Late Adaptation of Method Invocation Semantics

Regular Talk

Markus Hof

Department of Computer Science (System Software)
Johannes Kepler University Linz, Austria
hof@ssw.uni-linz.ac.at
Fax: ++42-732-2468-7138

Abstract. In distributed object systems, one has the possibility to make method invocations on objects located on other host. During such an invocation, data is sent to another host and back. However, the system tries to hide this and simulate a standcard method invocation as close as possible. Some systems [Voyager] try to offer other invocation semantics, e.g. asynchronous method invocation.
We try to go a step further and offer the actual invocation as first class abstractions. The programmer can build his own abstractions by either implementing his own or by combining existing abstractions. With this system, he can build arbitrary invocations semantics, e.g. synchronous method invocation with transactional semantics, which also logs all method invocations.

1 Overview

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.

However, distributed object systems, e.g. Object Management Group's OMG [OMG], Microsoft's DCOM [Micro], or JavaSoft's Remote Method Invocation [RMI98] use a fix scheme of a point-to-point request/response communication model. While appropriate for a subset of applications using distributed objects, this model inhibits exploiting the advantages of distributed objects for other domains. Other work has been done to widen the application domain for distributed objects by introducing new kinds of method invocation semantics, e.g. Voyager [Voyager] which introduces asynchronous method invocation.

This paper describes a novel approach to widen the application domain for distributed objects even further. We claim that introducing new special cases as done e.g. in Voyager is not sufficient. There are an infinite number of possible desirable kinds of method invocation, e.g. asynchronous vs. synchronous, unicast vs. multicast, replicated, transactional, logged, or atomic. We claim that just as any other aspect of a distributed system, the "invocation style" should be a first class abstraction. One should

be able to compose abstractions and use the most adequate ones according to the application needs.

We propose a generalization of the stub and skeleton code. In existing distributed object environments the programmer defines the interface and the corresponding stub and skeleton code is generated at compile time. In order to generalize the code generation, we delay it until run-time. Different access methods are handled transparently by the distributed object system.

1.1 Distributed Object Methology

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).

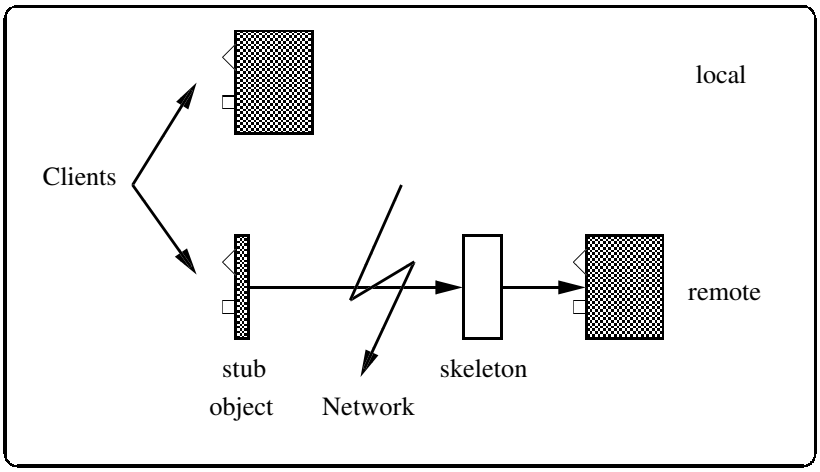

Figure 1: View on objects as seen by clients

The application has transparent access to all objects regardless of their actual location. For every accessed remote object, the system automatically generates a so-called *stub object*. A stub 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 stub to the skeleton. 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.

1.2 Code Generation for Stub and Skeleton

Stub and skeleton code is generated automatically from the interface definition of the given class. Typical stub and skeleton code consists of three parts:

- Marshalling of all input parameters
- Activation of the transport mechanism in order to signal the actual object the intercepted method invocation.
- Unmarshalling of output parameters and the return value

Logically seen we introduce one new additional layer (see figure 2). A method invocation, which is not handled locally, is intercepted by its corresponding stub method. Each stub method is tailored to its method and is mainly concerned with marshalling. After the marshalling is done, the stub code calls, regardless of the

invocation mode, the global invocation handler. This handler chooses and activates the previously assigned invocation mode. The structure of the invocation modes is explained in the next section.

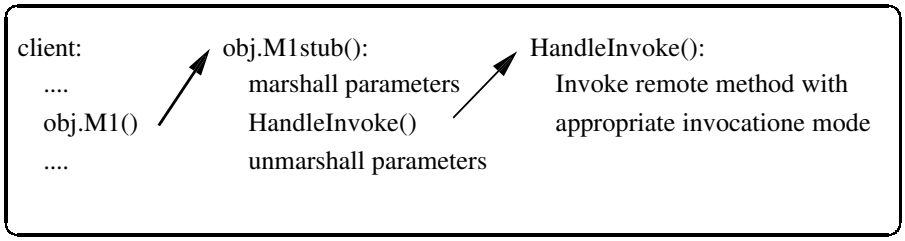

Figure 2: Control flow on method invocation

In our current prototype implementation, we achieve this behaviour without introducing a new layer by using an array of invocation abstractions. Each stub knows the method to be used and chooses the correct one through an index into an invocation array. With help of this mechanism we avoid the additional layer and achieve a faster dispatch.

2 Generating Invocation Modes

We offer the programmer a class hierarchy of invocation abstractions (see figure 3). When an object is to be exported (made public to other hosts). The programmer must specify the desired invocation modes for every method of the object he exports. One can have an individual invocation configuration for each object of a class or reuse a configuration for all objects of the same class.

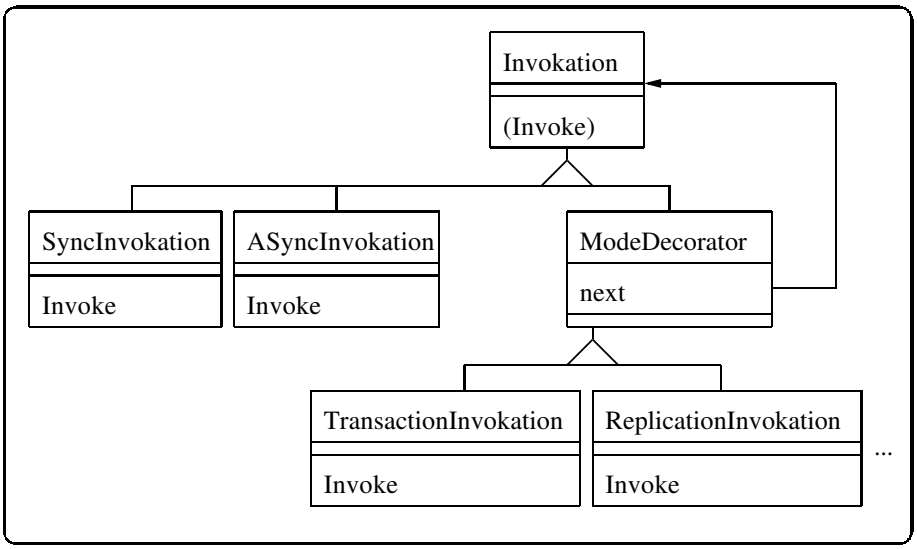

Figure 3: Hierarchy of invocation abstractions

To export an object one has to call the procedure *Export* that is part of a library. One has to specify the host on which the given object is exported, the name of the object and the desired invocation abstractions. As a result of this operation, the system generates 'on-the-fly' the necessary skeleton code to access this object (see figure 4).

```
invoke := Invokations.GetClassInfo("ClassName");
// ... modify abstractions to my current needs
Export(object1, host, name1, invoke);
// ... modify abstrations if neccessary
Export(object2,host, name2,invoke);
```

Figure 4: Exporting an object

When a client imports an object, it calls the procedure *Import*. One has to specify the name the object and the host where it resides. An appropriate request is sent to the server host. The server host sends back two kinds of information. First, the invocation abstractions of the exported object and second, the actual object data. With help of the received invocation abstractions the necessary stub code is generated and the actual object is generated (see figure 5).

```
Import(obj, host,name);
```

Figure 5: Importing an object

The necessary invocation information is generated with a call to *GetClassInfo*, which uses meta-programming facilities to collect it. It returns the default invocation information for a class. If an object is exported with this information, one gets the following default behaviour:

- Methods are called synchronous with the standard semantics of method calls.
- Parameters of a pointer type are copied using deep-copy semantics.
- If a method returns a pointer value, the referenced object is copied to the caller.

These three standard behaviours can be changed as described in the following three sections

2.1 Changing the Invocation Mode

By default, all method invocations are handled as standard method invocations, i.e. synchronous. However, one can change the behaviour as needed by composing your own invocation abstractions. Either one can create a new abstraction that suits the current necessities, or one can compose one with help of existing abstractions using the decorator pattern [Gamma95]. Let¥s look at some examples:

- A asynchronous invocation abstraction, which uses replication and logs the invocations:

```
VAR
   myInvoke: Invocation;
myInvoke := LogMode(ReplicationMode(ASyncInvocation()));
```

- A synchronous invocation with transaction semantics:

```
VAR
   myInvoke: Invocation;
myInvoke := TransactionMode(SyncInvocation());
```

After generation of the desired invocation abstraction one can assign it to the desired method(s) and assign it to an exported object:

```
invoke.Method("name of method", myInvoke);
Export(obj, host, name, invoke);
```

If one wants to implement an own transaction invocation one has to create a new subclass of the class *Invocation* and overwrite the method

PROCEDURE (inv:Invocation) Invoke(obj:PTR; id:LONGINT; s:Stream):Stream;

The method will be called whenever a method that uses this invocation abstraction is activated. *obj* is the invoked method, *id* contains a unique number defining the called method and *s* contains the marshalled parameters. The method has to return the linearized return value and output parameters.

2.2 Changing the copy-mode for individual parameters

Method can have pointer parameters. This implies, if the method is executed remotely, that the referenced objects have to be transferred from the client to the server and back. Either one can make a deep copy, actually generating a copy of the referenced object on the other host, or one can make a shallow copy. A pointer parameter copied in shallow copy mode is not transferred to the server. Instead, the object is automatically exported with an anonymous name. On the server side, before the server method is invoked, a corresponding import statement is executed automatically (see figure 6).

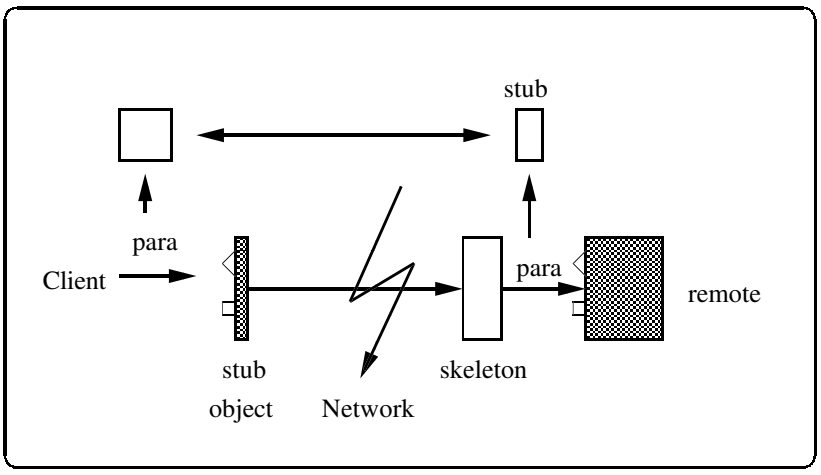

Figure 6: Shallow copy of parameter

2.3 Changing the copy mode of the return value

Methods can return pointer values. This implies, if the method is executed remotely, that the referenced object has to be transferred from the server to the client host. As with pointer parameter (section 2.2) one has the possibility to make a deep or a shallow copy, i.e. the procedure returns either the actual object (deep copy) or another stub object (shallow copy).

3 Conclusions

A method invocation that is not handled locally is intercepted by its corresponding stub method. Each stub method is tailored to its method. The stub is concerned mainly with marshalling. The actual invocation is delegated to the procedure *HandleInvocation* (see figures 1 and 2). When called, HandleInvocation decides on the actually used invocation mode:

```
HandleInvocation (rec: PTR; id: LONGINT; data: Stream) : Stream;
    info :=  ... Invocation information for the object rec
    invoke := invoke mode for method id in info
    data := invoke.Invoke(rec, id, data)
    RETURN data
```

An actual implementation of Invoke will do some invocation specific statements (open/close transaction...) and delegate the invocation to the decorated invocation mode, e.g. for the invocation mode resulting from the statement

```
invoke := LogMode(TransactionMode(SyncInvokation()));
```

the actual sequence of invocation modes is as shown in figure 7.

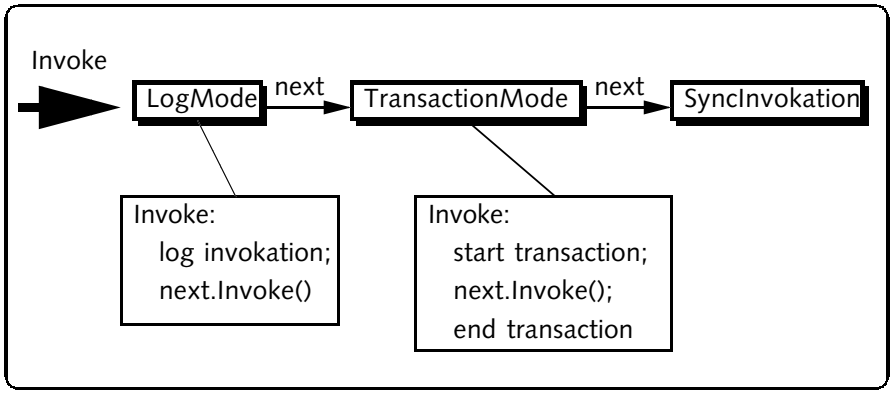

Figure 7: Example abstraction sequence

The invocation mode is defined only at runtime when an object is exported. Each time one exports an object, one can choose other invocation abstractions.

4 References

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

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

[Micro] Windows DNA, http://www.microsoft.com/dna/default.asp

[OMG] Object Management Architecture, http://www.omg.org/library/oma1.htm

[Voyager] Objectspace Voyager, http://www.objectspace.com/voyager

[RMI98] Remote Method Invocation specification, http://java.sun.com/producs/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

[Gamma95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 1995