Component Interfaces in a Microkernel-based System

Lars Reuther* Volkmar Uhlig† Ronald Aigner*

*Dresden University of Technology
Institute for System Architecture
{reuther,ra3}@os.inf.tu-dresden.de

†University of Karlsruhe
Institute for Operating- and Dialoguesystems
volkmar@ira.uka.de

Abstract

Existing component models are targeted towards flexible software design and load distribution between multiple nodes. These systems are mainly designed for interoperability. Thus, they are very general and flexible, but slow. Building a microkernel-based system using existing component technology would result in bad overall system performance. We propose an approach to overcome the limitations of existing component systems while maintaining their advantages. This paper gives an overview of a new IDL compiler, *FIDL*, which uses knowledge of the underlying communication mechanism to improve the performance of component-based systems.

1 Introduction

Microkernel-based systems are gaining more and more attention. They provide a flexible approach to deal with the complexity of operating systems by dividing systems into smaller units or *components*. However, early systems like Chorus [10] or Mach [4] suffered from poor inter process communication (IPC) performance. This resulted in the common opinion that microkernel-based system are inherently slow. Recent work [7, 6] has shown that modern microkernel architecture can improve IPC performance significantly and that microkernel-based systems can approach the performance of traditional monolithic systems.

However, there is another problem such systems have to attack: usability. The microkernel provides general abstractions such as address space protection and IPC. While this enables the kernel developer to highly optimize the microkernel, it can be difficult building large systems with such abstractions. Building a complex communication interface using only the microkernel interface is time consuming and error-prone. Instead, a software designer needs ways to specify the component interfaces at a higher level. Invocation-code can be generated automatically from these specifications.

A closer look at the structure of microkernel-based systems shows that they are quite similar to *Distributed Systems* like *CORBA* [9] or *DCOM* [2]. Both types of systems are designed of several servers—or components—which interact. In distributed systems, multiple components interact via well defined interfaces. These are declared in special languages, *Interface Description Languages (IDL)*. *IDLs* are languages that describe interfaces between components. They are independent of the programming language which is used to implement the components. An IDL compiler generates the function stubs for both the sender (client) and receiver (server) of an IPC (Fig. 1).

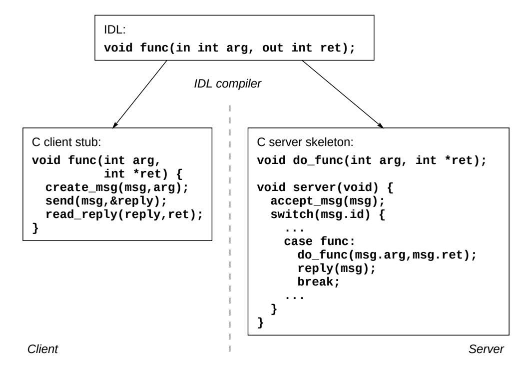

Figure 1: Automatic stub generation by an IDL compiler

The generated stubs perform two operations:

1. Convert the arguments of the function call to/from the message buffer (*marshaling*).

2. Do the IPC call.

The need for IDL systems in Distributed Systems arose for various reasons:

- Support for software design,
- Automatic code generation, this eliminates a common source of errors in the software development.
- Code reuse and integration.

But one aspect was ignored: performance of the generated code. First, component-based software was designed for distributed systems with interconnections of about 1MByte/s. Thus, the costs of argument marshaling were believed to be negligible because of the large costs of network communication. But this is not true for current network technology and definitely not true for current microkernel IPC. A large portion of the total cost of a function call between two threads is argument marshaling especially with current microkernel IPC.

This paper describes ideas for an IDL compiler using detailed knowledge of the underlying communication mechanism to optimize argument marshaling. Furthermore, we allow the user to influence the code-generation process with additional meta information about involved components. This work is motivated by experiences we gained using an existing IDL compiler (*Flick* [3]) to build a Multiserver File System on top of the L4 microkernel [12].

2 Towards fast IDL systems

One of the first projects to deal with the problem of building a fast IDL system was *Flick* by the University of Utah [3]. Its aim is to build a highly flexible IDL compiler which can be used with various IDL types as well as generate code for different communication platforms. One of its major ideas to improve the performance of the argument marshaling is to enable the native language compiler (i.e., the C compiler) to optimize the marshaling code by inlining this code instead of using separate marshaling functions. The work showed that runtime overhead of marshaling and unmarshaling can be reduced significantly by using inlined code.

But Flick still shares some drawbacks with other IDL systems. One of those drawbacks is the limited ability to optimize the marshaling towards specific communication platforms. Communication is based on a separate message buffer for the arguments. Fig. 2 shows how this traditional argument marshaling works for a function call between two components in separate address spaces. The arguments of the function call are copied to a communication buffer in the sender component, this buffer is copied to a communication buffer in the receiver component and then to the argument buffers of the server side function.

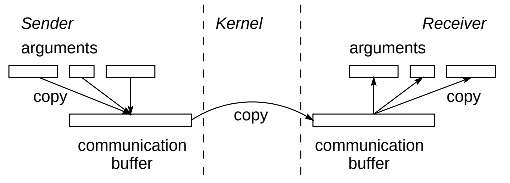

Figure 2: Traditional argument marshaling

This method involves three copy operations, although only one is necessary. The copy between the two address spaces by the kernel is mandatory to uphold memory protection. The separate communication buffers are not required in case of a communication between components on the same host. Instead, the arguments can be copied directly from the client buffers to the server buffers, causing only one copy operation.

But copy operations are still expensive, especially for large amounts of data. Instead of copying the data, the client buffer can can be shared between client and server.[1] That eliminates the copy operation, but causes additional costs for establishing the mapping. Those costs depend very much on the microkernel and hardware architecture. This requires the possibility to influence the generation of the marshaling code, e.g., to specify a threshold for the use of mappings instead of copying the arguments. But sometimes it is not even necessary to establish the mapping by the marshaling code, some systems already provide shared memory, which can be used to transfer the arguments. Again, this requires the possibility to specify the target environment to enable the IDL compiler to customize the code generation.

The ability to customize code generation is a very important requirement for the design of an IDL compiler for a microkernel-based system. The optimal marshaling method varies between different target architectures and system environments.

[1] Assuming client and server trust each other

This cannot be accomplished by a generic marshaling method. For example, modern microkernels use registers to transfer small amounts of data, but the exact number of available registers depends on the particular hardware architecture. A good example is scatter gather IPC, the ability to transfer data from/to scattered buffers (Fig. 3). The IDL compiler needs hints whether the target kernel is able to use this mechanism or not.

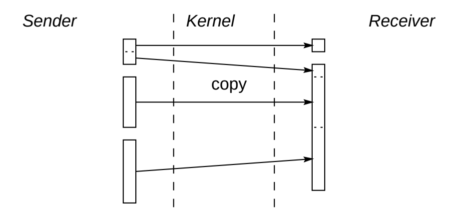

Figure 3: Scatter gather IPC

The optimal communication method also depends on the location of the communication partners. In the case of intra-address space communication, arguments can be passed by references. This is not possible in the case of inter-address space communication. The IDL compiler should consider this by generating different implementations for the marshaling functions dependent on the location and trust of the communication partners.

To summarize, an IDL system for microkernel-based systems must target the following:

- Reduce the marshaling costs, especially avoid copy operations.
- Provide a flexible mechanism to customize the code generation.
- Generate different versions dependent on the location of the communication partners.

The next section describes the approach for a new IDL compiler which considers these requirements.

3 FIDL

Attacking the mentioned problems and incorporating our ideas we developed a new IDL compiler, *FIDL*. FIDL is based on an extended COM IDL. We chose COM for the following reasons:

- COM allows the specification of additional attributes of function arguments. Those attributes can be used to give hints to the IDL compiler, e.g., whether a string should be copied or transferred by reference.
- The COM type system is oriented to the C/C++ programming language. This provides more knowledge for the optimization of the marshaling code.
- COM does not dictate a copy in/copy out semantic.

Like Flick, FIDL creates the marshaling code as inline functions, thus enabling the C compiler to do the optimization. The IDL compiler itself can optimize the remote function call even further by exploiting the IPC mechanism of the underlying microkernel. As described above, it may be faster to establish a temporary mapping to transfer a larger amount of data rather than copying it to a separate communication buffer. But the exact threshold for that decision depends on the hardware architecture. Similar to COM-IDL, that kind of meta information can be provided in a separate *Application Configuration File (ACF)*. This file can contain the following information:

- Hints for the IDL compiler, like the value of the threshold for using a temporary mapping.
- Information how the IDL compiler must marshal and unmarshal user defined data types. Those functions are used to transfer complex data type like linked lists, which can not be handled by the compiler itself. This is similar to the *type translation information* of the Mach interface compiler [1] or the native data types in CORBA.
- Specialized implementations for the marshaling/unmarshaling or the IPC code. This can be used to optimize some or all function calls manually, e.g. to use existing shared memory areas.
- Functions to customize the memory allocation and synchronization.

To summarize, the ACF contains all information to adapt the code generation to a particular system architecture.

The IDL generates different versions of the client and server stubs, depending on the location of the communication partner. If the sender and destination threads reside in the same address space, arguments can be passed by memory references. If the sender and destination are in different address spaces, the arguments must be copied or mapped. The proper implementation is assigned by a function table. This function table is created and initialized during the setup of a communication relation.

4 Measurements

To evaluate our ideas, we implemented a FIDL prototype. It generates the communication code for L4 IPC. Currently, it provides only a restricted functionality, it can only handle basic data types and arrays.

For our measurements we used a Pentium machine running the L4 microkernel resp. Linux (for rpcgen).

4.1 Marshaling costs

In our first test we measured the overhead caused by the argument marshaling. Fig. 4 shows the interface specifications for FIDL, Flick and *rpcgen*, the IDL compiler for SUN RPC [11].

SUN RPC:

```
struct msg {
  unsigned int a;
  unsigned int b;
  char c<20>;
};

program rpc_test {
  version testvers {
    void func(msg) = 1;
  } = 1;
} = 0x20000001;
```

Flick:

```
typedef string<20> string20;

module test {
  interface test {
    void func(in long a, in long b,
              in string20 c);
  }
}
```

FIDL:

```
library test {
  interface test {
    void func([in] int a, [in] int b,
              [in,size_is(20)] char *c);
  }
}
```

Figure 4: IDL sources

All IDL compilers used a separate communication buffer. Table 1 shows the marshaling costs and additionally the costs for a hand-coded version of the argument marshaling.

IDL compiler	Marshaling costs (cycles)
rpcgen	3275
Flick	471
FIDL	248
hand coded	161

Table 1: Marshaling costs

The large difference between the overhead of rpcgen on the one and FIDL and Flick on the other side is mainly caused by two reasons:

1. rpcgen creates a hardware independent representation of the arguments (XDR). This is required in distributed systems where components run on different hosts.

2. rpcgen uses separate marshaling functions rather than inlining the code to the function stubs.

4.2 End-to-End communication

The more interesting numbers are the costs of the actual function call. Those costs include the argument marshaling and the IPC call. Fig. 5 shows the times of a function invocation depending on the argument size for Flick and different versions of FIDL.

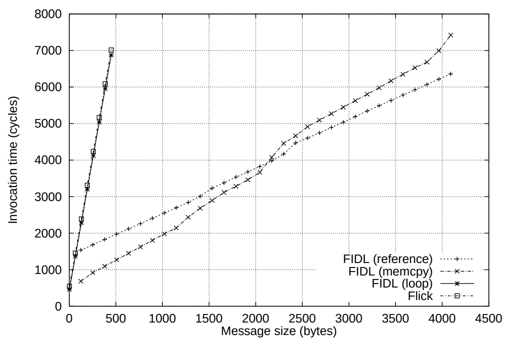

Figure 5: Function invocation costs, function prototype:
`void func(char *string)`

The three versions of FIDL use different methods to marshal the function arguments.

FIDL(loop) uses the same implementation like flick, each character of the string is copied to the message buffer in a loop. *FIDL(memcpy)* uses the `memcpy` function to copy the string to the message buffer. This function is much better optimized than the loop in the first case. *FIDL (reference)* does not use a separate message buffer, instead it passes a reference to the string to the microkernel, the microkernel copies the string directly from the original buffer. As explained above, this eliminates one copy operation, but introduces a larger overhead in the IPC communication as the measurement results in Fig. 5 show.[2] This confirms our claim, that a very flexible IDL compiler is required, which for example generates different different marshaling code for strings depending on the size of the string.

5 Outlook

FIDL combines established and well understood technology, IDL compilers, with new ideas for the optimization of communication between components. Our initial performance is promising.

A fast communication mechanism is a key requirement for the success of a microkernel-based system. However, communication is only one basic mechanism in a component system. More *services* are required on top of those mechanisms such as the *CORBA Services* [8]. Further work is required to provide a complete infrastructure for supporting the development of systems such as *DROPS* [5] on top of microkernels.

References

[1] Richard P. Draves, Michael B. Jones, and Mary R. Thompson. MIG — the MACH Interface Generator. Unpublished manuscript from the School of Computer Science, Carnegie Mellon University.

[2] Guy Eddon and Henry Eddon. *Inside Distributed COM.* Microsoft Press, 1998.

[3] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1997.

[4] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *USENIX 1990 Summer Conference*, pages 87–95, June 1990.

[5] Hermann Härtig, Robert Baumgartl, Martin Borriss, Claude Hamann, Michael Hohmuth, Frank Mehnert, Lars Reuther, Sebastian Schönberg, and Jean Wolter. DROPS - OS Support for Distributed Multimedia Applications. In *Proceedings of the Eigth ACM SIGOPS European Workshop*, 1998.

[6] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, 1997.

[7] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, 1995.

[8] The Object Management Group (OMG). *The Complete CORBAServices book.* http://www.omg.org/library/csindx.html.

[9] Alan Pope. *The Corba Reference Guide: Understanding the Common Object Request Broker Architecture.* Addison-Wesley, 1998.

[10] M. Rozier, A. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS Distributed Operating System. *Computing Systems*, 1(4):305–370, 1988.

[11] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. Technical report, Sun Microsystems Inc., 1995.

[12] Volkmar Uhlig. A Micro-Kernel-Based Multiserver File System and Development Environment. Technical Report RC21582, IBM T.J. Watson Research Center, 1999.

[2] The overhead is caused mainly by a more complex setup of the copy operation in the kernel.