CORBA Application Development Environment Using Reflection

Kazuhiro Fujieda **Takuo Watanabe** **Koichiro Ochimizu**
Japan Advanced Institute of Science and Technology, Hokuriku
{fujieda,takuo,ochimizu}@jaist.ac.jp

ABSTRACT
CORBA is the distributed object computing environment standardized by OMG. ORB, the core component of CORBA, allows developers to easily write servers implementing distributed objects and clients manipulating these distributed objects. This easiness depends on stub and skeleton files generated by the IDL compiler from IDL files. Development process of CORBA applications, especially building process, is conversely complicated by the existence of these files.

In this paper, we propose the runtime environment to allow CORBA applications to generate and incorporate necessary stubs or skeletons on demand at runtime. This environment makes the building process simplified very much. In order to realize this environment, we make use of the characteristics of programming languages providing the reflection, and two elements standardized in CORBA: the Inter-operable Object Reference and the Interface Repository. The sophisticated composition of them can simplify not only the building process but the whole development process.

Keywords
CORBA, reflection, runtime stub/skeleton generation

INTRODUCTION

The Common Object Request Broker Architecture (CORBA) [Obj98a] is the distributed object computing environment standardized by the Object Management Group (OMG). The Object Request Broker (ORB), the core component of CORBA, allows a server, providing services through a computer network, to be implemented as a collection of distributed objects, and its clients using these services by invoking methods (called *operations* in CORBA terminology) of these objects.

Distributed objects and their clients can be implemented with various programming languages including non-object-oriented languages. In object-oriented languages, distributed objects can be implemented by class definitions, and clients can handle them just like ordinary objects.

This uniformity between distributed objects in CORBA and objects in programming languages depends on program codes called stubs or skeletons stored in many files. These files are generated by the IDL compiler from IDL files which contain definitions of interfaces to distributed objects.

These files make the building process in developing CORBA applications much complicated, because it needs to manage dependency among many files which includes not only source files edited by developers but also IDL files and stub and skeleton files generated from them.

In this paper, we propose the runtime environment of CORBA application which automatically generate required stub and skeleton codes on demand at runtime and incorporate them into applications, so that the building process become much simpler. In order to realize this environment, we make use of the characteristics of programming languages providing the reflection, the notion of allowing programs to refer and operate themselves or runtime environment of them, and the Interface Repository defined in CORBA, the server storing definitions described in IDL files.

The remainder of this paper is organized as follows: Section 2 outlines the development process of CORBA applications and discusses problems about stub and skeleton files. Section 3 propose one approach to resolve this problem: an runtime environment generating stub and skeleton codes on demand at runtime. Section 4 outlines the implementation of it by the Python programming language. section 5 discuss related work. The last section presents conclusions and future work.

PROBLEMS OF CORBA APPLICATION DEVELOPMENT PROCESS

Outline of Development Process

An application development process in CORBA starts at describing interfaces to distributed objects with the Interface Description Language (IDL). The definitions are stored into one or more files (called IDL files). If developers construct only clients accessing distributed objects, development process starts at obtaining existing IDL files containing definitions of interfaces to them.

Once IDL files are prepared, developer can execute the IDL compiler to generate files containing program codes called stubs or skeletons. Stubs are source codes necessary to operate distributed objects of which interfaces described in the IDL files. Definitions of interfaces can include user-defined types or constant values, and their definitions are also stored in IDL files. So stub files include source codes necessary to handle not only objects but also these types and constant values. Skeletons are source codes necessary to implement distributed objects of which interfaces described in the IDL files.

In general, stubs and skeletons are separately stored into files. The number of files generated by an IDL compiler varies in implementations of CORBA and target programming languages. An IDL compiler in a typical implementation of CORBA for the C++ programming language generates two header files and source files from one IDL file. One pair of header and source file contains stubs and another pair contains skeletons. In some implementations for C++, the IDL

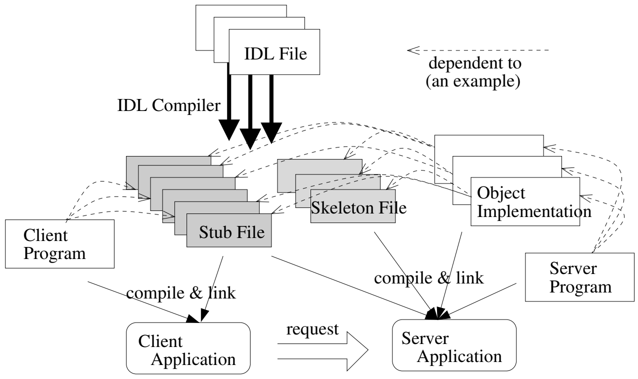

Figure 1: The development process in CORBA

compiler generates more files in order to separate elements necessary to clients and to servers strictly. In almost all implementations for the Java programming language, the IDL compiler generates many files from one IDL file. It generates three stub files by each interfaces or user-defined types defined in the IDL file and one skeleton file by each interfaces.

What developers have to write are following source files: implementations of interfaces described in IDL files, a server main routine which setup instances of distribute objects, and clients which use these distributed objects.

For building applications running actually, at first developers need to compile all of stub and skeleton files and source files written by them. Then, in using C++, developers link some of generated object files to generate each executable file of a server and a client (Figure 1). In using Java, there is no need to link, but developer need to deployment generated class files to each runtime environment of a client and a server.

Problems about Stubs and Skeletons

The procedure for building CORBA applications outlined in the previous section can be supported by a building system (such as *make*), based on rules described in advance about dependencies of files and building instructions (such as *Makefile*).

In CORBA application, the maintenance cost of such rules is much larger than one in normal applications by existence of stub and skeleton files. The addition of only one IDL file in C++ or only one definition in an IDL file in Java increases several participants, stubs & skeletons, in such rules, consequently brings many additions and modifications to them.

The approach of one large IDL file can reduce the maintenance cost to a certain degree especially in C++. But this approach increases the rebuilding cost extremely when the IDL file has been modified. The IDL compiler in almost all implementation always overwrites all files generated from the IDL file, even if the modification remains very small portion of it. Therefore all files should be recompiled. Moreover, a large IDL file is not suitable to reuse in other projects. So IDL files are usually partitioned by each interface.

In initial phases of software development, such as a prototyping phase in order to design interfaces properly, the cost to maintain building rules become a heavy load to developers because there are many changes related to IDL files in this phase. On developing only simple clients, developers can easily implement the client programs for the sake of stub files, while its building process is just like one in complicated software projects, so they often feel irrational.

Necessity of Stub and Skeleton Files

The origin of the problems is that stubs and skeletons take the form of source files previously generated from an IDL file. But this is quite essential in the following two reasons related to the power of description in general programming languages.

One: Incorporate Type Definitions In order to write a program accessing distribute objects, developers need to incorporate the class or/and type definitions corresponding to the interfaces and user-defined types in IDL files into the program. In most programming languages including C++ and Java, the only practical way to incorporate new definitions of types or classes into programs is to incorporate source programs defining them.

For example, If you implement a client to operate a distributed object having the **NameTableServer** interface defined in the IDL file shown in Figure 2, you have to incorporate the class or type definitions into the client program correspond to **NameTableServer** interface and the following types defined in the IDL file: the **VType** enumerated type, the **VValue** union type, and the **NotMatch** and **AlreadyExist** exception types.

```
module NameTable {
  enum VType { VT_BOOL, VT_INT,
                   VT_FLOAT, VT_STRING };
  union VValue switch (VType) {
    case VT_BOOL: boolean vb;
    case VT_INT: long vi;
    case VT_FLOAT: float vf;
    case VT_STRING: string vs;
  };
  exception NotMatch {};
  exception AlreadyExist { VValue v };
  interface NameTableServer {
    void insert(in string name,
                in VValue value,
                in boolean ifexist)
                  raises (AlreadyExist);
    void delete(in string name)
                  raises (NotMatch);
    VValue lookup(in string name)
                  raises (NotMatch);
  };
};
```

Figure 2: An example of IDL file

By making use of the incorporated classes or types, you can easily write the program to invoke the **insert** operation as shown in Figure 3

```
NameTable. NameTableServer srv;
...
String name = "Dummy";
NameTable.VValue val =
    new NameTable.VValue();
val.vi(1000);

try {
  a.insert(name, val, false);
} catch (NameTable.AlreadyExist ex) {
  System.err.println("'" + name
                     + "' already exists.");
}
```

Figure 3: Invocation the `insert` operation in Java

Two: Implement Operations Depend on Types Stub and skeleton files contain operations depend on the knowledge of both kinds of type definitions described in IDL files and mapped onto the target programming language. The following operations come under this category: the mutual conversion between a value of a user-defined type and its representation on a network stream, and mutual conversion between a method invocation and a request protocol.

By the way, CORBA defines the following three API sets to write generic applications independent of specific interfaces and user-defined types, such as debugging tools and bridges between different ORBs.

- The *Dynamic Invocation Interface* allows to invoke operations of distributed objects without corresponding stub files.
- The *Dynamic Skeleton Interface* allows to write implementations of distributed objects without corresponding skeleton files.
- The *DynAny API* allows to handle values of user-defined types without corresponding stub files.

These APIs certainly allow developing CORBA applications without stub and skeleton files. But they provide no uniformity between distributed objects and ordinary objects in programming languages, and require developers to write down operations depending on the knowledge of type definitions by hand, Therefore they are unsuitable for development of ordinary applications.

RUNTIME ENVIRONMENT GENERATING STUB ON DEMAND

For resolving the complexity of the building process, we propose the approach automatically generating and incorporating stubs and skeletons on demand at runtime. In our approach, There is no need of stub or skeleton files generated from IDL files in advance, so the absence of these files simplify the building process very much.

In order to realize this runtime generation, we make use of characteristics of programming languages allowing reflection, and the Interface Repository in CORBA as the repository of IDL files.

Runtime Stub Generation by Reflection

For generating stub or skeleton code at the runtime, it is essential to use programming languages which can generate and incorporate type or class definitions and the operations depends on these definitions.

Programming languages allowing the reflection can realize this feature. The reflection is the notion allowing a programming language to write such a programs as refer or operate the program itself or the runtime environment of the program.

The former ability operating the program itself, allows to write programs to refer or modify existing type/class definitions or generate new one. This ability of the reflection is distinguished as the *linguistic reflection* from the generic reflection [KMS98]. By making use of the programming languages supporting the linguistic reflection at least, the runtime generation of stub or skeleton codes can be realized.

Retrieve Interface Definitions from Interface Repository

The runtime generation of stub or skeleton codes needs the definitions of interfaces and user-defined types described in IDL files. In our approach, these definitions are retrieved not directory from IDL files but from the Interface Repository storing these files as shown in Figure 4.

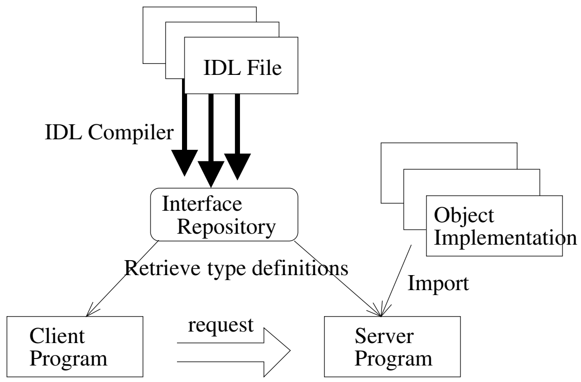

Figure 4: Retrieve type definitions from Interface Repository.

The Interface Repository is the server defined in the CORBA standard. It can store interface and user-defined types defined in IDL files in the form of an abstract syntax tree and exporting the nodes in the tree as distributed objects. Such nodes are uniquely identified by their *Repository IDs* derived from their full scoped name in the IDL file.

Most implementations of CORBA provide such an IDL compiler as not generate stub or skeleton files from an IDL file, but feed it to the Interface Repository. Syntactical errors in the IDL file are checked by the compiler in the same way as normal one, so the resulting abstract syntax tree constructed in the repository has no syntactical error.

Retrieving the definitions file from the repository can reduce the cost of parsing the file and detecting errors in it at runtime. Furthermore it can ensure that the client and server referring the repository to refer the same definition of a user-defined type or interface.

In case of using IDL files, a server and client can refer different versions of the same IDL file, consequently refer different definitions of the same interface or user-defined type. With luck, this inconsistency can be detected by system exceptions caused by it at runtime, otherwise this is hard to detect. So it is important to prevent this.

Stub Generation Based on IOR
In order to retrieve necessary definitions from the repository, it is necessary to know Repository IDs corresponding to these definitions. The easiest way is that developers specify necessary Repository IDs in client or server programs in advance, and the runtime environment refers them. But this way loads developers with the management of dependency among their programs and Repository IDs.

Our approach is to uses the type ID included in the object reference of the distributed object which a client program intend to access. In general, a client accessing a distributed object get the object reference of it by invoking appropriate APIs or querying the NamingService in CORBA at first. Almost all implementations of CORBA adopt the Interoperable Object Reference (IOR) standardized in CORBA as the way of representing object references. In the IOR, an object reference consist of one type ID indicating the type of the object referred by it, and at least one tagged profile indicating the location of the object. In most implementations, the Type ID is identical to the Repository ID of the interface definition.

Therefore, by investigating the object reference obtained by a client, the runtime stub generator can obtain the Repository ID of the interface which a client program intend to operate. Other definitions including user-defined types or other interfaces can be obtained by navigating through the repository with the ID as the starting point. Finally the stub generator generates all stub codes necessary for the client to operate the interface and incorporate them into the client. Now the client become to be able to send requests to the distributed object. Figure 5 illustrate this process.

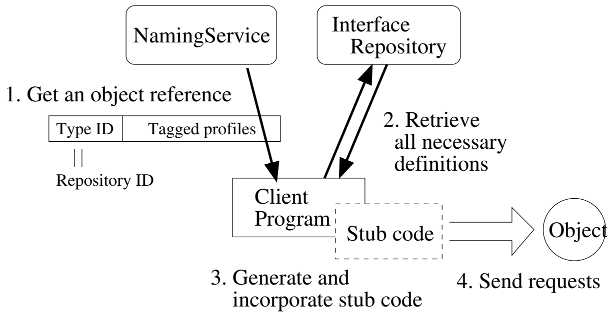

Figure 5: Generating stub codes based on IOR

Stub Generation on Demand
Among stub codes generated at the time when a client obtained a object reference, method definitions corresponding to operations of interfaces are not necessary until the client actually invoke these methods. Therefore, the generation of these methods can be deferred just after the client invokes them. If the client uses distributed objects of which interface supports many operations and it invokes only small part of these operations, this approach reduces the cost for generating methods.

This approach is realized by using another characteristic of programming languages allowing the reflection. As mentioned above, in such programming languages, programs are allowed to refer or operate the runtime environment of itself [1]. By this type of reflection, a program can modify behaviors of the runtime environment of the program itself. So in such programming language, the behavior of the language when an undefined method is invoked, can be modified not to cause the error signal, but to generate its definition and invoke it.

Improvement of Deferred Synchronous Invocation
The approach generating method definitions on demand is useful to improvement the language mapping of the deferred synchronous invocation in CORBA.

In CORBA, ordinary invocation of all operations except for ones specified as `oneway` in the IDL file is synchronous. It send the request of the specified operation to the target object and wait for the result of it returned from the object. CORBA also defines the deferred synchronous invocation. This type of invocation returns just after sending the request, and the program can receive the result later on.

But deferred synchronous invocation are possible only when the invocation is performed by the Dynamic Invocation Interface (DII). In DII, it is necessary to describe all steps for sending requests, and programs become very complicated. Figure fig:deferred synchronous shows the deferred synchronous version of the invocation shown in Figure 3.

The forthcoming version 3.0 of the CORBA standard is planed to include the new standard called CORBA Messaging [Obj98b] supporting two style of asynchronous method invocation. In this standard, deferred synchronous invocation [2] can be performed by ordinal method invocations in programming languages just like synchronous invocation.

This improvement is realized by the IDL compiler compliant to the standard. It generates definitions of methods specialized to deferred synchronous invocation. These methods are distinguished from normal ones by the prefix `sendp_` of their name.

In this approach, the IDL compiler has no means to know which combinations of operations and styles of invocation are actually used by clients. Therefore, it has to generate the definitions of all combinations of them. Consequently the size of stub files become very large. Moreover, most of method definitions are not used by clients.

In our approach, methods for deferred synchronous invocation can be generated when a client invokes methods of which name have the `sendp_` prefix. In this manner, a client program can perform deferred synchronous invocation by a

[1] This type of reflection is called the *behavioral reflection* [KMS98]
[2] called polling model in the standard

```
NameTable.NameTableServer srv;
...
String name = "Dummy";
NameTable.VValue v =
    new NameTable.VValue();
v.vi(1000);

// setup arguments and a return value
org.omg.CORBA.Request req =
    srv._request("insert");
req.add_in_arg().insert_string("Dummy");
org.omg.CORBA.Any v_any =
    req.add_in_arg();
VValueHelper.insert(v_any, v);
req.add_in_arg().insert_boolean(false);
// invoke in the deferred synchronous style
req.send_deferred();
...
...
// receive the response
req.get_response();
// check raised exceptions
org.omg.CORBA.ExceptionList exlist =
   req.exceptions();
if (exlist > 0) {
  omg.org.CORBA.TypeCode extcode =
    exlist.item(0);
  if (extcode.equal(
      NameTable.AlreadyExistHelper.type()))
  {
    System.err.println(
       "\'" + name + "' already exists.");
  }
}
```

Figure 6: Deferred synchronous version of Figure 3

method invocation just like synchronous invocation, and no unuseful methods can't be generated.

THE IMPLEMENTATION BY PYTHON

We implemented the runtime environment by making use of the Python programming language[vR99] and the implementation of CORBA for Python called Fnorb[Chi99]. In this section, we outline the current implementation of the environment.

Using Runtime Libraries of Fnorb

The current CORBA standard doesn't define the language mapping for CORBA. But there is an implementation of CORBA for Python called Fnorb [Chi99]. Fnorb is almost compliant to the CORBA standard; the Interface Repository is available, and object references are represented by IOR. Fnorb is implemented by Python except for the mutual conversion between primitive values and their representation on network streams. We currently implement the runtime environment by modifying runtime libraries for Fnorb.

Runtime Stub Generation in Python

Python supports the linguistic reflection. All elements which consist of Python programs such as modules, classes, methods, and so on, are values of built-in types. So programs can refer and modify them. Moreover there are methods for generating values of such built-in types. With the use of these methods, a program can generate new programs and incorporate them into itself. Programs can also generate new programs as character strings and incorporate them by the `exec` statement.

The current implementation of the runtime stub generator borrows stub generation codes of the IDL compiler in Fnorb and generate stubs as character strings and incorporate them into application by `exec` statements. This manner takes much runtime costs of spaces and times, should be replaced by smarter one in future.

The runtime stub generator is invoked in the unmarshal routine of the Interoperable Object Reference included in the runtime library of Fnorb. The unmarshal routine can extract a Repository ID from an object reference, and provide it to the generator.

On Demand Stub Generation

Python allows programs to customize the behavior when an undefined method is invoked. This customization is realized by defining the method of which name is `__getattr__` in a class. When an undefined method is invoked to an instance of the class, the Python interpreter call this special method instead of rising the error exception.

The generation of methods to invoke operations on demand can be realized by this special method. At first, The stub generator defines the special method in a stub classes corresponding to an interface. When a client invokes a method to an instance of the stub classes, Python interpreter calls the special method with the argument of the name of actually called method. The special method calls the stub generator to generate and incorporate the stubs for the method specified by the argument.

RELATED WORK

ILU

ILU (Inter-Language Unification) [JSLJ98] is the system with almost the same functionality as CORBA ORB. ILU is originally designed to support interaction between modules described by various programming languages, so supports more languages than CORBA.

ILU supports Python and its runtime library supports the runtime stub generation. In ILU, client programs can specify the name of an IDL [3] instead of the name of stub modules generated from it. In this case, the runtime library execute IDL compiler and generate stub files into a temporary directory, and incorporate them into the client program.

The runtime cost in this approach much larger than one in our approach because of the cost of creating the process for the compiler, parsing IDL files, and writing stub files. The advantage in this approach is to automatically execute the IDL compiler. The runtime library interrupts the client process,

[3] called ISL (Interface Specification Language) file in ILU but recent versions of ILU also support CORBA IDL

when the IDL compiler detects a syntactical error in the IDL file.

CorbaScript

CorbaScript [LO98] is the object-oriented scripting language specially designed for development of CORBA applications. The runtime environment of CorbaScript is highly integrated to ORB. The structure of name spaces in CorbaScript are directory mapped onto one in IDL. If client programs access an undefined name in the name space in CorbaScript, the runtime environment access the Interface Repository with the full scoped name of the undefined name, and retrieve the definition of the names.

By this approach, the application development process in CORBA become very simple on the same level with our approach. However, there is few class libraries because CorbaScript is a newly developed language. Our approach has no such problem because we make use of existing programming language supporting the reflection.

LuaORB

LuaORB is the runtime library to integrate the Lua programming language and an implementation of CORBA ORB. Lua support the reflection, and allows programs to customize the behaviors of the language. LuaORB customize the behavior of LUA in accessing undefined methods. In the customized behavior, LuaORB accesses the Interface Repository to gathering necessary definitions, and generates a request and send it to distributed objects with the DII provided by the ORB. In LuaORB, developer must explicitly specify the full scoped names of the interfaces necessary for a client.

LuaORB can't provide stubs necessary for client programs to manipulate user-defined types in arguments of operations because it accesses the repository just after clients invoke methods corresponding to the operations. In LuaORB, it is troublesome to handle such user-defined types as highly structured or involving constant values. In our approach, the runtime library generates and incorporates stubs for user-defined types just after clients obtain object references, client programs easily handle these types.

CONCLUSIONS

In this paper, we discussed the organization of runtime environment for CORBA applications to resolve the complexity of development process especially of its building process. This runtime environment can generate stubs/skeletons on demand and incorporate them into applications at runtime. Therefore, developers can proceed development of CORBA applications without trouble about management of stub/skeleton files.

In order to realize this environment, we make use of the following characteristics of programming languages supporting the reflection: the linguistic reflection and the customization of the language behavior. And we make use of the Interoperable Object Reference and the Interface Repository defined in the CORBA standard.

In this paper, we have mainly discussed only the client side of the environment. The detailed design and implementation of the server side remains in future work. By the way, we make use of Python as a programming language supporting reflection at present, but we are going to implement the same environment by the Java programming language because several approaches [KMS98, LB98, Chi98] introducing the notion reflection into Java have been reported.

ACKNOWLEDGMENTS

We greatly appreciate useful discussions by Mr. Kumagai and Mr. Higashida of PFU Limited. This work was partly supported by Grant-in-Aid for Scientific Research (c) #10680345, and Grant-in-Aid for Priority Areas (A)(1) #09245105, Ministry of Education, Science, Sports and Culture, Japan.

REFERENCES

Chi98. Shigeru Chiba. Javassist — a reflection-based programming wizard for java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

Chi99. Martin Chilvers. *fnorb version 1.0*. CRC for Distributed Systems Technology, February 1999. http://www.dstc.edu.au/Fnorb.

JSLJ98. Bill Janssen, Mike Spreitzer, Dan Larner, and Chris Jacobi. *ILU 2.0alpha14 Reference Manual*. Xerox Corporation, 1998. ftp://ftp.parc.xerox.com/pub/ilu/2.0a14/manual-html/manual_toc.html.

KMS98. Graham Kirby, Ron Morrison, and David Stemple. Linguistic reflection in java. *Software: Practice and Experience*, 28(10):1045–1077, 1998.

LB98. Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA'98 Proceedings*, October 1998.

LO98. Laboratoire d'Informatique Fondamentale de Lille and Object-Oriented Concepts, Inc. *OMG CORBA Scripting Language RFP Revised Submission*, December 1998. OMG TC Document orbos/98-12-09.

Obj98a. Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.2*, February 1998. formal/98-02-01.

Obj98b. Object Management Group. *CORBA Messaging*, May 1998. OMG TC Document orbos/98-05-06.

vR99. Guido van Rossum. *Python Reference Manual*. Corporation for National Research Initiatives(CNRI), release 1.5.2 edition, April 1999.