Systems Design and Implementation

II.3 Stub Code Generation with IDL4

System Architecture Group, SS 2009

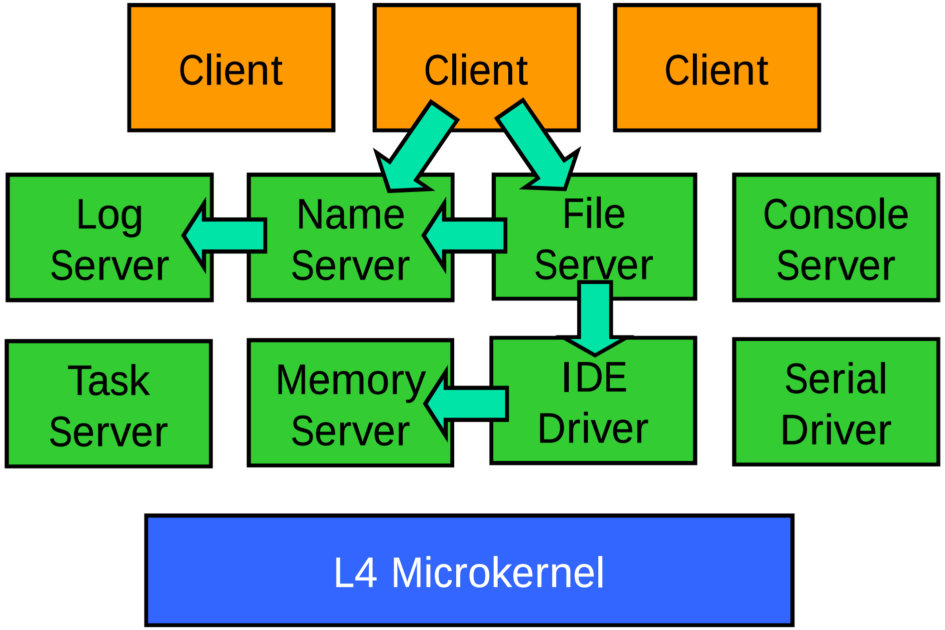
University of Karlsruhe

May 8, 2009

Jan Stoess

University of Karlsruhe

Introduction



- Goal: Multiserver Operating System
- Components need to interact frequently
- Common operation: Send request to another component, wait for reply
- System will contain a lot of communication code

Introduction

```
void some_function(l4_idl_service_t *_service, const char *str1,
        int len1, const char *str2, int len2)
{
   l4_msgdope_t _result;
   unsigned _offset, _tmp_size;
   struct __msg_buffer_struct__ {
       l4_fpage_t fpage;
       l4_msgdope_t size;
       l4_msgdope_t send;
   } *_msg_buffer;
   _tmp_size = 20+strlen(str1)+1+4+strlen(str2)+1+4;
   _tmp_size = (_tmp_size & ~0x3) + ((_tmp_size & 0x3) ? 4 : 0);
   _tmp_size += sizeof(l4_fpage_t) + 2*sizeof(l4_msgdope_t);
   _msg_buffer = (struct __msg_buffer_struct__ *)alloca(_tmp_size);
   _tmp_size = _tmp_size >> 2;
   _msg_buffer->size = L4_IPC_DOPE(_tmp_size, 0);
   _msg_buffer->send = L4_IPC_DOPE(_tmp_size, 0);
   *((dword_t*)(&(_msg_buffer->buffer[0]))) = some_opcode;
   *((int*)(&(_msg_buffer->buffer[4]))) = len1;
   *((int*)(&(_msg_buffer->buffer[8]))) = len2;
   _offset = 12;
   _tmp_size = strlen(str1)+1; // include terminating zero
   *((dword_t*)(&(_msg_buffer->buffer[_offset]))) = _tmp_size;
   memcpy(&(_msg_buffer->buffer[_offset+4]), str1, _tmp_size);
   _offset += _tmp_size+4;
   _tmp_size = strlen(str2)+1; // include terminating zero
   *((dword_t*)(&(_msg_buffer->buffer[_offset]))) = _tmp_size;
   memcpy(&(_msg_buffer->buffer[_offset+4]), str2, _tmp_size);
   _offset += _tmp_size+4;
   l4_i386_ipc_call(_service->server_id, _msg_buffer,
     *((dword_t*)(&(_msg_buffer->buffer[0]))), *((dword_t*)
     (&(_msg_buffer->buffer[4]))), *((dword_t*)(&(_msg_buffer->
     buffer[8]))), L4_IPC_SHORT_MSG, (dword_t*)&(_msg_buffer->
     buffer[0]), (dword_t*)&(_msg_buffer->buffer[4]), (dword_t*)
     &(_msg_buffer->buffer[8]), _service->timeout, &_result);
   if (L4_IPC_IS_ERROR(_result))
     THROW_EXCEPTION(_service, L4_IPC_IS_ERROR(_result));
}
```

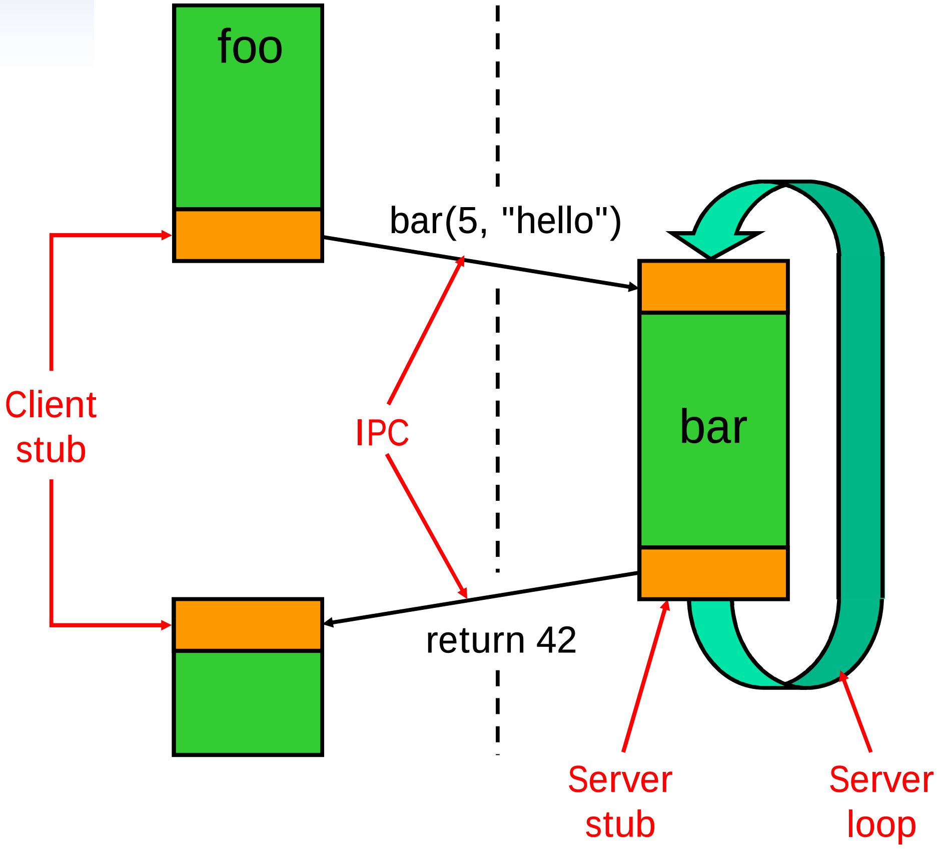
TOO MUCH WORK

- Writing communication code is a tedious and error-prone task

  ⇒ Don‘t do it

- Tools like IDL^4 can generate this kind of code automatically

Remote Procedure Call



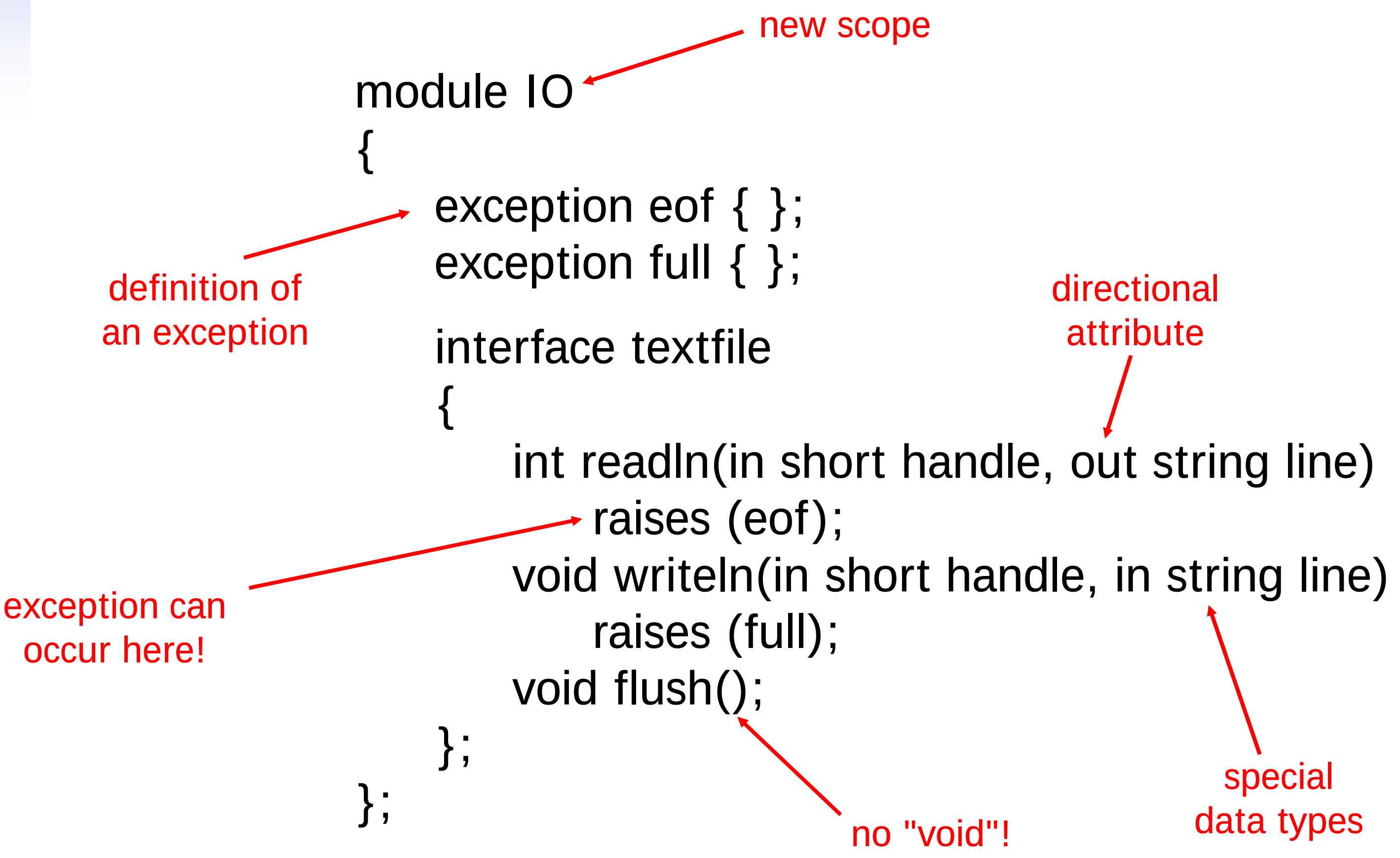
int bar(in int len, in string message)

- Parameters and return values must be copied via IPC
- Stub code required on both sides
- Messages need to be created (marshalled) and analyzed (un-marshalled)
- Server loop demultiplexes requests
- Formal specification

Outline

- Motivation
- Remote Procedure Call
- Defining Interfaces with CORBA IDL
  - General Structure
  - Available Data Types
  - Inheritance
- Using IDL^4
- Working with Generated Code

IDL: General Structure



```
module IO
{
     exception eof { };
     exception full { };

     interface textfile
     {
          int readln(in short handle, out string line)
               raises (eof);
          void writeln(in short handle, in string line)
               raises (full);
          void flush();
     };
};
```

new scope

definition of an exception

directional attribute

exception can occur here!

no "void"!

special data types

- More details: See IDL^4 User Manual

IDL: Data Types

- Basic Types

```
char
short
long
long long
float
double
long double
boolean
octet
```

- Alias Types and Arrays

```
typedef short word_t
typedef char sector[512]
```

- Structs

```
struct foo {
  int a;
  word_t b;
  char c;
}
```

- Strings

```
string
string<30>
```

bounded length

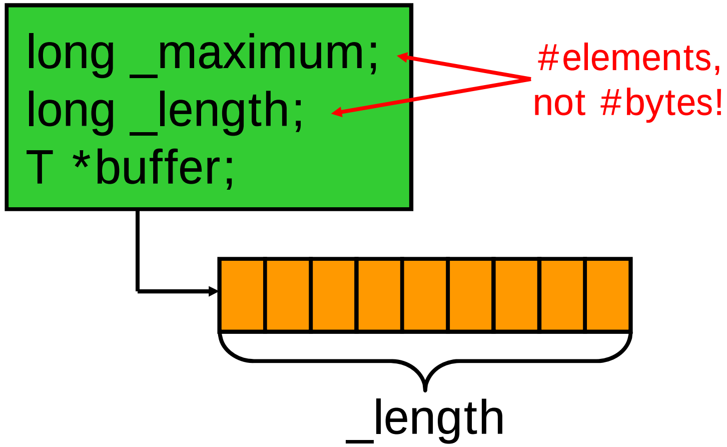
- Flexpages

```
fpage
```

introduced later

IDL: Sequences

```
typedef sequence<char>
   char_seq_t;

typedef sequence<short, 10>
   short_seq_t;
```



```
long _maximum;
long _length;
T *buffer;
```

#elements, not #bytes!

~~int foo(in sequence<char> x)~~

- Sequences are arrays of variable length
- Storage for out sequences is allocated via CORBA_alloc() and must be freed with CORBA_free()
- Maximum size must be known before the call. Unbounded sequences?
- Sequences can only be used with typedef
- No sequences of sequences

IDL: Inheritance

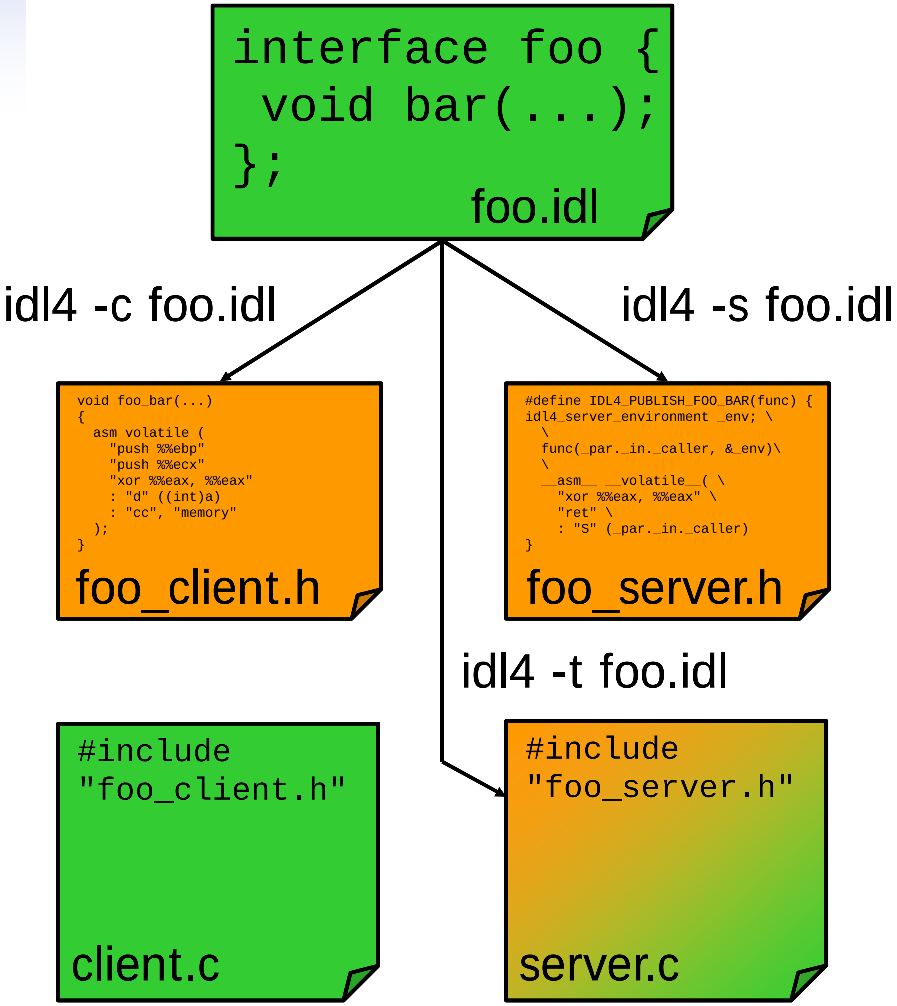
```
[uuid(1)]
interface fruit {
  void eat();
};

[uuid(2)]
interface merchandise {
  void buy(in int price);
};

[uuid(3)]
interface banana : fruit,
             merchandise
{
  void peel();
};
```

- Interfaces can inherit from other interfaces
- Multiple inheritance is allowed
- Functions cannot be overloaded
- Individual threads can only serve a single interface
- To avoid conflicts, assign unique IDs to every interface!

Outline

- Motivation
- Remote Procedure Call
- Defining Interfaces with CORBA IDL
- Using IDL4
  - Generated Files
  - Command Line Parameters
- Working with Generated Code

Invoking IDL^4



```
interface foo {
 void bar(...);
};
```

foo.idl

idl4 -c foo.idl

```
void foo_bar(...)
{
  asm volatile (
    "push %%ebp"
    "push %%ecx"
    "xor %%eax, %%eax"
    : "d" ((int)a)
    : "cc", "memory"
  );
}
```

foo_client.h

idl4 -s foo.idl

```
#define IDL4_PUBLISH_FOO_BAR(func) {
idl4_server_environment _env; \
  \
  func(_par._in._caller, &_env)\
  \
  __asm__ __volatile__( \
    "xor %%eax, %%eax" \
    "ret" \
    : "S" (_par._in._caller)
}
```

foo_server.h

idl4 -t foo.idl

```
#include
"foo_client.h"
```

client.c

```
#include
"foo_server.h"
```

server.c

- Two separate header files for client and server stub code
- #include client header in every client application
- #include server header in the server
- Generate server template once, then add implementation for each operation

Command Line Options

```
idl4 [OPTIONS] input.idl
```

`-c, -s, -t`	Choose output: Client header, server header, or server template
`-Wall`	Enable all warnings
`-I path`	Search this path for #includes
`-D macro=val`	Define a macro
`-p platform`	Select another platform (ia32, generic)
`-i api`	Select another kernel API (v2, x0, v4)
`-m lang`	Select language mapping (c, c++)

Outline

- Motivation
- Remote Procedure Call
- Defining Interfaces with CORBA IDL
- Using IDL^4
- Working with Generated Code
  - Invoking an operation
  - Implementing an interface
  - Customizing the server loop

Client side

```
#include "io_client.h"

int main(void)
{
  CORBA_Environment env
    = idl4_default_environment;
  IO_textfile server;
  int fhandle; char *line;

  /*get server and file handle*/

  IO_textfile_readln(server,
     fhandle, &line, &env);

  switch (env._major) {
    case CORBA_USER_EXCEPTION:
    case CORBA_SYSTEM_EXCEPTION:
  }

  CORBA_free(line);
}
```

- Implicit parameters: server threadID, environment
- Always initialize the environment!
- System exceptions can always occur, e.g. when IPC fails
- Out strings and out arrays must be freed using CORBA_free()
- Simple alloc/free in the sample code

Server side

```
#include "io_server.h"

int IO_textfile_readln(
  CORBA_Object _caller,
  int fhandle, char **line,
  idl4_server_environment *env)

{
  strcpy(*line, "Hello world");
  /* or */
  *line = "Hello world";

  if (handle<0) {
    CORBA_exception_set(env,
      ex_eof, NULL);
    return;
  }

  return strlen(*line);
}
IDL4_PUBLISH_IO_TEXTFILE_READLN
(IO_textfile_readln);
```

- Extend the skeleton function in the server template file!
  - Remove duplicate interfaces

- Implicit parameters: ThreadID of the caller, environment
- Stub provides buffers for output values; other buffers may be used instead
- No need to call CORBA_free()

Server loop

```
#include "io_server.h"

int IO_textfile_vtable[] = ...;

void IO_textfile_server()

{
  struct {
    unsigned int stack[768];
    unsigned int message[...];
    idl4_strdope_t str[...];
  } buffer;

  /* initialize string dopes */

  while (1)
    {
      reply_and_wait(...);
      process_request(...);
    }
}
```

- Reply&Wait is used to send reply and to receive next request
- Function number is extracted, and the corresponding stub is called
- Preallocated buffers are used for output
- Loop performs a stack switch to the buffer; make sure stack is big enough!

Summary

- IDL^4 generates communication code from a formal interface definition
- To build a component,
  1. Define the interface(s) in CORBA IDL
  2. Run IDL^4 with -c and -s to generate client and server stubs
  3. Get a server template with –t and implement the operations of each interface
- Recommended reading: IDL^4 User Manual (available from the course website)

Questions?

The Example in DCE IDL

```
library IO
{
    exception eof { };
    exception full { };

    interface textfile
    {
        int readln([in] short handle, [out, string] char **line)
            raises (eof);
        void writeln([in] short handle, [in, string] char *line)
            raises (full);
        void flush();
    };
};
```

IDL: Page faults

```
interface pager {
    [kernelmsg(idl4::pagefault)]
    void pagefault(
        in long addr,
        in long uip,
        in long access,
        out fpage fp
    );
};
```

- Client sends the address of the fault and its instruction ptr
- Server replies with the requested page
- The fpage type maps to struct idl4_fpage_t, which also contains map base, permissions
- IDL^4 provides macros to access this struct
- Also works for interrupts and exceptions
- Details: See manual

IDL: Files and Attributes

- Multiple IDL files
  - **`#include “idl-file.idl”`**
- Type import from C/C++ header files
  - **`import “header.h”;`**
  - No need to define types twice

- Function attribute **`oneway`**
  - No result, no **`out`** or **`inout`** parameters
  - No exceptions
- Output parameter attribute **`prealloc`**
  - Pre-allocation of buffers by user
  - No implicit *_alloc() by stub

IDL: Implementation BUGS

What IDL4 doesn't do (even if the manual claims so)
From SDI@UKa Wiki

- you cannot add some data to an exception (even though it appears so from reading the include files)
- The standard server loop from the server template allocates 8000 bytes, no matter how much is actually needed. If you want to receive more, adjust it by hand!
- sequences without a maximum length are NOT supported, neither are sequences of strings; actually idl4 might crash while compiling
- idl4 cannot deal with namespaces. Therefore, we unfortunately have to do without them.
- It seems to be that idl4 does not recognize [prealloc] in conjunction with strings.
- when receiving a sequence in a server, the _maximum value might not be set correctly. So don't rely on that.

Next week

- Tuesday: Lecture (Naming)
- Thursday: Debugging Tutorial
    - Failcase Session
    - Takes Place in Room 148, 50.34
- Homework – IDL^4 exercise
    - Make privileged system calls available to threads outside the root task!
        - Ignore MemoryControl and ProcessorControl
    - Use IDL^4 for stub code generation!
    - Detailed instructions in the SDI Wiki
    - See also assignment02.pdf

Lecture Schedule

21.4.	Introduction
28.4.	Communication
5.5.	OS Interfaces
12.5.	Naming
19.5.	J. Stoess – Project Kittyhawk
26.5.	File Systems
2.6.	Threads, Scheduling
9.6.	Memory Management
16.6.	Drivers
23.6.	Device Service Design (2)
30.6.	Lab
7.7.	Lab
14.7.	Lab
21.7.	Lab
23.4.	L4 API Crash Course (I)
30.4.	L4 API Crash Course (II)
7.5.	IDL4, Debugging on L4
14.5.	Debugging on L4 (Lab)
21.5.	- Christi Himmelfahrt -
28.5.	Name Service Design (3)
4.6.	File Service Design (2)
11.6.	- Fronleichnam-
18.6.	Task Service Design (2)
25.6.	MM Service Design (2)
2.7.	Lab
9.7.	Lab
16.7.	Lab
23.7.	Lab Demos + Conclusion