of macros, which implement the marshaling and unmarshaling code. Thus most of the adaptation could be done by changing these macros. The manipulation of the marshaling sequence of the parameters caused problems, because the code of Flick is hard to adapt. This flaw was overdone by selectively marshaling parameters and executing the marshaling code multiple times. Flick generates communication code, which is not as efficient as it could be if using a properly implemented IDL compiler.

2.2.4 Summary

The Devil example shows, that IDL compilers operating at low programming levels, such as device programming or system service development, have to support low-level data types to be able to provide a support to the developer. These data types have to be presented in the IDL in a way, which makes the easy to identify, but they have to be translated into a target language representation, which makes them usable across platforms.

When any kernel or communication specific implementation detail should be hidden from the user, the integration into the target system must be invisible to the user. The compiler must present an abstract view at the usage of the kernel internal data structures. The MIG compiler is an example of too much integration into the target platform. It includes too much of Mach to enable any migration to another platform.

These two examples show that the representation of platform specific semantics in the IDL is necessary to allow the user of the IDL to utilize these semantics. But the representation should be as abstract as possible to be portable to other platforms. The Flick adaptation showed that a tool, which has to be adapted to support new target platforms, has to be designed and written to allow an easy and fast adaptation.

2.3 Message Passing Introduction

Former IDL compilers almost always implemented procedure call semantics as depicted left-most in Figure 2.1. These IDL compiler do mostly generate client/server scenarios, where the component (or server) receives messages from clients, calls the appropriate functions and eventually sends a reply with the return values. This strict synchronous communication coupled with the complex server-loop is a very confined way to implement the communication between two components. Sometimes the reply is superfluous and sometime the server-loop only waits for a single message. Thus it is necessary to think about a more general approach to communication than the remote procedure call (RPC). This general approach is *message passing.*

When using the classical client/server approach, only the synchronous RPC communication semantic (as introduced in Figure 2.1) can be mapped to this approach. The other two semantics can also be implemented using the client/server approach, but without justifiable effort. If using message passing, the asynchronous RPC as well as simple send message can be implemented easily. Of course does message passing support the synchronous RPC as well.

Message passing is widely used in multi-processor facilities. It differs from the RPC semantics by not depending on synchronous processing of the procedure. A single message initiates a procedure. The caller continues its processing after sending this message. Thus the developer might rather implement a *flow of control following data* then a *flow of control following instructions.*

Strictly speaking, a Remote Procedure Call is a special implementation of message passing. First a message starting the procedure is sent to the component and eventually a reply message arrives. Thus a RPC can be regarded as a pair of messages, and is fully contained in message passing.

When looking at existing IDL compilers they incorporated the ideas of message passing to some extend. The CORBA IDL allows message passing by using a keyword, which signals the compiler to produce a message from the caller to the component without a return message. This keyword allows only message passing from clients to the component, but not back from the component.

To include the concepts of message passing into an IDL compiler provides its user with more possibilities to implement components. It will make it more convenient to use simple messages. This feature is another step towards making the development of multi-server operating systems easier.

2.4 Language Integration

Communication between different processes or threads can be implemented in different ways. The developer may use the communication primitives of the underlying platform directly. He also may specify an interface and let an IDL compiler translate it into communication code. Or the programming language or other tools provide other ways to write a communication invocation.

2.4.1 Message Passing Interface Library

The Message Passing Interface (MPI) Library [JT99, JT00] is a library which provides functions, that allow the easy implementation of communication between processors. The MIP library is mainly used in distributed computing environments. The MPI functions send messages to different processors and receive messages from those processors. These functions have parameters specifying the processor(s) to send to or accept messages from, the type of the data (integer, float, etc.), the starting address of the data, which might be the starting address of a array of data values, the size of the data (e.g. the array's size or 1 if it is a single value), and some other status information, e.g. the number representing the position of the message within a series of messages. There are also many other functions for communication (e.g. a broadcast call) which make it easy for developers of distributed programs to use communication code.

The advantages of the MPI library are, that the use of these functions directly inside the target language, instead of specifying an interface description. The developer does not have to mess around with the details of the communication. Because the basic mechanisms of message passing – send and receive – are provided, a simple message passing application as well as synchronous function calls can be implemented (the latter is not an issue of distributed computing, but possible).

2.4.2 Rendezvous Concept in Ada

Ada implements multitasking using *tasks*. A task is an active object, which implements functions. In a single-task application functions are called just like in C or C++ – by specifying the function to be executed and its parameters. In a multi-task application the functions are called just the same. It is transparent to the developer whether the function is in another task or in the same. A task definition in Ada contains statements to execute functions, if they are called from another task. As described in [Bur01], the main body of the task, can contains a `accept` statement followed by the function declaration. This statement specifies, that the task will accept requests from other tasks to execute this function.

Because request may appear in any order, Ada has the keyword `select`, which allows to specify multiple functions, which can be received in parallel. Not all at once, but any one of the specified functions can be received. If requests for functions must be executed in a defined sequence, the `accept` statements are specified in the appropriate order, without a `select` statement. To allow the arrival of a request only under a certain condition, *guards* can be associated with an `accept` statement. Only if the expression, specified for the guard, evaluates to true the request will be accepted.

2.4.3 Synchronous C++

Synchronous C++ is an extension to the C++ programming language which allows the definition of active objects besides the already existing passive objects. "An active object may contain an internal activity, which runs in parallel with the activities of other active objects." [Pet98, p.65]. The Synchronous C++ extension implements the rendezvous concept of Ada into C++ using similar keywords and source code layout. The difference is that only active objects may send or accept messages, but it is not possible to integrate these messages into the normal source code of existing passive objects.

The declaration of an active object in Figure 2.4 includes the definition of three functions `a`, `b` and `c`. The function with the `@` in front of it is the *body* of the active object, which carries out the internal activity. The body includes an infinite loop (`for(;;)`), which includes a select statement similar to the select statement of Ada. The delimiter between the choices is the "`||`" string. The accept statement has as parameter, which is the function to be executed in case respective message is received. The above code can be expressed using BNF like this:

$([\mathtt{a},\mathtt{b}]\mathtt{c})^{+}$.

Similar to Ada a message is send when the respective function is called. To send the message `a` to the instance `instA` of the object `ActC1`, the call `instA.a()` would be sufficient.

```
active class ActC1 {
  public:
    virtual void a() { printf("a\n");}
    virtual void b() { printf("b\n");}
    virtual void c() { }
  private:
    @ActC1() {
      for (;;) {
        select {
          accept a;
        ||
          accept b;
        }
        accept c;
      }
    }
};
```

Figure 2.4: Example of Active Class in Synchronous C++

2.4.4 Signals and Slots in the Qt Library

"Qt is a cross-platform C++ GUI application framework. It provides application developers with all the functionality needed to build state-of-the-art graphical user interfaces." [Tro00a]. When using the Qt libraries a developer might also use the integrated communication facility of Qt. Because Qt is designed to develop user interfaces it provides constructs to exchange messages between different interface objects. Qt introduced two concepts into the programming language of Qt (C++): *signals* and *slots*. Implementations of these concepts can be associated with an interface object [Tro00b].

Signals represent messages, which the object (or an instance of it) may send. Slots represent points, where an object can receive messages. When a developer writes objects, which should communicate with each other using the signals and slots, he has to connect the signal of the one object with the appropriate slot of another object. This is done using macros and special reserved words.

A pre-processor used with Qt, called Meta Object Compiler – *moc*, replaces the used macros and reserved words with function declarations and calls. The message handling is done using meta object, which contain the references to the appropriate slots for a signal. Whenever a signal is released by calling the signal function, this call is redirected to the meta object, which checks the types of the message and directs the message to the appropriate slot. This mechanism only works inside one thread, because the signal-slot mechanism is a simple redirection of a function call.

2.4.5 Summary

Beside the advantage of the target language integration the disadvantage of the MPI library is the precompiled interface description, which allows to send only basic data types in a uniform manner. It is not a general integration of function calls across tasks as Ada provides, nor a general communication interface provided by interface description languages. It is also restricted to the communication between processors. There is no way to communication with different tasks on the same processor or between threads within the same address space.

Communication across task boundaries is a fundamental concept of Ada. This integration frees the user from writing bare communication code. It seems very convenient for a developer not to care about the communication implementation and simply write a language statement into the code where he wants to receive a message and which message it shall be. An stand-alone IDL compiler will never reach this level of integration, except it is integrated into the target language compiler and extends the target language.

The concepts of Synchronous C++ show a way to incorporate the concepts of Ada into the C++ programming language. Synchronous C++ does provide a concept to relieve the developer from implementing basic communication code.

Even though the Qt concepts to exchange messages are designed to be used inside one thread, it shows ways to integrate communication mechanisms into a target language too. Qt, as well as Synchronous C++, can accomplish these tasks only by introducing new keywords into the target language.

2.5 L4 Communication Basics

When writing an IDL compiler for L4 we have to be aware of the L4 communication mechanisms. L4 provides threads running on top of it with a communication API, which allows the threads to exchange messages. To exchange data between address spaces, data usually is copied from the sender's address space into the receiver's address space. The kernel provides different mechanisms to do so. The sender might specify a message buffer, which contains data to be copied. The kernel will copy this data from the sender's message buffer (address space) into the receiver's message buffer (address space). This kind of transfer is called direct IPC. To copy the data fast, the kernel establishes a temporary memory mapping of the receiver's buffer into the sender's address space. Thus the costs for this direct IPC include the establishing of the temporary mapping plus the costs for one copy operation of the data.

The kernel skips the temporary memory mapping and copy operation if the amount of data to be transferred fits into the registers the kernel can spare during the IPC. These are two registers for L4 version 2 and three for L4 version X.0. Data which fits into these registers is transferred directly through the registers. This kind of communication is also called short IPC and should always be preferred.

Another mechanism is the usage of “indirect strings”, where the message buffers specify only a memory location and the amount of data to be copied by the kernel. The intention of indirect strings is to avoid unnecessary copy operations in and out of the message buffer.

To communicate with different threads, several kinds of IPCs are provided in the kernels API. These include a simple send operation from one thread to another, a receive operation to accept a message from a specified sender, a wait operation to accept a message from any thread. For convenience and performance the send and receive operations can be combined. The combination of a send-and-receive operation is called call. This call is an atomic operation. The respective send-and-wait operation is called reply and wait, because the send operation is to a specific thread and the wait operation accepts any thread. This can be used in a client/server scenario for the replies of the server to the clients. This operation is atomic as well.

Another message with special semantic is the flexpage IPC. The message contains descriptions of memory regions and perhaps other data. The kernel uses the information in the message buffer to either map, grant or flush a memory region to or from the receiver’s address space [Lie96]. When sending a flexpage the sender transfers (or revokes) the rights to access the memory region to the receiver. An IDL compiler has to generate appropriate code for these special messages.

2.6 Optimization

Another goal when developing system components for a multi-server operating system, is to introduce as little communication overhead as possible. To generate fast communication code, the IDL compiler has to know the communication primitives of the underlying platform and the specialties of the compiler which will translate the target code (see Figure 2.2).

Even the best IDL compiler will not be used if the generated code is by magnitude slower than hand-written code. A developer would rather take the burden of writing the code himself and be sure it is fast, than letting the compiler generate the code and know that the code will be too slow to be of real help. Thus, the IDL compiler has to minimize the communication overhead by generating fast marshaling and unmarshaling code. (Communication also involves the exchange of the IPCs, but the IDL compiler has only indirect influence on that.)

In [Aig01], I researched different optimization strategies and found out, that for different data types different copy mechanisms minimized the copy overhead. One example is the marshaling of a variable sized buffer using Flick [EFF$^+$97] compared to a simple `memcpy` operation. Depending on the size of the array the `memcpy` operation is five to twenty times faster than the element by element copy operation used by Flick. Constructed types may also be copied using `memcpy` instead of marshaling every single member. This example shows, that optimization of the copy operation can make the marshaling stubs faster.

2.6.1 IDL4

Another IDL compiler for the L4 μ-kernel is IDL^4 [L4K00]. It has been implemented at the University of Karlsruhe as work towards a replacement for Flick. The project started using the same code base as DICE did, but diverged into a special case implementation for L4. Several optimization strategies have been tested using IDL^4.

One of the optimizations tested, is the direct stack transfer ([HLP$^+$00]). When calling a function in C (and the C file is translated using a compiler like `gcc`) the stack has a special format, which contains the values of the function's parameters and other information, such as return address etc. If the portion of the stack, containing the parameter values is copied to another stack, the same function could start working on the other stack as it would have with the original stack. This idea is used to call the functions of a component. The client imitates the component's function and copies the relevant portion of its stack to the stack of the component. The component can now jump to the start of the function and start working. Using this mechanism, IDL^4 could achieve a performance improvement by factor 3 in comparison with Flick generated stubs [HLP$^+$00, pg. 6]. This example shows that clever optimization strategies provide the developer with valuable tools.

2.6.2 Summary

The results of [Aig01] and IDL^4 show, that several strategies can be used to minimize communication overhead. The most important one is copy minimization. The marshaling and unmarshaling code has to copy as little as possible. Another strategy is to deploy the features of the μ-kernel, such as indirect strings and short IPC.

Chapter 3

Design

The Design Chapter discusses different approaches to the main goals of this work, which are:

1. implement support for the message passing semantic into the IDL compiler,
2. integrate support for special semantics of the underlying platform, which is L4, and
3. use enhanced optimization strategies in the IDL compiler to generate fast communication code.

These three goals have to be met by appropriate representations in the IDL and the target language.

3.1 Message Passing

Since message passing generalizes the concept of transmitting data compared to the RPC approach, the changes for message passing were a complex task. There are several aspects to message passing, which have to be considered. Firstly: How does the developer specify message passing in the IDL. Secondly: Which code must be generated. And thirdly: How can the generated code be integrated into the application.

3.1.1 Communication between System Components

To find an appropriate representation for message passing in the IDL and target language, we have to examine relevant application areas. From experiences, which we collected in the DROPS project, we know that there are numerous application areas for message passing. These experiences revealed several, typical communication scenarios between system components. I will introduce these scenarios and name the necessary communication primitives and their respective message passing constructs.

Communication Scenarios

Communication between system components can also be based on other mechanisms than the exchange of message, e.g. shared memory. This Section concentrates on different message based scenarios. These scenarios are:

1. The sending component sends a simple message to the other component.(Figure 3.1)

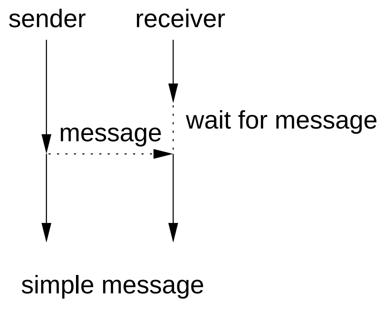

Figure 3.1: A Simple Message

2. The sending component calls a function at the other component, just like an RPC. The sender initiates a send operation with an immediately following wait. The receiver has to wait for the message, execute the function and eventually sends a reply. (Figure 3.2)

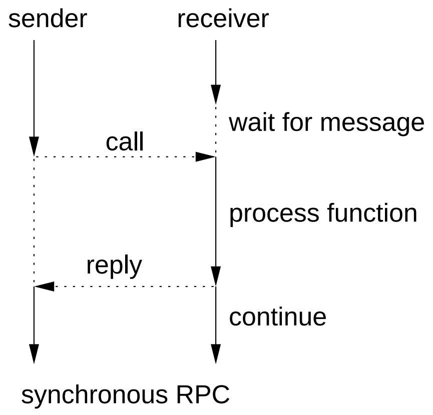

Figure 3.2: Synchronous RPC

3. After the receiver send the reply, it returns to wait for a new message. The message contains a function identifier, which the receiver uses to

determine which function to call. This is a classic client/server scenario. This scenario is different from the second in waiting for multiple function, whereas the former (single synchronous RPC) scenario waits only for one function. (Figure 3.3)

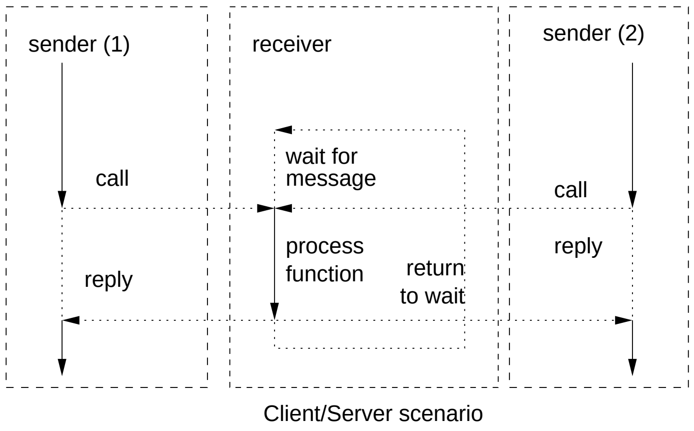

Figure 3.3: An Synchronous RPC with Receive Loop (Client/Server Semantic)

4. The sender executes a function asynchronously, meaning it sends the receiver a message, which initiates the execution of a function at the receiver's side. The sender does not immediately wait for an reply, but does continue its execution and eventually waits for the reply. (Figure 3.4)

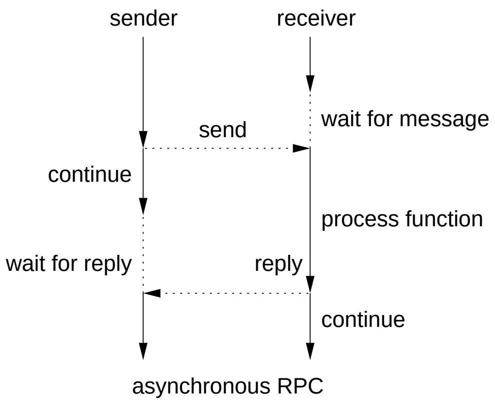

Figure 3.4: Asynchronous RPC

5. The sender executes a function asynchronously and specifies a position

to deposit the reply at – the *reply point*. The receiver has to send the reply to the reply point, after it finished the execution of the function. (Figure 3.5)

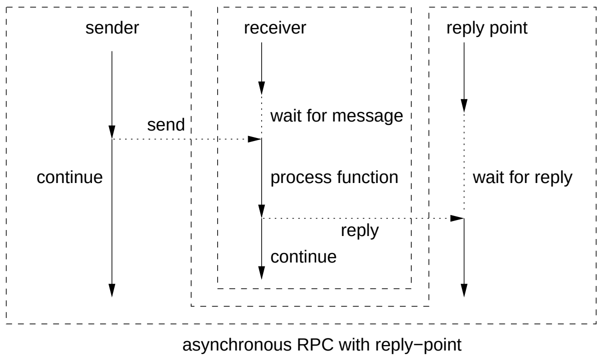

Figure 3.5: An Asynchronous RPC with Reply sent to another Thread

6. The sender calls a function , but receives the reply from a different thread, than the original receiver. Using this mechanism, the receiver may dispatch the function to worker threads and continue to wait for requests. (Figure 3.6)

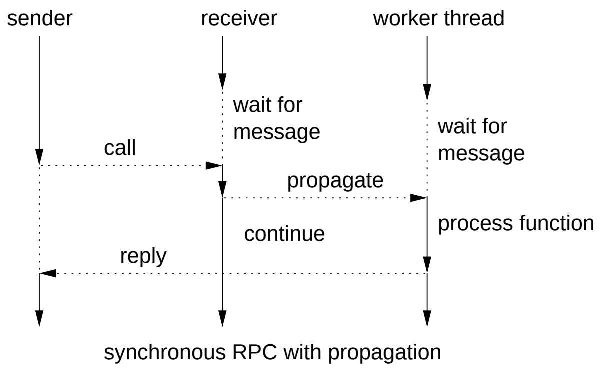

Figure 3.6: A synchronous RPC with Reply from another Thread

7. This actually is a mixture of some of the above scenarios. The sender asynchronously calls a function, which is propagated by the receiver to the worker thread. The worker thread eventually send the reply to a reply point. The receiver may immediately return into a wait state to receive new requests. (Figure 3.7)

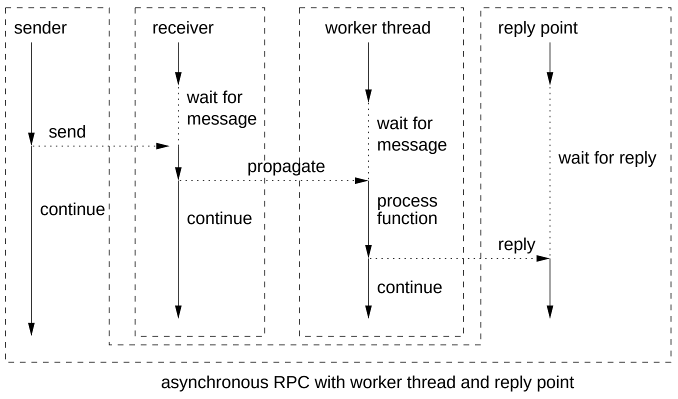

Figure 3.7: A asynchronous RPC with Worker-Thread and Reply-Point

Communication Primitives

To implement these scenarios, different communication primitives are necessary. In the following the primitives corresponding to the communication scenarios are listed.

1. To send and receive a simple message only send and receive primitives are necessary. (see Figure 3.1)

2. To call a function remotely the sender needs an atomic operation to send a message and receive the reply. At the receivers side only a receive and a send primitive are needed. (see Figure 3.2)

3. To implement a client/server semantic we need the same primitives as for the second scenario. But this time the receiver uses an atomic reply-and-wait primitive, because the server will reply to a client and is ready for the next request immediately. (see Figure 3.3)

4. To call a function asynchronously, the sender only needs a send primitive and a receive primitive. The receiver can be implemented using the respective part of the second or third scenario. (see Figure 3.4)

5. For a reply to a *reply point* no other primitives are necessary. The receiver must be able to send the reply to the correct reply point. (see Figure 3.5)

6. To receive the reply from a different thread than the original request came, the communication primitives must be able to accept such a reply. Because the sender often waits for a reply from the thread, where it send the original message to, the replying thread has to fake its identity to send the reply on behalf of the original receiver. This

identity fake is only necessary if the sender sends and receives with an atomic operation. (see Figure 3.6)

7. Because this scenarios is a combination of some of the above scenarios, no additional communication primitives are necessary. The sender needs a simple send primitive. The receiver and the worker thread need a receive and send primitive. The reply-point only needs a receive primitive. (see Figure 3.7)

The above scenarios only need some communication primitives, such as:

- a send operation,
- a receive operation,
- an atomic send and receive operation (call),
- an atomic reply and receive operation, and
- an atomic send and receive operation, which accepts the reply from another thread.

The IDL compiler has to generate functions which implement these primitives.

Communication Context

The last but one scenario can be modified in the following way. Instead of sending the reply from the worker thread directly to the sender, the worker thread replies to the receiver. The reply includes the results of the job and a notification which job has been finished. The receiver has to associate the job with the corresponding client, since several other request might have arrived in the meantime. The receiver needs some sort of communication context. After it found the correct client it sends the reply to that client.

This scenario does not require any new communication primitives. But the IDL Compiler can be used to generate code, which eases the management inside this scenario. The compiler may provide some sort of communication context and management functions for it.

3.2 Design Options

Now that we have discussed the different communication primitives which have to be supported, there are several opinions on how to implement the IDL compiler in respect to the usability. The most obvious and formerly followed methodology is the implementation as a separate tool. This tool compiles an interface specification file into communication code of the target language. Another method, which is much more user-friendly is the integration of communication into the target language, just like Ada or Synchronous C++ do it.

As shown in Section 2.4.2, Ada provides language constructs to call function of another task. The design is bare of every kind of communication specific data, such as identifiers of the communication partner or similar. The language provides everything necessary to communicate between different tasks. When trying to implement this into our target language – C – we would have to make adaptation of a target language compiler (e.g. `gcc`) or a pre-processor would have to replace the communication keywords with code of the target language (see Figure 2.2).

An extension of an existing compiler, such as `gcc`, would require the developer doing the adaptation to get acquainted with the inner workings of the compiler and integrate all of the "new" concepts of message passing and communication into the compiler. This requires more time and work than this Diploma could provide. To implement a pre-processor is the next logical step when having successfully implemented a stand-alone tool. The generated code can easily be used by the target language compiler. Compared to a stand-alone IDL compiler a pre-processor has to replace code in existing files rather than creating new files. This allows only a narrow view onto the whole communication relationship, whereas a stand-alone tool always knows both parts of the communication.

Because of these facts I decided to stay with a stand-alone compiler, which translates an IDL file into the respective communication functions. As described above, there are several communication primitives, which have to be represented by functions in the target language.

3.2.1 Message Passing in the Target Language

Because the current target language of DICE is C, it seems necessary to talk about design decisions regarding C. When using message passing in C we have to check which functions need to be created to cover the whole spectrum of possibilities. First we start by looking at a simple send operation from a client to the component, and explain its progress using an example.

A function `F1` takes the parameters `p1` and `p2` as arguments, whereof each is send to the component. `F` is a message passing function, which only sends to the component, but does not expect a reply. The C send function is called `send_F1`, which takes the arguments `p1` and `p2`. This function has to marshal the arguments into the message buffer and invokes a send operation. How the send function knows where to reach the component, is a platform specific implementation detail and is mentioned in the Implementation Chapter. The component signals the underlying system, that it is ready to receive the message from the client. This can be done using a receive function, which is called `recv_F1`. The receive function will execute the receive operation of the underlying communication mechanisms and unmarshal the arguments of the function from the message buffer. There have to be two kinds of receive functions: one to receive from a specified source and one to receive from any source. We will call the latter a wait function, which is called in C `wait_F1`.

Further more, there is an additional function `F2`, which is an RPC style function and has the parameter `p3`, which is send to the component and the parameters `p4` and `p5`, which are expected as result values from the component. When the client initiates the RPC function it uses the C function `call_F2`, which takes the parameters to send to the component as values and the parameters received from the component as references, but only `p3` is marshaled and send to the component. The component has to wait for the message by using one of the `recv_F2` or `wait_F2` function. When the message arrives the parameter `p3` is unmarshaled and the appropriate component implementation of the function is called. It returns with values for `p4` and `p5`. These two parameters are marshaled and send to the client using a `reply_F2` function, which takes only `p4` and `p5` as parameters. If the component wants to wait for a new message from a client, the reply function would also contain code to wait for the next message and be called `reply_F2_and_wait`.

Assuming, we want to wait for both functions, the compiler has to generate a function `wait_F1_or_F2`. This function needs to decide which function is requested, and has to unmarshal the respective parameters. In C the function would have the parameters `p1`, `p2` and `p3` (by reference). But only a subset of the specified parameters is used. If more functions are added to the interface the compiler would have to generate for all permutations of the functions receive and wait functions.

Because this strategy increases the target code size tremendously the more functions an interface has, we use another approach. When analyzing the implementation of these function, all of them have the same job: they receive a message, check which function this message is for, and unmarshal the respective parameters. Only the last step is specific to the separate IDL functions. We divide the `recv_F1_or_F2` function into a function which can receive any message called `recv_any`, and separate unmarshaling functions `unmarshal_F1` and `unmarshal_F2`. The disadvantage of this strategy is, that `recv_any` will also accept message for function `F3`, `F4` and `F5`. The decision, whether a message will be accepted and which unmarshaling function to use, is left with the developer. For an RPC function the C functions `unmarshal_F2` and `reply_F2_wait_any` are generated for the component's side.

3.2.2 Client/Server Scenario

The implementation of a server loop using the above functions is illustrated in Figure 3.8.

```
opcode = wait_any(&sender, &buffer);
while (true) {
    switch (opcode)
    {
        case opcode_P:
            unmarshal_P(&buffer, &t1, &t2);
            P_server(t1, t2);
            opcode = wait_any(&sender, &buffer);
            break;
        case opcode_S:
            unmarsahl_S(&buffer, &t3, &t4);
            ret = S_server(t3, t4, &t5);
            opcode = reply_S_wait_any(ret, t5,
                &sender, &buffer);
            break;
    }
}
```

Figure 3.8: Example of a Server Loop using Message Passing functions

3.2.3 Message Passing in the IDL

When using an IDL file to specify interfaces using message passing we need an IDL which enables the user to group the functions. The groups contain functions which only send, or functions which only receive, and those which do both. In case the IDL provides a mechanism to associate keywords with every function this is an easy task. Such a mechanism can be the usage of *attributes*. An attribute is a keyword which provides additional information about a language construct it is associated with. The attributes in the DCE IDL have to be enclosed by brackets (`[<attribute>]`). To mark a function as a function, which only sends messages to the component, it may receive a send attribute.

If the IDL provides no possibility to add information on a per-function manner, as would be possible with attributes, there must be some other way to mark the different functions. This could be done using different sections in the IDL file, e.g. different interfaces or sub-interfaces for each kind of the message passing types. Or the IDL may provide keywords identifying the sections inside an interface, similar to the `private`, `protected` and `public` keywords in C++.

The changes, which have to be made to the IDL compiler, are very simple when using the first approach, because it only has to understand two or three more attributes for the functions of an interface. When implementing the latter strategy the changes are somewhat more complicated. The IDL compiler would have to understand a different grammar, because the file is structured differently. Both variants would have to adapt their back-end to create correct message passing stubs.

3.3 Data Semantics

Sending a message also means to transfer data. This Section describes the various semantics associated with data. It explains the user-required semantics and how they can be integrated into the IDL.

3.3.1 Copy Semantic

From procedural programming languages we know about two major ways to transfer the values of parameters of a function. The first is known as “call by value”. It copies the values of the parameters into the scope of the function. Changes to these values are only visible inside this scope. They are lost if the scope of the function is left. The other kind to transfer parameters is called “call by reference”. Which means the function only receives a reference to the value instead of the value itself. If the function changes the value it follows the reference to the original location of the value. This way, the changes survive the scope of the function.

Translated into the semantics of communication across address space boundaries, “call by value” means, that the data is copied from the sender to the receiver. Each component has an own copy of the data in its address space. When the receiver manipulates the data and wants to make the changes known to the sender, it has to copy the changes back to the sender’s address space. This is called “copy-in-copy-out”. “Call by value” can be implemented for the communication between components using a normal message buffer. The parameters of a function are copied to the message buffer, which is transferred to the receiver, which uses the values of the message buffer. This way both components have an own copy of the data.

“Copy by reference” implies that both components operate on the same copy of the data. If they reside in different address space they have to share the data. Sharing data is only possible using memory shares between the address spaces. To implement this feature, the IDL compiler has to know which of the parameters have to be shared, and the generated communication code has to establish a memory share between the components. To make this semantic known to the compiler, an attribute, such as `share`, is used.

If data is shared between two components using memory shares, other data, which is located in the same memory page, can also be read or even manipulated by the other component. This is a serious security leak. If security plays is more important than the performance advantage of shared data, the sharing has to be implemented using explicit copy operations. Another possibility is to copy the data into an own memory page and share this page only.

If the call to the function is synchronous, the sharing of data can be emulated by copying the data to the component and copying the modified data back to the sender. Thus the sender has the impression that it shared its data with the component.

3.3.2 Live Span of Data

The semantics, required by the users of an IDL compiler, regard mostly the live-span of data. The named semantics "call by value" and "call by reference" regarded the live span of data within the sender's address space. When communicating between different address spaces we have to regard the live-span of data in the receiver's address space, as well.

If one component invokes a function of a remote component the data is only valid in the receiver's address space until the function exits. It does not matter whether the data is "call by value" or "call by reference". To make the data valid beyond the scope of the function the user has to specify an attribute with the respective parameters, which signals this semantic to the compiler. This attribute is `permanent`. This semantic is orthogonal to the above copy semantics, because it involves the provision of memory to store the data outside the functions scope.

When using a simple message, there is no live-span of a function. Thus the data must either be "call by value" or it has to be associated with the `permanent` attribute. Data which is "call by reference" and without the `permanent` attribute is transferred as "call by value".

3.4 Representation of Kernel-Specifics

Kernel-specifics are primarily special IPC semantics. Kernel-specifics also enfold the different communication mechanisms, which will be discussed in the Implementation Chapter. In the following some insight will be provided into the IDL representation of these special semantics.

As could be seen with MIG in Section 2.2.1, does a deep involvement of the kernel specifics into the IDL hinder adaptability. Thus we have to find ways to exploit the kernel specifics, but keep their representation in the IDL and the target code as abstract as possible. The most specific kernel data type is a flexpage.

As described in Section 2.5 do flexpages represent the transfer of access rights to a memory region. Because the compiler has to generate special communication code to transfer a flexpage, we need some way to signal the compiler which parameter represents a flexpage. This can be done using an own data type. The advantages of the explicit flexpage type is that the interface description language contains simple rules to identify a flexpage. It allows the user to specify arrays of flexpages and to use flexpages as members in constructed types. The disadvantage is similar to the involvement of ports in the MIG IDL. The usage of an own data type hinders the portability to a platform which does not support flexpages.

An alternative is to specify the basic elements of a flexpage separately using basic data types. The flexpage describes a memory region, which has a starting address, a size, access right and is owned by a task. To transfer a flexpage can be done by transferring these elements. The advantage is the abstract representation of a flexpage. The disadvantage is that the interface description language has multiple meanings for an address. It can either be a pure address or the starting address of a flexpage. Another disadvantage is that the user has to name several parameters to transfer a single logical unit.

Another possibility is the usage of attribute. E.g. could the attributes `share` and `permanent` in conjunction with a starting address (of type `void*`) indicate a flexpage. The rights and ownership flags have to be represented by individual parameters, which might be coupled to the flexpage through attributes. Such a specification could look like this:

```
[access_rights(rights), ownership(own),
 size_is(pages), share, permanent, in] void * base,
 [in] int pages, [in] fpage_rights_t rights,
 [in] fpage_owner_t own
```

But this specification is rather complex and involves a lot of detail a developer always has to be aware of. These parameters can be combined into a complex type, which is defined in globally available IDL file. The difficulty with both specifications is, that the compiler's parser has to identify these parameters as flexpage and the compiler has to generate appropriate communication mechanisms.

Another alternative could be the usage of *native types* similar to the *native types* of CORBA IDL. A native type is marked as such and whenever a parameter of that type is marshaled or unmarshaled a user-defined function is invoked. The advantage of such a native type is the independence of the underlying communication platform. The disadvantage is that every time a flexpage is transferred in a message a user-defined function has to be invoked. Regarding the fact, that the whole memory management uses flexpages and is used quite often, this could involve a major performance drawback. A user-defined function cannot be included into the optimization of the IDL compiler.

To develop general purpose components for micro-kernel based systems, flexpage are not really essential. But to develop system services for micro-kernel based systems, flexpages are essential, because memory management is an essential part of an operating system.

3.5 Optimizer

As described in Section 2.6 the integration of an optimizer into the IDL compiler is a mandatory task. This Section will have a look at the various tasks an optimizer would have to perform inside an IDL compiler. The compiler does neither target intra address space IPC nor communication across nodes. Hence we perform no in-depth analysis of these possibilities, but mention them for completeness.

The communication inside the same address space, between different threads, allows the optimization of the data transfer by exchanging references to the data. These reference might be used to directly access the data. The problem, which emerges from this mechanism, is synchronization. As soon as it is possible that multiple threads access the same data, e.g. when issuing an asynchronous function call, the accessed data has to be protected to allow only one thread to access it.

A similar problem is the communication across nodes. In case data has to be shared between two components the compiler has to either use a distributed shared memory mechanism of the underlying platform or provide the semantic itself. But like the above scenarios, this one is still beyond the means of this compiler. We will now research the optimization mechanisms, which might be exploited when communicating between threads in different address spaces.

3.5.1 Data Sharing

As mentioned in Section 3.3, data can be explicitly shared between two components. For performance reasons the compiler can decide to temporarily share data between two components. It generates communication code, which establishes a temporary mapping, send a message to the receiver and revokes the mapping after the reply arrived. The message only contains the function identifier and an address where to find the data.

Another level of optimization is to establish a permanent mapping between two components and to exchange data only through this memory window. The disadvantage is, to copy the data into the shared memory region. A message must also be send to signal the receiver to execute a new function. A shared memory region also bears the risk of a security breach.

3.5.2 Indirect String IPC

Another mechanism, mentioned in Section 2.5, can also be used to optimize the performance of the communication code – the indirect string IPC. Instead of copying data multiple times (into the message buffer, the message buffer across address spaces, and the data out of the message buffer), the communication partner only specify the memory area to copy from and to. This way, several copy operations can be spared.

The disadvantage is, that the receiver has to provide the memory for indirect strings. Thus it has to be know which data is received before the message arrives. This problem is discussed in more detail in Section 4.3.

3.5.3 Short IPC

Also mentioned in Section 2.5 is the short IPC of L4. If the data to be sent fits into two CPU word, the short IPC can be used. This kind of IPC is much faster than an IPC transferring a message buffer. Hence this kind of IPC should be used as often as possible.

3.5.4 Copy Optimization

Most of the generated code is marshaling or unmarshaling code. Hence we expect the higher performance gain in optimizing this code. The marshaling and unmarshaling code of the generated stubs mostly consist of copy operations. In a straight forward implementation the data is copied into the message buffer, the communication mechanism copies the data from the caller to the component or the other way, and the receiver copies the data from the message buffer to local variables. Thus the main goal of the optimizer is to minimize the copy operations.

This can be done by simply analyzing data types, but also by using communication mechanisms of the underlying architecture. Those are, in the case of the L4 μ-kernel, the indirect strings or register IPC. To adjust the data structures to these communication mechanisms the data may also have to be reordered, meaning it is marshaled or unmarshaled in another sequence than it is specified in the function's parameter list.

The analysis of the data to be marshaled should also find redundant information, or fill empty spaces in the message buffer, which appear due to alignment differences. The optimizer should also find out which copy operation is the fastest for specific data types. E.g., to copy an array of values as a chunk is faster than copying every single value. If an array is fairly large, but consist of zero values, an optimizer might also compress such information to avoid "pollution" of the message buffer. Another method to compress data is bit-stuffing. Assuming a function has multiple parameters, where each parameter is only a few bits in size. In sum all parameters would be the size of CPU word. Instead of using a CPU word for each parameter, the optimizer should compress them into one CPU word.

3.6 Summary

The Design Chapter discussed several options to implement an IDL compiler, which conforms to the mentioned goals. To support the semantics of message passing the compiler's structure has to be extended. The back-end has to generate more C functions per IDL function than for RPC semantics.

I decided to support the kernel semantics to the following extend. Because of backward compatibility to Flick IDL files I decided to use an own data type for flexpages. This allows the compiler to identify the semantic easier. To provide a more abstract representation of flexpages to the users in the future, I will support a constructed IDL type, which represents a flexpage as a constructed type. The support of a *native type* is not regarded, because the compiler has no influence on user-defined functions.

The possibilities to optimize the generated communication code are numerous. The current implementation is based on the following decision: the compiler uses the features of the underlying platform, it minimizes the copy operations and it uses appropriate copy operations for different kinds of data.

Chapter 4

Implementation

Chapter 4 describes the changes and adaptations made at the DICE compiler to implement the concepts introduced in Chapter 3. Because the main work was coupled to the changes for message passing, it will make up a large part of this Chapter, as well. Other tasks have been the integration of kernel data types and enhancement of optimization mechanisms used by the compiler. A description of general design decisions can be found in Section 4.6.

4.1 IDL

The first interface visible to the user is the IDL file. This Section explains the changes made to the IDL to support message passing, kernel-specifics and optimization.

DICE does support the CORBA IDL and the DCE IDL. And because the CORBA IDL does not support attributes to an extent needed by an optimizing IDL compiler [FHL95], the DCE IDL is the *prime language* of the compiler[1]. Thus all features mentioned below apply to the DCE IDL if not explicitly stated otherwise.

The parser, generating the in-memory representation of the IDL, is created using the GNU tools `flex` and `bison`. The parser reads the input files, checks their grammar and syntax, and generates the in-memory representation. Further on it performs a semantic check of the IDL (whether all used types are defined, etc.). After the compiler finishes pre-processing an IDL file, it can be assumed, the in-memory representation of this particular file is correct.

[1]The CORBA IDL can be extended to support attributes as well, but this would violate the OMG standard. For compatibility reasons I declined changes to the CORBA IDL and use the DCE IDL, which supports attributes naturally.

4.1.1 Message Passing in the IDL

In Section 3.1 I explained the concept of message passing being more general than RPC. When implementing a more general concept into an IDL one might expect the IDL to become overly complex. But when analyzing the basic elements of message passing – sending a message to a component or receiving one from it – the implementation becomes fairly simple.

When using RPC semantic in an interface specification, the parameters of a function are associated with attributes to indicate the direction of their transfer. The attribute `in` means, the parameter's value is send to the component. The `out` attribute indicates parameters returned from the component.

Instead of only associating single parameters with a directional attribute, the whole function may be associated with a directional attribute. Using the attribute `in` means: it is a message which is send from a client to the component. Using the `out` attribute marks messages the specified component may send to other components. To further support the RPC semantic messages without an directional attribute are assumed to follow the RPC semantic. Thus former interface specifications will work with the new compiler, as well. The following is an example of an interface using message passing:

```
interface test_interface {
    [in] void test_func1(int parameter1);
    [out] void test_func2(int parameter2);
    void test_func3([in]int parameter3,[out]int parameter4);
}
```

The first function – `test_func1` – represents a message, which can be send from a client to the component. Its parameter – `parameter1` – is automatically regarded as data to be transferred from a client to the component. The second function – `test_func2` – represents a message, which the component might send. And the last function – `test_func3` – does comply with the RPC semantic.

4.1.2 Flexpages in the IDL

As discussed in Section 3.4 there are several strategies to implement flexpages into the IDL. For portability reasons the communication specifics of the platform should be invisible in the IDL and the generated target code. Therefore the compiler is responsible for using the right communication mechanism for different kinds of data. But if there exists no possibility to distinguish an array of bytes from a memory page, the compiler hardly knows which mechanism to use in the right context.

The current implementation of DICE uses an own data type – `flexpage`. The compiler's parser can identify an explicit data type more easily than data types with implicit semantic. Future work will provide the IDL with a more abstract approach to flexpages, e.g. a constructed data type. The translation into the target code is described in Section 4.2.2.

4.1.3 Indirect Strings in the IDL

If direct IPC is used to exchanged data, the transferred data is only valid until the next receive operation, which will overwrite the message buffer. If the receiver likes to use the data past this point, it has to explicitly copy the data into a different memory area. The message buffer is provided by the compiler generated code and can always be reused by it.

An indirect string IPC has the semantic of a receiver, providing an extra memory area the data can be received in. Because this mechanism provides a great optimization potential, I support it in the IDL compiler. The memory area is fully under the control of the receiver, meaning the receiver has to allocate and free it (see Section 4.3 for memory management details.). But this special semantic has to be specified by the user. He has to provide the memory management code to allocate and free the memory area (see Section 4.3).

If the compiler optimizes the data transport of normal data using indirect strings it has to provide the functionality to manage the memory areas at the receiver's side, which are used to receive the data. The compiler also has to mimic the semantics of an indirect string IPC if it actually uses a direct data transfer, but the parameter was specified with the indirect string attribute.

To illustrate the concept, it might be useful to introduce a short example. The specification of the parameter, which should be transmitted using an indirect string looks like this:

```
[in, ref] char* data
```

Because the communication code has to know the size of the memory area, a parameter using the `ref` attribute has to specify a `size_is` attribute, as well. This size attribute can be omitted, if a `string` attribute is used instead. The `string` attribute implies the parameter of type character array is a zero terminated string. This way the marshaling code can calculate the size of the data itself. If only the `ref` attribute is specified with a character array parameter, the `string` attribute is added automatically. The above example is the same like:

```
[in, ref, string] char* data
```

If the type is different from a character array, the size attribute has to be used to enable the marshaling code to asses the size of data. The specification could look like this:

```
[in, ref, size_is(count)] int *array, [in] int count
```

The `ref` attribute may also be used with any other data type:

```
[in, ref, size_is(sizeof(complex_type))] complex_type * t1
```

When writing the interface specification using the CORBA IDL the indirect string support has been implemented using an own data type — `refstring`. This is a remainder of the Flick usage. The L4 adaptation of Flick implemented indirect strings using an own data type.

4.2 Target Language Representation

Another interface, important for the users of the compiler, is the resulting target code and how it can be employed. To allow the reuse of this code across changing platforms, it is important to export as little kernel specifics as possible to the target language. The current target language of the compiler is `C`. This Section shows the implementation of two of the mentioned kernel specifics – flexpages and indirect string IPC, and support for the message passing semantic.

4.2.1 Message Passing

The former implementation of DICE supports only the RPC semantic. To allow the support of message passing, the back-end of the compiler had to be extended and restructured. As described in Section 3.2.1, there are multiple C function per IDL function. On the client's side as well as on the component's side. In contrast to two C function per IDL function for RPC semantic – one for the sender's and one for the receiver's side. This proportion of multiple C functions to one IDL function has to be considered.

The message passing C functions do also contain only portions of the former C functions. E.g does the unmarshaling function consist only of the unmarshaling code. It uses only the `out` parameters of the IDL function and a reference to the message buffer.

The wait- or receive-any functions will accept any kind of message. To be able to identify a requested function, the function identifier has to be extracted from the message buffer. Because it is placed into the same position for every message, the unmarshaling of the function identifier can be done by the receive-any or wait-any function. The function identifier is the return value of these functions.

4.2.2 Flexpages

When sending flexpages, their descriptions have to be first in the data portion of the message buffer and an additional flag has to be turned on. Therefore they have to be marshaled first. The first CPU word of the data section of the message buffer is usually used for the function identifier. If flexpages are stored in the data section, the function identifier has to be stored after the flexpage descriptions. Therefore the receiver has to find out if a flexpage arrived and use a special case implementation to find the function identifier. The flexpages are separated from the rest of the data by an invalid flexpage.

To illustrate the target code representation of flexpages, I use a short example. When sending a flexpage to the component, the specification looks like this:

```
[in] flexpage page
```

The function to send the flexpage has a parameter which is the flexpage type of the underlying communication system. This type is also used for portability reasons. Most of the existing memory management code uses this type internally. Future versions of DICE will support a more abstract representation of a memory region. Drawbacks of a more abstract representation are additional indirections, which will be introduced into the communication code, because values have to be copied into the internal flexpage type. An advantage is the portability of the code using the abstract representation of a flexpage instead of the internal flexpage type.

The component has to specify a receive window where it will receive the memory region. This receive window is basically the same as the receive buffer for indirect strings. How this receive window can be provided is described in Section 4.3. The receive window has to be made available to the receive function. This is either done by using the parameter, which contains the received memory region. It contains the receive window when the function is called. This receive window has to be set either explicitly by the user before calling the receive function. Or it is set implicitly by the receive function itself, if the function has no flexpage parameter. Both approaches have their advantages and disadvantages, which are described in Section 4.3.

4.2.3 Indirect Strings

The specialty about indirect strings is, that the developer specifies the source and the target location of the data and the communication code only transfers the data from the source to the target. The compiler does not generate any code to copy the data into or out of a message buffer.

There are some restrictions to the usage of indirect strings, which I will explain using an example. The following IDL specification is part of a message passing function, which sends data as indirect string.

```
[in, ref, size_is(size)] char * data, [in] int size
```

The compiler generates at the sender's side a C function, which has the following parameters.

```
char * data, int size
```

At the receiver's side a similar C function header is generated for the receive function, but the parameters have a special semantic.

```
char * date, int * size
```

When receiving an indirect string the receiver has to specify a receive buffer for the indirect string. This receive buffer is handed to the receive function using the parameters, which will also contain the received data. The `data` parameter contains the reference to the receive buffer and `size` the size of the receive buffer. (`Size` is a reference parameter, because the actually received size has to be stored in it.)

The specification of a receive buffer is somewhat more complicated, if the receiver uses a receive-any or wait-any function. These functions do not have parameters other than the message buffer. Thus the message buffer has to be initialized with appropriate memory areas to receive the indirect strings. The next Section discusses the management of these buffers.

4.3 Memory Management at Receiver's Side

Each piece of data, which is received by some component has to be copied to a dedicated memory location. When using receive functions, which are specific to one IDL function these memory locations can be provided by using the "call by reference" mechanism for the parameters. For reasons, described in Section 3.2.1, do the receive-any and wait-any function have only the message buffer as a parameter. Thus the memory location for specific parameters cannot be specified. These specific memory locations are provided when unmarshaling the parameters from the message buffer. This assumes that the data can be stored in the message buffer until the unmarshal function is executed. Because indirect strings and flexpages do not use the memory of the message buffer, this memory has to be explicitly specified.

As indicated above, an indirect string or flexpage parameter can contain the target memory location when the receive function for one specific IDL function is called. This way, the user may specify the memory location explicitly. If a receive-any or wait-any function is called, no such parameter exists. The message buffer has to be initialized appropriately to contain references to the memory locations. This can be done either explicitly by calling an initialization function before calling the receive-any or wait-any function. Or it is done implicitly by calling an initialization function from the receive-any or wait-any function. This initialization function is a call-back function, which is provided as skeleton by the compiler and has to be implemented by the user.

Such a call-back function can be used to support all different kinds of memory. These include pinned memory or memory, which can be used for DMA. Because these possibilities are beyond the scope of the current research, I concentrated on common, page-able memory.

The drawback of such an call-back function is the additional indirection. An advantages is the possibility to reuse memory for the next message and thus minimizing memory usage. This call-back function also allows an elegant and abstract way to handle the memory allocation for received parameters.

4.3.1 Indirect Strings

To accentuate this problem I explain it in more detail for indirect strings. One of the problems when using indirect strings with receive-any or wait-any function is the provision of receive buffers. To illustrate the problems, I will use the following sample IDL:

```
interface foo {
  void f1([in, ref] char* data,
          [in, ref] char* name);
  void f2([in, ref] char* name);
};
```

The user intends to use the parameters as follows: the `name` parameters can be received in any kind of buffer, which can be reused when the component function[2] returns. Their data is used only inside the component's functions. The size of such a name does not exceed 100 bytes. The `data` parameter, on the contrary, has to survive the scope of the component's functions. Ideally the receive buffer for such data should be taken from a special pool, which contains memory blocks of the appropriate size, which are much bigger than the buffers needed for the names, e.g. 1 KByte.

All functions use indirect strings to exchange their parameters. As described in Section 2.5, does the receiver of an indirect string has to provide memory for the indirect string. The receive function `wait_f1` has two parameters – `data` and `name`. These parameters are used to initialize the receive buffers. The user has to specify a valid memory area for them when calling `wait_f1`. The indirect strings are copied into these memory areas.

The receive function has only the message buffer parameter for reasons, which are described in Section 3.2.1. Thus the `wait_any` function does not have specific parameters, which contain the memory areas to initialize the indirect string buffers. One possibility to initialize these buffers is to allocate memory for the buffers inside the `wait_any` function. To enable the user to implement a tailored memory allocation routine a call-back function is executed to allocate the memory. This call-back function is provided as skeleton[3] by the compiler.

To implement the initialization of the indirect buffers inside the `wait_any` function entails that a repetitive execution allocates memory over and over. Even if the old buffer can be reused. The `wait_any` function does not have any knowledge about the further usage of the buffers and hence cannot decide which of the buffers to reinitialize and which not. The solution is to

[2]The component's implementation of the specified function.

[3]A skeleton is a function with an empty body. Skeletons are created by the IDL compiler to simplify the implementation of a function.

place the initialization of the indirect string buffers outside the `wait_any` function.

If the `wait_any` function is used inside a receive-loop[4], the message buffer has to be initialized as well. The generated receive loop does call the call-back function to initialize the indirect strings before it enters the very first wait. The receive-loop does not call the call-back function again. This way, the indirect string buffers will be reused over and over until a function's implementation[5] initializes them with another buffer. This is done using the call-back function again.

If the `permanent` attribute is specified with an indirect string, the receive-loop initializes the indirect string buffer with a new memory area itself. The user must not initialize this buffer himself.

The call-back function can be regarded as a "memory allocation" function to provide the buffers for the indirect strings. There are several ways to use such an allocation function. One is to use it similar to `malloc`. The function takes as parameter the size of the memory to allocate and returns a reference to the allocated memory. Such an allocation function cannot be used, because a function's implementation does not know the size of the buffer of the next indirect string. The function's implementation receives the parameters, which contain an indirect string as double referenced parameters. Thus the function's implementation can assign a new memory area to the variable. Still the implementation does not know the size of the memory area to be allocated.

Another possibility is an allocation function, which knows the message buffer of the receiver loop (e.g. by an global variable). The user indicates to this function, which of the indirect string buffers to allocate a new memory area for. E.g. does the server implementation of `f1` replace the data buffer with a new buffer by calling the allocation function with a parameter saying: replace buffer which contains the data of the parameter `data` of the function `f1`. This could look like this:

allocate_buffer (BUF_F1_DATA);

The allocation function knows by macro magic, the correct indirect string to be initialized with a new buffer. Again the question is, which kind of buffer to use. Because this information cannot be made available to the allocation function (it would have to predict the future – which function will be called next), the only solution is to use one kind of buffer for both, `data` and `names`, and internally copy `data` into the special buffers.

This copy operation could be omitted if the indirect string parameter could be grouped. This is done by extending the `ref` attribute to specify the number of a group this indirect string belongs to. This would make the

[4]Also called server-loop. It receives request from other components, determines the respective function, calls it and returns the result. After that it waits for the next message.

[5]The implementation of an IDL function at the component's side.

above IDL look like this:

```
interface foo {
  void f1([in, ref(2)] char* data,
          [in, ref(1)] char* name);
  void f2([in, ref(1)] char* name);
};
```

This option can only be used to sort the indirect strings. And sorting can only be performed if there is more than one element. Thus if the second parameter of `f1` wouldn't exists, these parameters could not be sorted and thus be grouped. This way both kinds of buffers use the same indirect string again.

Alternatively the `ref` attribute can be used to directly specify an allocation function to be used with the parameter. The IDL specification could look like this:

```
void f1([in, ref(alloc_data)] char * data,
        [in, ref(alloc_name)] char * name);
```

4.3.2 Sparse Strings

Another strategy is to use one indirect string for every possible variable. This would mean for the above IDL to provide a message buffer with three indirect strings, where each one of them has to be backed by memory to receive in. This could be a tremendous waste of memory, but if cleverly organized a good alternative. Assuming we use the above IDL. The IDL compiler will provide a message buffer for each function which uses indirect strings. This message buffer has as many indirect strings as are indirect string parameters in the interface specification (for the above example: three).

At the receiver's side these buffers are initialized using a number to identify each indirect string. We call an indirect string with a number a *slot*. E.g. will the parameter `data` of function `f1` be received into the first slot; the parameter `name` of `f1` will be received into the second slot; and the parameter `name` of `f2` will be received into the third slot. Now the allocation function knows which parameter it initializes, which indirect string to use for it and may set the correct receive buffer for it (e.g. the special kind of buffers needed for the `data` parameter). If some mechanism, such as the indirect string grouping, is used, it may also reuse buffers for indirect strings of the same group. This means, that slot two and three reference the same buffer.

At the client's side, the parameters are marshaled into the message buffer's slots respective to their global position. The unused string identifiers are initialized to zero values. This means that for function `f1` the slots one and two are filled with `data` and `name` and slot three is set to zero. For function `f2` the first two strings are set to zero and the third is set to the `name` parameter. This way the kernel could transfer the correct data into the correct receive buffer.

To make the usage of indirect string buffers more efficient, the string grouping idea could be used again. If two indirect strings are within the same string group, they may share the same receive buffer and thus use the same slot. This idea implies, that all indirect strings of one function have to use a different string group. The effect is, that for the above example we only need two slots in the message buffer. The first slot is reserved for the `data` parameter and the second for the `name` parameters. Thus the sender function `f1` would fill slot one with the reference to `data` and slot two with `name`. Function `f2` would leave slot one empty and fill slot two with a reference to `name`. At the receiver's side we need two slots as well. The allocation function initializes slot one with the special kind of buffer, used for `data`, and the second slot with a "normal" buffer.

The restriction of such an approach is the sparse usage of indirect string[6]. I don't know whether the compiler will return an error or stop processing when it discovers a string of size zero or simply skips it. I also don't know if future kernels will support this feature. And of course there is a restriction of the number of strings to be transferred.

4.4 Optimization

The optimizer influences the generation of fast code by reorganizing the data, which should be sent. Some examples for reorganization are:

- Use as little space as possible. As explained in Section 3.5.4, may multiple values fit into one CPU word. If the resulting data fits into a short IPC, the benefits outweigh the effort to stuff the values into one CPU word.
- Use fixed values (especially indices) as much as possible. Fixed sized data is copied into the message buffer before any variable sized data. Thus the fixed sized data can be marshaled using fixed offsets into the message buffer.

Optimization is not only done by reorganizing data, but also by influencing the code generation. The copy operations can be minimized if a fast copy operation is used for each data type. Constructed data of fixed size, such as `struct`s or arrays, are copied using the `memcpy` operation. The `memcpy` operation copies a chunk of data. This is faster than copying every single element of the constructed data.

Optimization does not only consist of making the marshaling and unmarshaling code fast. Another part of optimization is the usage of the correct

[6]Hence the name – Sparse Strings

communication primitives. E.g. does the compiler use indirect strings for variable sized data. The communication primitive copies the data only once. To copy the data into the message buffer and out of it involves two to three copy operations.

To minimize copy operations the compiler might even decide to use temporary memory shares between two components. This has to be transparently to the user. The optimizer and the code generator generate code for the target language compiler. This means that the code has to be translated another time. The IDL compiler has to know how the target language compiler interprets different code and generate the most appropriate. Thus the code might look awkward.

4.5 Side-Effects

This Section contains parts of the work, which have been done, because the demand for it arouse during the implementations. The test-suite was created to provide a way to test the generated stubs, which is easy to use and automatic. It uncovered several bugs and has thus proved beneficial already. Nonetheless did it take some time to implement this feature. The Flick compatibility mode has been added due to the need to make the transition from Flick to DICE as easy as possible for the developer. The pre-processor is an essential part of the compiler and since a lot of work has been invested in it, it should become reusable by others. Therefore the XML parser output has been integrated.

4.5.1 Test Suite

The compiler generates marshaling and unmarshaling code, which is often hard to read for the developer. If an error occurs and the developer is not sure whether this error occurs because the marshaling/unmarshaling code is incorrect or his own code has a bug, it is a tedious task for the developer to check his own code **and** the generated marshaling/unmarshaling code. That's why the compiler has to generate correct code. To be able to proof this (to some extend) the compiler is able to generate a test-suite for a specified IDL file.

The idea to implement a test-suite has — again — been taken from the IDL^4 compiler. In meetings with Andreas Häberlen, the developer of IDL^4, he pointed out to me, that a test-suite has many advantages. With his suggestions and an example test-suite, generated by IDL^4, I could integrate a test-suite generator into DICE within three days.

The test-suite generates an application, which launches a server-thread to simulate the component and calls this component using the generated stubs. To fill the parameters of the function the test-suite contains code to assign random values to the parameters. These random values are generated in a controlled way, so each error is reproducible. The client stub then marshals and transfers these values to the server thread, which runs the receive loop.

The receive loop unmarshals a request and calls the appropriate component function. Inside the component function the transmitted values are compared to globally stored reference values. If the values don't match an error message is printed. The return values are filled with random values and sent back to the caller. The caller compares the values to global reference values and prints an error message if a value does not match.

The generated test-suite uses the DROPS L4 environment to start threads and print status output, such as the error messages, and is thus limited to run on top of L4 version 2 kernel. Because the code generation for the test-suite does also follow the class organization of the compiler it is very easy to adapt the test-suite generation to the respective target platform.

It showed very helpful to implement this test-suite, because several bugs showed up when testing example IDL files with the test-suite. Thus the test-suite is not only a nice feature for the developer using the compiler, but also for the developer writing the compiler. This feature allows the user of the compiler to be confident about the generated code. Surely, it is questionable to proof the correctness of a tool with the tool itself.

4.5.2 Flick Compatibility Mode

The *Flick compatibility mode* has been implemented into DICE to support the migration from Flick towards DICE. Flick has been used widely to build system service for L4 and it is a ridiculous thought to demand from the users of Flick to change all their code written for the Flick stubs. The Flick compatibility mode allows them to use their legacy code with a new IDL compiler. DICE had to support all the features, Flick had supported. But instead of implementing client and server stubs, which are identical to the Flick generated stubs, DICE provides only a C function interface compatibility.

This means that the client stubs and server functions, which have been generated by Flick have the same parameter number and sequence as DICE generated functions. This allows the developers to simply bind their code against new client stub libraries. But DICE generates different code inside the stubs than Flick does. Thus the communication protocol is quite different. A Flick generated client stubs will not work with a DICE generated server loop. But this means the least effort for the users of Flick. All they have to do, is to replace their hand-written server loop with a call to the DICE generated server loop and a recompile.

4.5.3 Pre-Processor

Because the University of Karlsruhe has a similar project – the development of an IDL compiler for the L4 μ-kernel – it was reasonable to join forces. First steps toward this cooperation have been the exchange of ideas, as mentioned at the beginning of this Chapter, and the implementation of a front-end, which both could use. Because DICE supports more IDLs than IDL^4 does, it was chosen to be the compiler to generate an intermediate format, which both compiler could then use with their back-ends. The format of the intermediate files is XML, because it is a standard format to exchange data.

After DICE parsed the specified IDL files and checked them syntactically and semantically it generates the corresponding XML file. The front-end might be distributed separately from the rest of the compiler to be used as syntax checker or pre-processor by other tools, such as IDL^4.

4.6 Adaptability and Maintainability

The main concept of adaptability is the factory concept, introduced by Volkmar Uhlig. One of the factories is the name-factory. Names for functions, types or variables are used at multiple places within the compiler. If a name is not suitable for a target platform it has to be changed. E.g. is the word `protected` a valid variable name in C, but it is a reserved keyword in C++. Thus we might generated a variable name `protected` if the target language of the compiler is C, but we have to use another string if it is C++. Assuming we used the name in several places, we have to find and change all these occurrences. If, instead, we use a central spot to create the names, we would only have to change the name at a single spot. Additionally, if introducing a new back-end, which uses a different naming scheme, by suffixing every variable name, we do not have to change all names, but may instead overload the respective function, call the base-class' function to get the original name and add the prefix or suffix to this name.

A similar problem arouses, when using classes in the compiler. Assuming you would like to change the class to be used or modify its functionality, because you implement a different back-end, you can overload the base-class with an own implementation of its functionality. But how does the compiler know, that it has to use your new class now. The easiest way to may this class available is to maintain a class-factory as a central "authority", which creates the appropriate classes. When using a derived class the class-factory will produce the new class.

Other ideas, concerning the design of the IDL compiler are common to most compilers. The compiler is separated into a front-end, which reads and checks the input files, and a back-end, which writes the target files and optimizes the generated code. To make the optimization part easier to find I inserted an extra sub-module into the back-end. It consists of the data-representation classes, which are used to find the suiting optimization strategy.

Chapter 5

Performance Evaluation

I measured the following performance numbers on a Intel Pentium Pro (166 MHz) computer with 256KB cache and 64MB RAM. The IPC were exchanged between two L^{4}Linux task, running on L^{4}Linux [HW97] and Fiasco.

5.1 Message Passing vs. RPC

The advantages of message passing over the RPC semantic can hardly be expressed in numbers. The most significant advantage is the broader application spectrum. I like to explain these advantages with three examples.

5.1.1 Region Manager

The region manager is currently implemented using a server loop, which receives requests (page-faults). It locates the corresponding data-space manager and send a request to this data-space manager to provide the appropriate memory region. This is done synchronously. If a request arrives, which should be handled by a different data-space manager, it has to be blocked until the former request is completed, because the region manager is still busy waiting for the data-space manager.

With message passing the region manager can be implemented using asynchronous request to propagate the request to the data-space manager. Thus it can be ready to receive the next request, while the data-space manager is still processing the first request. After the data-space manager finishes the first request it signals the completion to the region manager, which can send the reply to the thread, which initiated the request.

5.1.2 Thread Library

Most of the communication code of the thread library is hand-code, because it it mostly only sends a message. Therefore the code is very prone to errors. If the thread library should be ported to another platform the whole communication code has to be replaced by hand. If using an IDL file the port can be done by recompiling the IDL file. The message passing attributes allow the developer of the thread library to specify all the simple messages as functions of an interface.

5.1.3 DSI

The DSI model is based on the idea, that simple messages are exchanged between components to signal processing states. These messages may also have to appear in a specific sequence. This cannot be done easily using former IDL compilers. Using message passing the developer may specify, that after a specific message arrives another message is expected. Thus he may declare a sequence of messages.

5.2 Flexpage vs. Indirect vs. Direct

As described in Section 3.5 do several strategies to make data available in another address space. I measured and compared three strategies on L4 to determine, which strategy is best for different kinds of communication. These strategies are:

- share memory pages,
- use indirect strings to transfer data, and
- use copy operations and the message buffer.

I measured the performance numbers for data from 8 bytes up to 16KBytes. These are the numbers. Because the minimum size that can be shared is one memory page, the measurements for memory shares start with 4KByte.

5.2.1 Numbers

This Section includes the performance numbers measured for the different communication mechanisms.

pages	average	minimum	maximum
1	4700	3956	16346
2	5474	4717	17984
4	7107	6301	23355

Table 5.1: Number of Cycles to establish a Memory Share

If a memory share is used to exchange data between two address space, the share may either be established before the communication starts or

pages	average	minimum	maximum
1	1757	1652	7200
2	2497	2380	8817
4	4110	3929	10245

Table 5.2: Number of Cycles to revoke a Memory Share

it may be established for each function call. Because the first possibility involves advanced memory management and connection handling, we ignore this possibility in this work.

Because each function call has to establish and eventually to revoke the temporary memory share, I added the two parts in Table 5.2.1.

pages	average
1	6457
2	7971
4	11271

Table 5.3: Number of Cycles to establish and revoke a Memory Share

The numbers in Table 5.2.1 show the cycles needed to transfer one indirect string of the specified size. The string is mapped in the sender's address space (this has been ensured by initializing it). The transfer includes the establishment of a temporary memory mapping and copy operation inside the L4 μ-kernel.

bytes	average	minimum	maximum
8	5081	4265	11131
16	5052	4285	11045
32	5074	4367	11851
64	5135	4890	11204
128	5138	4893	11931
256	5298	4542	12374
512	5379	5077	11437
1024	5800	4982	12844
2048	6548	5884	12554
4096	8288	7806	14972
8192	11823	10874	26729

Table 5.4: Number of Cycles for indirect string transfer

Table 5.2.1 show the performance numbers for the transfer of data using the message buffer. The measurements included a copy operation from the parameter to the message buffer. At the receiver's side the data has been copied out of the message buffer. For the copy operations the `memcpy` operation has been used. The kernel copies the data from the sender's to the receiver's address space. The Table 5.2.1 shows the cycles needed for the pure IPC. The difference is due to the additional copy operation into and out of the message buffer.

bytes	average	minimum	maximum
8	4466	3693	10483
16	4536	4222	10747
32	4517	3800	10394
64	4589	3822	10786
128	4738	4024	10333
256	4980	4277	10388
512	5319	4563	11529
1024	6290	5346	12099
2048	7767	6845	13758
4096	11405	10893	17209

Table 5.5: Number of Cycles for direct transfer (including copy)

bytes	average	minimum	maximum
8	4500	4089	10472
16	4536	4176	9959
32	4503	3817	10401
64	4555	3754	10606
128	4601	4288	10651
256	4701	3965	10609
512	4852	4063	10864
1024	5373	4537	11326
2048	6003	5093	11348
4096	7349	6620	13701

Table 5.6: Number of Cycles for pure direct IPC

A short IPC needs an average of 835 cycles with a minimum of 814 cycles and a maximum of 1246 cycles.

5.2.2 Comparison

I compared the above numbers to find out, which communication mechanism is best suited for different data sizes. The numbers always assume that there is only one big chunk of data to be transferred. That's why the real-world scenarios may differ from the above numbers.

As can be seen in Figure 5.1 does the direct IPC perform best, even though it copies the data three times. The break even with indirect string

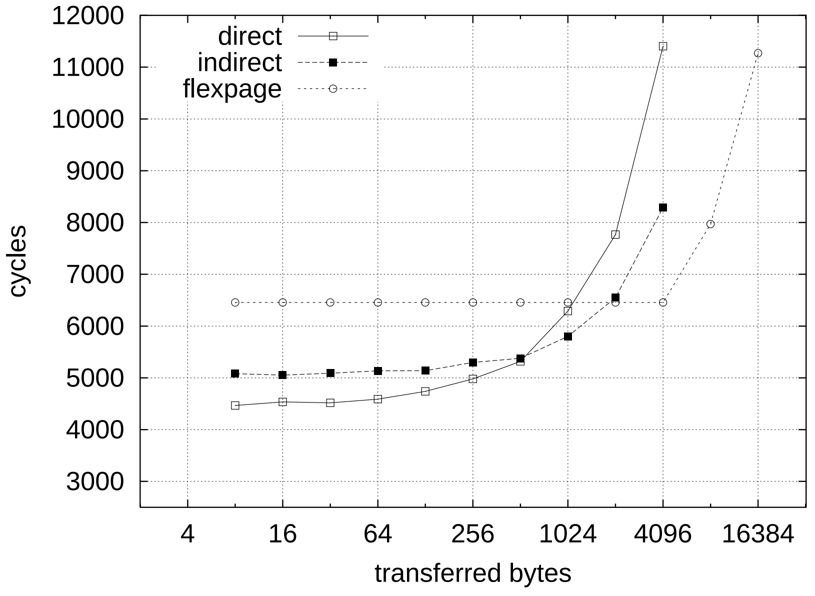

Figure 5.1: A Comparison of direct IPC, indirect string IPC, and flexpage IPC

IPC is for data with a size of 512 bytes. After that the indirect string IPC is faster than the direct IPC. To establish a memory mapping and to revoke it at the end of a function is constant for the size of the first memory page. Direct IPC performs worse for data larger than 1024 bytes. Indirect string IPC performs better than a temporary memory share until about 2 KByte. The memory share does not contain any copy operation yet. So these numbers are only correct if the memory page, which contains the data is directly shared with the other component. But this bears the risk of security breaches.

5.3 Performance Overhead for Message Passing

This section contains the performance numbers for a simple send message. This is the overhead, which is introduced when a compiler generated function is used to exchange messages. I divided the whole communication path into several sections to show the constant overhead, which is mandatory for cross-address space communication, and the overhead introduced by the marshaling and unmarshaling code.

To measure the numbers in Table 5.3 I used a simple message passing function with two parameters. Because the function has to marshal the function identifier too, no short IPC can be used. The code included the

	average	minimum	maximum
sender marshal	65	65	87
IPC send	4524	3761	10567
receiver unmarshal	49	44	68
sum	4638	3870	10722

Table 5.7: Measurements of Performance of Simple Message Passing Function

marshaling of the two parameters the IPC invocation and the unmarshaling at the receiver's side. The send and receive function have been called repeatedly. Because the cycles to enter and leave the send and receive function have been comparatively small I added them to the marshal and unmarshal numbers. The main chunk of the communication is used for the IPC. The performance of the marshaling and unmarshaling code make up only 2,5 percent of the whole message.

5.4 Memory Footprint

When writing system components it is very important to keep the memory usage of the components low. Hence the memory usage of the communication code should be as small as possible. To allow the users of DICE an estimation of the memory usage I wrote down the number of bytes used by the communication code. These are number for code generated with the current release of DICE (version 1.3). The numbers may change for new versions of DICE.

The numbers differ between communication code using a short IPC and communication code using long IPC. But both have some common numbers. All number are counted without using the communication code `inline`. If this option is chosen, the numbers can reduce significantly, because the parameters do not have to be stored on the stack of the communication function.

5.4.1 Common Memory Usage

The common memory usage consists of the standard parameters and the function's parameters. These are:

1. the service structure,
2. the parameters, and
3. the return value

The service structure consist of 8 bytes for `l4_threadid_t` plus 4 bytes for `l4_timeout_t` plus 4 bytes for the exception number. Thus the service structure uses 16 bytes on the sender's stack. Additionally all parameter's and return values of this function use memory on the stack. This is for scalar values their size and for referenced values, such as structures and `out` parameters, the size of a pointer, which is 4 bytes.

5.4.2 Internal Variables for Short IPC

The short IPC does not allocate that many variables on the stack. They are only a few variables to store status information or to variables to be used as place-holder for return value. Keep in mind that a short IPC is only used if the `in` parameters as well as the `out` parameters do not make up more than two[1] double words.

The variables always allocated on the stack is the result variables, which is of the type `l4_msgdope_t`, which is 4 bytes in size. If the function has a return value, a variable has to be allocated on the stack to receive the value. (`gcc` might optimize this additional variable to use the stack-location it already reserved for the return value.) On of the double word of the `in` direction are used by the function identification code, so at most 4 bytes for `in` variables can be added.

The `out` direction can contain (DICE is run with standard options) up to 8 bytes. If at least one of the `out` double words is not used, there has to be a place-holder for the return value. It can be used for both `out` double words.

This is all of the memory needed by the C code. The function does contain assembler code, which performs the actual communication. The memory needed for that piece of assembler has to be added to the memory footprint of the short IPC function.

This sums up to a memory footprint of 24 bytes plus 4 bytes if an `in` parameter is used (if it is 4 bytes in size) plus 4 bytes if at least one `out` parameter is used.

5.4.3 Internal Variables for Long IPC

To measure the memory footprint of a communication function using a long IPC, we have to use a more complicated formula. These function always use a message structure, which contains information about the data sent. Additionally to the 4 bytes for the result variable the stack contains a variable for the return value and the message structure. The message structure has some default member, which are a flexpage descriptor (4 bytes), two message descriptor (each 4 bytes) and at least two double words. The rest of the message structure depends on the data to be transferred. Each indirect string data needs a string descriptor, which is 16 bytes in size. For each flexpage two double words are used in the message buffer. If at least

[1] Three for L4 version X.0

one flexpage is marshaled, there has to be a invalid flexpage in the message buffer as delimiter to other data. All other data is also marshaled into the message buffer.

The above description is about the memory the parameters need. The marshaling code uses some functions, which also might needs some memory. If at least one of the parameters is a character array with the `string` attribute, the `strlen` function is invoked to determine its size. The function `strlen` of the `glibc` needs memory on the stack for the pointer to the string, the return value and an additional integer variable. These three values are on an x86 architecture each 4 bytes in size, which sums up to 12 bytes for `strlen`.

If at least one of the parameters is of a constructed type, a variable sized array or a fixed sized array, the `memcpy` function is invoked. The function `memcpy` uses two pointers and one integer parameter and returns a pointer. It has three internal integer variables. This makes seven 4 byte variables, which sums up to 28 bytes. But the `memcpy` function is declared as `inline` function, which allows `gcc` to optimize the usage of memory for the parameters. Thus only 12 bytes are needed by the internal variables of `memcpy`.

A formula for the memory usage could look like the one describe in Table 5.4.3. This formula does not include additional memory usage, which can be introduced by the used L4 communication primitives. The above formula does also ignores any memory usage of the parameters on the stack.

	4 bytes	result variable
+	4 bytes	flexpage descriptor
+	8 bytes	message descriptors
+	8 bytes	the minimum two double words
+	`str` x 16 bytes	`str` is the number of indirect strings
+	`fpage` x 8 bytes	`fpage` is the number of flexpages
+	8 bytes	delimiter flexpage if `fpage` > 0
+	other parameters	all other data
+	12 bytes	(`strlen`) if at least one `string` parameter
+	12 bytes	(`memcpy`) if at least one constructed, variable or fixed sized array parameter
=	24 bytes	minimum variables
+	parameters	parameters in message buffer
+	parameters	parameters of function (incl. return value)
+	4 bytes	reference to service structure

Table 5.8: Memory Footprint of a long IPC communication function

Chapter 6

Future Work

A major argument for using IDL compilers is to hide changing underlying architectures from its users. That's why the developer of an IDL compiler should always be a step ahead, when thinking about possible future extensions of the compiler. The compiler has to be easily adaptable to new architectures or variations of existing architectures. Proof for this adaptability is the implementation of support for two different version of the L4 μ-kernel and providing the Flick compatibility mode.

To make the integration into the target language even simpler and more transparent the compiler should also be usable as a pre-processor to replace "macros" with communication code. This step does provide significant advantages to the developer using the compiler but implies a lot of work to be done, such as analyzing code and control structures of the target language code.

When looking at the near future, the main work will be done integrating the optimization strategies of IDL^4 into DICE and searching for new possibilities to generate faster stub code. On the other hand does DICE has to provide more and more capabilities toward component based systems. The generated code should integrate not only with the underlying platform – L4 – but also with the generated environment. So maybe, it is possible to extend DICE to use the L4 environment services to build stubs, which are even easier to use. Examples for this could be to "automatically" find out the communication partner's address using the naming service, etc.

Other task regard the component based systems as well. E.g. does the interface inheritance bear new problems. One of them is the version control of the interfaces. How does a components receive loop know which version of the component is requested and which implementation of the specified function it has to call or if the identifier still matches the function. These problems could partially be solved using function tables, where a receive loop manages the different function versions using different function tables. But this indirection will also cause a performance drawback. An advantage of such a feature could be the exchange of components on the fly, by simply inserting or removing a function table from the receive loop's function table list. The costs and advantages have to be analyzed.

In the near future I will also implement an abstract target language representation of kernel specifics, which allows the integration with target platforms other than L4.

Chapter 7

Conclusion

Writing a compiler can be a never-ending story. To make this story a bit shorter it is necessary to ease the changes and adaptation to be included in the future. This requires a well-designed compiler but it also requires the writes of the compiler to think a step ahead to be ready for the next steps, before they have to be done.

An IDL compiler may definitely be used to develop system components for μ-kernels. It has to regard some of the specifics of the underlying architecture, but so does every compiler.

DICE can be regarded as a tool suitable to be used to develop system components for the L4 API and its successors. To fulfill this claim a good foundation has been laid and the basis to extend the compiler to meet future needs has been created.

Bibliography

[Aig01] Ronald Aigner. "Development of an IDL Compiler". Grosser Beleg, Dresden University of Technology, January 2001.

[BBH$^+$98] R. Baumgartl, M. Borriss, H. Härtig, Cl.-J. Hamann, M. Hohmuth, L. Reuther, S. Schönberg, and J. Wolter. "Dresden Realtime Operating System". In *Workshop of System-Designed Automation*, March 1998.

[Bur01] Ken O. Burtch. "The Big Online Book of Linux Ada Programming". http://www.vaxxine.com/pegasoft/homes/book.html, July 2001.

[DJT89] Richard P. Draves, Michael B. Jones, and Mary R. Thompson. MIG - The MACH Interface Generator. Research report, Carnegie-Mellon University, November 1989.

[EFF$^+$97] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. "Flick: A Flexible, Optimizing IDL Compiler". In *PLDI '97*, 1997.

[FHL95] Bryan Ford, Mike Hibler, and Jay Lepreau. "Using Annotated Interface Definitions to Optimize RPC". Technical Report UUCS-95-014, University of Utah, March 1995.

[GJP$^+$00] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. "The SawMill Multiserver Approach". In *9th SIGOPS European Workshop*, September 2000.

[HBB$^+$98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. "DROPS - OS Support for Distributed Multimedia Applications". In *Eigth ACM SIGOPS European Workshop*, September 1998.

[HLP$^+$00] Andreas Häberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. "Stub-Code Performance is Becoming Important". In *WIESS 2000*, October 2000.

[HW97] Michael Hohmuth and Jean Wolter. Prinzessin auf der Erbse – Linux-Portierung auf den Mikrokern L4. *iX*, 1:94ff., January 1997.

[JT99] P.K. Jimack and N. Touheed. *"An Introduction to MPI for Computational Mechanics"*, pages 24–45. Saxe-Coburg Publications, 1999.

[JT00] P.K. Jimack and N. Touheed. *"Developing Parallel Finite Element Software Using MPI"*, pages 15–38. Saxe-Coburg Publications, 2000.

[L4K00] L4KA. "IDL4 - IDL Compiler ". http://www.l4ka.org/projects/idl4/, October 2000.

[Lie96] Jochen Liedtke. "L4 Reference Manual - Version 2.0". 486, Pentium, Pentium Pro, September 1996.

[Mic88] Sun Microsystem. rpcgen - An RPC Protocol Compiler, Sun Microsystem, Inc., 1988., 1988.

[MM01] Fabrice Mérillon and Gilles Muller. "Dealing with Hardware in Embedded Software: A Retargetable Framework Based on the Devil Language". Research report 1391, IRISA, March 2001.

[MRC+00] Fabrice Mérillon, Laurent Révillèrre, Charles Consel, Renaud Marlet, and Gilles Muller. "Devil: An IDL for Hardware Programming". In *"OSDI 2000"*, pages 17–30, October 2000.

[Pet98] Claude Petitpierre. "Synchronous C++, a Language for Interactive Applications". *IEEE Computer*, pages 65–72, September 1998.

[RM01] Laurent Révillèrre and Gilles Muller. "Improving Driver Robustness: an Evaluation of the Devil Approach". Research report 1385, IRISA, March 2001.

[Tro00a] Trolltech. Qt Library On-Line Reference Documentation - About Qt. http://doc.trolltech.com/aboutqt.html, 2000.

[Tro00b] Trolltech. Qt Library On-Line Reference Documentation - Signals and Slots. http://doc.trolltech.com/signalsandslots.html, 2000.

[Uhl99] Volkmar Uhlig. "A Multi-Server Filesystem and Development Environment". Master's thesis, Dresden University of Technology, October 1999.

[WT89] Linda R. Walmer and Mary R. Thompson. "A Programmer's Guide to the Mach User Environment". Tutorial, Carnegie-Mellon University, November 1989.

Appendix A

Sample Code

To ease the understanding of the workings of the IDL compiler I will present some examples for some of the discussed goals.

A.1 Message Passing

For simplicity reasons I included all possible message passing semantics into one IDL file.

A.1.1 The IDL Specification

The following IDL specification (Figure A.1) includes three different functions. The first function represents a message send from the client of a component to the specified component. The second function represents a simple message, which is send the other way – from the component to any receiver, which intends to receive this message. The last function – `f3` – is the function, which sends a message to the component, and receives a reply, with the return values of the function – a typical RPC semantic.

```
interface test {
    [in] void f1(int p1, int p2);
    [out] int f2(int * p3);
    int f3([in] int p4, int p5, [out] int * p6);
};
```

Figure A.1: Sample IDL file

The `out` parameter are pointers, because the values of these parameters are set inside the functions. Thus they have to be “copied by reference”. To make the user aware of the fact, that he has to pass a pointer to the respective C function, the `out` parameters have to be pointers.

A.1.2 The Client's Code

The compiler generates these respective C functions for the client's side – Figure A.2.

```
// test-client.h
#include "dice.h"

void test_send_f1 (l4_idl_service_t * _serv,
    int p1, int p2);
int test_recv_f2 (l4_idl_service_t * _serv,
    int * p3);
int test_call_f3 (l4_idl_service_t * _serv,
    int p4, int p5, int * p6);
```

Figure A.2: A Sample Client Side Header File

The `l4_idl_service_t` parameter is added to allow the identification of the communication partner and to return an error code, in case e.g. the IPC failed. The structure – see Figure A.3 – is defined in the file `dice.h` and included automatically into all generated C header files.

```
typedef struct {
    l4_threadid_t server_id;
    l4_timeout_t timeout;
    dword_t exception;
} l4_idl_service_t;
```

Figure A.3: The IDL Service Helper Structure

The first element – `server_id` – is used to find the communication partner. It has to be set before calling the C function. The second specifies the time to wait for the component to accept the message. It is set by default to timeout never. The last element is set by the C function if an error occurred in the server. This is meant for future use and currently not used.

A.1.3 The Component's Code

The component's side code contains a few more functions, as can be seen from Figure A.4. The most obvious difference to the client code is the declaration of a message structure. This message structure type is used by the `test_wait_any` function and the unmarshal functions. Therefore it has to be declared globally.

The function `test_wait_any` may receive any kind of message for the `test` interface. It will store the received message buffer in the message structure and return the already unmarshal function identifier. This identifier can then be used to determine which unmarshal function to call. Each function,

```
// test-server.h
#include "dice.h"

typedef struct {
    l4_fpage_t  buffer;
    l4_msgdope_t  size;
    l4_msgdope_t  send;
    char  buffer[26];
} test_msg_buffer_t;

unsigned int  test_wait_any(test_msg_buffer_t * _msg_buffer);

void test_unmarshal_f1(test_msg_buffer_t * _msg_buffer,
    int * p1, int * p2);
void test_recv_f1(l4_idl_service_t * _service,
    int * p1, int * p2);

void test_send_f2(l4_idl_service_t * _service,
    int _return_val, int p3);

void test_unmarshal_f3(test_msg_buffer_t * _msg_buffer,
    int * p4, int * p5);
void test_reply_f3_wait_any(test_msg_buffer_t * _msg_buffer,
    int _return_val, int p6);

int  test_server_loop();
```

Figure A.4: A Sample Component Code Header File

which can be received at the component's side, has an unmarshal function, such as `test_unmarshal_f1`. A receive function for a message, which can only be sent to the component is e.g. the `test_recv_f1` function. It will accept only messages, which conform to this function's format. Function, which represent a message which can only be sent by the component are the send functions, such as `test_send_f2`. Because functions with the RPC semantic are usually received with an wait-any function, they have no special receive function, but only a reply-and-wait function. This function sends the reply to the component, which called the function, and waits for the next request.

A.1.4 Usage of the Generated Code

Before calling a function at the client's side, the service structure has to be initialized. It has to contain the identifier of the component, which can be determined using a naming service or similar. The structure also has a timeout member. This member is set by default to zero. This means that the client will wait forever for a reply and the delivery of the message. If an error occurred during the transmission the exception member is set. After the service structure has been initialized the client functions can be invoked.

At the receiver's side the component may implement either server loop for the requests which can be received by the component, or it uses the receive function within other functions. Thus, it may implement a communication mechanism different from the classical client/server approach.