Just-In-Time Stub Generation

Markus Hof

Department of Computer Science (System Software)
Johannes Kepler University Linz, Austria
hof@ssw.uni-linz.ac.at

Abstract. In distributed object systems, one generates local surrogate objects to achieve transparent remote method invocations. These surrogates intercept method invocations, transfer the invocations to the actual (remote) object, and invoke the respective method by using so-called stub code. We describe a method which automatically generates surrogate and stub code. The actual generation is delayed until run time, which allows late adaptations to current needs and restrictions. Objects using this mechanism are not necessarily derived from a common base class.

1 Introduction

Today's highly interconnected systems put more and more emphasis on the exploitation of the advantages inherent to a network, i.e. increased fault tolerance, better availability, and easier scalability. However, network systems have their disadvantages as well, and it is not easy to actually exploit their advantages. Independent failure modes, which have to be handled when dealing with several computers, increase the complexity of software development. Additionally, networked systems are often heterogeneous and highly dynamic. The configuration of available computation resources may change on a moments notice. To cope with these problems different approaches have been proposed. A common approach is to put part of the additional complexity into the object system, i.e., to hide it from the developer, by extending the notion of objects and classes.

The purpose of the work described in this paper is to add support for distributed objects to the Oberon system [Rei92]. In particular, distributed objects should support the implementation of distributed models (in the sense of the MVC paradigm [KrP88]), i.e., they should support the adaptation of existing Oberon applications (ease the transition of existing MVC applications) to a distributed environment. To ease this transformation we proclaim automatic just-in-time generation of surrogate and stub code. Generating stub code not in advance, but only on demand, allows the system to adapt the generated stub and surrogate code to current circumstances (e.g., recent changes in the network topology). Additionally, delaying code generation as long as

possible allows one to distribute objects, which are not aware of the network and the notion of distributed objects.

1.1 Overview

A client sees an object as a reference into memory, some data fields, and a set of type bound procedures (methods). An application does not have to distinguish between local and remote objects (see Figure 1). Different access methods are handled transparently by the distributed object system.

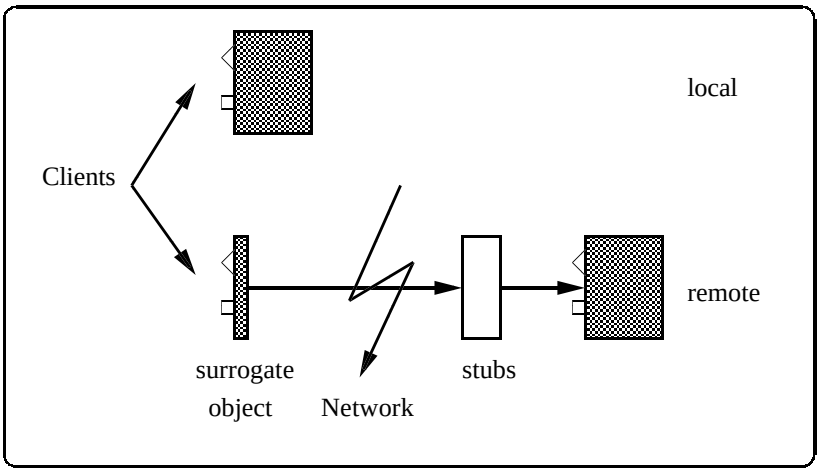

Figure 1: View on objects as seen by clients

The application has transparent access to all objects regardless of their actual location. Oberon's strict type checking is still enforced and run-time type information is still available. For every accessed remote object the system automatically generates a so called *surrogate object*. A surrogate object is the local representative (placeholder) of an object located on another site. It offers exactly the same interface as its associated actual object, but redirects incoming requests to the actual object. The request (object ID, invoked method, and actual parameters) is transformed (marshalled) to a byte stream which is sent from the surrogate to the stub. This stream includes all information needed to reconstruct the receiver object, the called method, and the actual parameters. This mechanism is similar to the RPC mechanism [BiNe84, Tan95], except that a receiver object is passed along with each new invocation.

For every method there is a surrogate part and a stub part, as each method has its own individual interface (signature). The surrogate part

- offers the same interface as the actual method,
- transforms the incoming data into a byte stream,
- transfers the stream to the corresponding stub code,
- waits for the result,
- and returns the resulting data to the client.

The respective stub

- receives a byte stream,
- reconstructs the streamed parameters,
- calls the method of the actual object, while handing on the reconstructed parameters,
- converts the results into a byte stream,
- and sends the stream back to the surrogate.

In order to put records and even complex data structures into a byte stream, the stub and surrogate code uses a so-called *linearizer*. Linearizers convert arbitrary data structures into machine-independent byte streams and vice versa.

1.2 Just-In-Time Stub Generation

Generation of stub and surrogate code can be highly automated. The common approach, e.g. chosen in CORBA [COR95] or Network Objects [BiN94], is to generate stub and surrogate (also called skeleton code) statically, in advance, previous to the compilation of the client code. We delay the generation as long as possible and generate the needed code dynamically on demand. The advantages of dynamic code generation over static code generation are:

- Using dynamic code generation, different variants of surrogates and stubs can be generated depending on current needs of the application (e.g., parameters could either be transformed using deep copy or shallow copy).
- If surrogates and stubs are only generated at run time they need no space on the disk.
- Statically generated stub code is no longer portable and has to be generated for each new platform.

2 Implementation

The implementation consists of two parts. First, the generic marshaller, which is responsible for the linearization of parameters, and second, the stub and surrogate generator. This paper mostly elaborates on the second part, but gives a quick overview over the marshalling mechanism.

The generated code actually works as a framework. The stub and surrogate code itself does not contain any dependencies on distribution, but acts only as a general-purpose redirection mechanism. Distributed objects is just one possible application of this scheme. Other applications might be a method invocation logging facility.

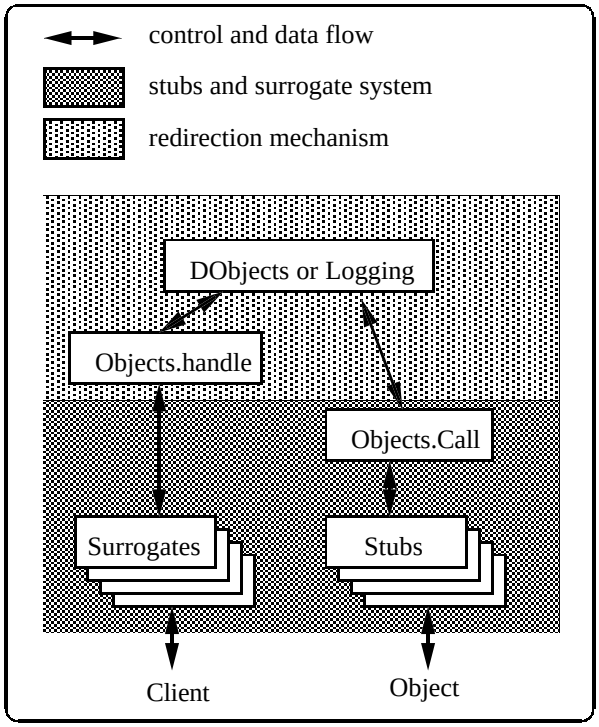

Figure 2: Overview of redirection mechanism

When a surrogate method is invoked, it transforms its parameters to a byte stream and passes this stream to the currently installed redirection mechanism in *Objects.handle* (up-call) (see Figure 2). The redirection mechanism may transfer the stream over a network, log the method invocation, or do anything else. If the stream is transferred over the network, the receiving site passes it to a procedure called *Objects.Call* which is part of the stub system. *Objects.Call* reads the method identifier from the stream and invokes the appropriate stub, which then calls the actual method. *Objects.Call* can invoke arbitrary stubs, as they all have the same interface.

The installed redirection mechanism is completely hidden from the client and from the implementation of the invoked object. Therefore, one can freely swap between different mechanisms invalidating neither the client's nor the object's implementation.

2.1 Marshalling

The marshalling mechanism consists of a manual part and an automatic part. For the manual part, the programmer has to write a marshaller for every structured type used as a parameter or receiver type. This marshaller is activated whenever an instance of this type is marshalled or unmarshalled. It has to decide which instance variables are written to or read from the stream.

The automatic part of the marshalling mechanism takes care of the inter-record dependencies. It stores information about inter-record dependencies together with the actual record data as stored by the individual marshallers. This information allows the restoration of arbitrary data structures (trees, circular lists, ...).

All data stored to a stream is automatically converted into a well-defined byte- and bit-ordering, i.e. the resulting stream is platform independent. Data read from a stream is automatically converted back to the local processor specific ordering.

2.1.1 Manual Marshallers

A marshaller consists of a sequence of calls to the linearizer for each record field to be read or written. The mechanism is completely orthogonal with respect to writing and reading, i.e. the same calls are executed while reading from or writing to the linearizer. Therefore we only need one marshaller per type (see Figure 3), which is used for writing to, as well as for reading from a stream. The linearizer offers calls for every basic Oberon type, e.g. INTEGER, BOOLEAN, PROCEDURE, ...

```
TYPE
   MyPtr = POINTER TO MyPtrDesc;
   MyPtrDesc = RECORD
      i: INTEGER;
      r: REAL
   END;

PROCEDURE Marshall (lin: Linearizer.Linearizer; o: SYSTEM.PTR);
VAR obj: MyPtr;
BEGIN
   obj := SYSTEM.VAL (MyPtr, o);
   lin.Integer (obj.i);
   lin.Real (obj.r)
END Marshall;
```

Figure 3: Example marshaller for type *MyPtr*

Special considerations:

- A marshaller for a given type *T* does not have to care about possible base types of *T*. They are handled automatically and the correct run-time type is restored.
- For special data structures it is possible to distinguish between reading and writing, e.g. when marshalling a reference to a font (see Figure 4), one may only write the name of the font instead of the complete font information:

```
TYPE
   MyPtr = POINTER TO MyPtrDesc;
   MyPtrDesc = RECORD
      f: Fonts.Font
   END;

PROCEDURE Marshall (lin: Linearizer.Linearizer; o: SYSTEM.PTR);
VAR obj: MyPtr;
BEGIN
   obj := SYSTEM.VAL (MyPtr, o);
   IF lin.writing THEN lin.String (obj.f.name)
   ELSE lin.String (name); obj.f := Fonts.Font (name)
   END
END Marshall;
```

Figure 4: Marshaller distinguishing read and write

- When storing or retrieving pointers, a marshaller has two possibilities. It may either perform a deep copy or a shallow copy (see Figure 5). A deep copy of an object *x* duplicates object *x* onto the stream. Objects referenced by *x* are handled as defined in the marshaller handling x. An independent copy of *x* is generated, whenever the byte stream is unmarshalled. This may be slow and quite space consuming. A shallow copy of object *x* puts only the object identifier of *x* onto the byte stream. This is fast and needs just a few bytes. However, the program, which unmarshalls the stream, has to cope with incoming object identities and create a suitable surrogate object.

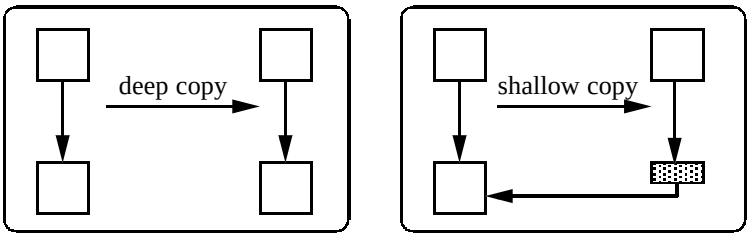

Figure 5: Deep versus shallow copy

2.1.1 Automatic Marshalling

If a parameter represents a data structure (e.g., a list or a graph), the whole data structure is saved and restored automatically. Object identities are saved onto the stream, i.e., references to the same object will still reference the same object after the data structure has been restored. The linearizer achieves this by putting each object onto the stream only once. If an object is referenced for the first time, its run-time type and data is written into the stream. If the same object is referenced again, the linearizer stores only an identifier, which allows the restoring process to identify the actual - previously restored - object. This allows marshalling of arbitrary data structures, as well as reducing the size of the resulting byte stream.

2.2 Just-In-Time Stub Generation

Stub and surrogate code are generated on demand, i.e., if either an external object is requested or a local object is made public. The code is generated, loaded, and discarded as necessary. This allows late adaptation to current needs and to recent restrictions.

Our system offers two choices when implementing a new class. First, one may choose for each method, wether it is executed locally on the surrogate object, or wether the request is forwarded to the actual object. Second, one may influence the mode in which parameters are passed.

Our stub generator needs three different kinds of information:

- Which methods should be executed on the surrogate object and which on the actual object. This information has to be supplied by the developer.
- Each parameter with a pointer type is - by default - passed by making a deep copy. However, in order to achieve a shallow copy, it is possible to change this behaviour for individual parameters. This information has to be supplied by the developer.
- The complete interface of the object's type. Our current stub generator uses the corresponding source file as input. However, using existing metaprogramming facilities which offer the complete reference information [Tem94, StMö96] of all types, would allow us to distribute objects without making the corresponding source code public.

The stub generation is platform independent. It only requires an Oberon compiler using the portable front-end OP2 [Cre90].

As an example let's look at the following type definition (Figure 6):

```
TYPE
   MyPtr = POINTER TO MyPtrDesc;
   MyPtrDesc = RECORD
      i: INTEGER;
      PROCEDURE (obj: MyPtr) Method (VAR i: INTEGER; p: MyPtr)
   END;
```

Figure 6: Example type definition

The stub generator will generate two Oberon procedures (surrogate method and stub procedure) (see Figure 7). As one can see, both stub and surrogate code, mainly linearize and restore parameters (see Figure 7). The surrogate code makes an up-call to *Objects.handle* in order to redirect the method invocation. It passes as parameters the receiver object, the streamed actual parameters, and a method identifier. The method identifier is a unique constant value, which identifies the invoked method. This constant is generated automatically by the stub generator.

```
PROCEDURE (obj: MyPtr) Method (p: MyPtr) : INTEGER;
VAR lin: Linearizers.Linearizer; s: Linearizers.Stream; retVal: INTEGER;
BEGIN
   lin := Linearizers.NewWriter ();
   lin.Ptr (p);
   s := lin.Stream ();
   s := Objects.handle (obj, s, xxx); (*  xxx is a method identifier *)
   lin := Linearizers.NewReader (s);
   lin.Integer (retVal);
   RETURN retVal
END Method;

PROCEDURE Stub (o: SYS.PTR; VAR data: Linearizers.Stream);
VAR lin: Linearizers.Linearizer; obj, p: MyPtr; retVal: INTEGER;
BEGIN
   obj := SYS.VAL (MyPtr, o);
   lin := Linearizers.NewReader (data);
   lin.Ptr (p);
   retVal := obj.Method (p);
   lin := Linearizers.NewWriter ();
   lin.Integer (retVal);
   data := lin.Stream ()
END Stub;
```

Figure 7: Generated stub and surrogate code

Actually, the shown source code (see Figure 7) is never generated in this form. The generator directly creates an abstract syntax tree, as it is created by OP2 [Cre90]. This syntax tree is then translated to code by the compiler back end. Generating the syntax tree instead of human readable source code is simpler and faster, since no scanner and parser has to be used during code generation.

Recent work [FrKi96, Java95] shows advantages of just-in-time compilation (e.g., platform independence). However, delaying not only the actual code generation, but also allowing late parameterization has some additional advantages:

- The stub generator is sufficiently fast and runs only once per type. There is no significant speed penalty when using delayed code generation. The generated stub and surrogate code remain valid for the lifetime of their respective type.
- The stub generator is small and simple (approximately 400 lines of source code).
- The generator is system independent, i.e., it runs on every computer equipped with an OP2 Oberon compiler.
- The stubs and surrogates are platform independent, since the actual coding into machine instructions is done at run-time.
- Since there are no persistent files (interface definition, stub sources, stub object files, ...) there are no version conflicts.

- The development of clients is independent of the stub generator (no stub headers have to be included).
- It is possible to adapt stub and surrogate code at run-time.

3 Related Work

Other stub generators require the source code generation to be done before the application is developed. The application includes information about the stubs and the surrogate objects (e.g., using include files in the CORBA C++ mapping [COR95]).

We don't build a new system with a new language, as e.g. Emerald [JuLe88], but emphasize the reuse of existing systems. Similar to the network objects [BiN94] we primarily try to keep our system simple without discriminating its generality. The main difference between our system and other systems is that we delay stub generation until run time. Contrary to the network objects we support parameters of type PROCEDURE, i.e. one may initiate up-calls on remote systems.

4 Conclusions

Our system offers a simple, fast and flexible redirection mechanism for method invocations. It is intended for supporting distributed objects with transparent remote method invocations. However, the mechanism is more general and can be used to implement other applications, e.g. logging of method invocations.

By delaying stub generation until run-time, we achieve a higher degree of flexibility than similar systems, which force the programmer to generate stub and surrogate code before the actual application is developed.

Some important points have not yet been addressed in our distributed object system. Currently, we work on the implementation of distributed garbage collection. Additionally, the Oberon language has, unfortunately, no exception handling. We therefore have difficulties to offer an elegant solution for handling exceptions due to the redirection of the method invocation, e.g. broken connections. Currently we handle these problems using a state associated with each surrogate object, which indicates earlier failures.

5 References

[BiN94] Birrell A., Nelson G., Owicki S., Wobber E.: Network Objects. SRC Research Report 115, 1994

[BiNe84] Birrell A., Nelson B.: Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, vol. 2, Feb. 1984

[COR95] The Common Object Request Broker: Architecture and Specification. OMG, Revision 2.0, July 1995

[Cre90] Crelier R.: OP2: A Portable Oberon Compiler. Institut für Computersysteme, Report 125 Februar 1990, ETH Zürich

[FrKi96] Franz M., Kistler T.: Slim Binaries. University of California Irvine, Technical Report 96-24

[Java95] The Java Virtual Machine Specification. Sun Microsystems, Release 1.0, August 1995

[JuLe88] Jul E., Levy H., Hutchinson N., Black A.: Fine-grained mobility in the Emerald system. ACM Transactions on Computer Systems, 6(1):109-133, 1988

[KrP88] Krasner G., Pope S.: A Cookbook for Using the MVC User Interface Paradigm in Smalltalk. Journal of Object Oriented Programming Aug./Sep. 1988

[Rei92] Reiser M., Wirth N.: Programming in Oberon - Steps beyond Pascal and Modula-2. Addison-Wesley 1992

[StMö96] Steindl C., Mössenböck H.: Metaprogramming Facilities in Oberon for Windows an Power Macintosh. University of Linz, Department for Systemsoftware, Report 8, July 1996

[Tan95] Tanenbaum A.: Distributed Operating Systems. Prentice Hall 1995

[Tem94] Templ J.: Metaprogramming in Oberon. PhD thesis, Swiss Federal Institute of Technology (ETH Zürich), 1994, Number 10655