Custom Invocation Semantics and Corpuses in Data Multicasting*

An application of custom invocation semantics in object-oriented middleware

Attila Ulbert
Department of General Computer Science
Eötvös Loránd University, Budapest, Hungary
mormota@elte.hu

February 24, 2003

Abstract

The widespread application of object oriented frameworks for distributed programming lead to their penetration into such areas, where they were not intended to be used before. However, these frameworks basically use synchronous interactions and at most a small and fixed set of other invocation semantics; and this is very restricting and even penalising, especially in case of multimedia and mobile communication. In order to comply with the requirements bound to these new areas the framework should allow the developer to extend the circle of applicable invocation abstractions and enable them to exploit the special characteristics of the networking/computing environment and the problem.

In this paper we present our extensible architecture through the problem of multicast group communication. Our ORB(M) framework allows developers to implement their own semantic elements and deploy various new methods bound to the process of remote invocation. The invocation semantics are composed of semantic elements supplied by the developer or the framework itself, thus the framework gives us the freedom to replace and arbitrarily customise the invocation semantics used by a given method. Illustrating our approach and solving the problem of group communication we extend the library of semantic elements and introduce a class template called Corpus, assess the applicability of the new elements and measure the performance of our prototype system.

Keywords: distributed communication frameworks, invocation semantics, multicast communication

Introduction

Most of the distributed object frameworks [OMG01a] are not suitable for multimedia communication currently. Although the concepts of multimedia can be modelled by OOP concepts, modelling elements requiring high performance or special communication pattern can easily come up against the intrinsic logics of distributed object programming. For example, consider the problem of video streaming or broadcasting. Although the Object Management Group [OMG] has already released its Audio/Video Stream Specification [OMG00], this specification doesn't serve OOP just builds upon it and implements a stream-based solution.

Even if the common Object Request Brokers are not developed for multimedia or wireless communication [OMG98], they can be enabled them to meet the new challenges by allowing the user to extend the framework itself [SCH98, HOF99]. Our ORB(M) system [ULB01] is strongly based upon the concept of invocation semantics and its building blocks, called semantic elements. A semantic element is an object which implements a fundamental step of the remote method invocation. Such a fundamental step is the creation and the interpretation of network messages or the network communication.

The semantic elements form an extensible class library [LEA97]. The developer can easily supplement the library with elements investing the middleware with such capabilities that exploit the knowledge they possess about the problem they are solving. The framework allows the user to determine or change these elements statically (at compile-time) or even dynamically (at run-time). Of course it does not mean, that the user has to choose all of the necessary semantic elements, he can use the default elements provided by the IDL compiler.

The ORB(M) framework offers the user the possibility of high-level transparency, but allows the developer to exploit the potential advantages of the low-level transparency. The default invocation semantics provide transparent communication, however the developer can implement semantic elements requiring

*This research has been partially supported by OMFB 01548.

additional assistance of the user in order to achieve high performance or unusual communication characteristics of the problem they solve.

We implemented the prototype ORB(M) framework in *C++*. The system already includes a fairly large library of semantic elements, implementing synchronous, asynchronous, oneway and iterated invocation, TCP and UDP based oneway and twoway communication.

The paper is organized as follows. Section 1 introduces our "guinea-pig" of multimedia communication. Section 2 presents the concept of corpus. Section 3 explains the implementation of group communication infrastructure in our ORB(M) framework. Section 4 introduces the use of corpus and the new session managers by modifying the invocation semantics of a given method. Section 5 discusses the applicability of the corpus and the group communication infrastructure and measures their performance in ORB(M). Finally, Section 6 concludes gives some concluding remarks.

1 Data multicasting in object-oriented distributed systems

Consider the task of video broadcasting: we have to broadcast the (video) frames generated by a server among a number of clients[1]. In this example the clients want to get the frames, but they also need some additional information before starting to receive them: the maximal frame rate. The problem is expressed by the following interface:

```
interface Station {
  attribute Frame frame;      // the last video frame
  unsigned short frameRate(); // the maximal frame rate (additional information)
};
```

Attribute `frame` of interface `Station` is refreshed by the server with the newly generated frame and method `frameRate` returns the maximal frame rate. Theoretically, the problem of multicasting the newly generated frames could be solved by calling the *get* method of attribute `frame`. However, issuing RMIs by the clients accessing the attribute is far from perfect solution as it can overload the system and most importantly, the clients do not know when the attribute is refreshed.

Therefore, clients cannot initiate the RMI as usual, but the server has to play the role of actor and clients have to passively wait for the new frames. Developing the thought of *reversed roles*, the *get* and the *set* method of the attribute should not issue remote invocations on the client, but act like local methods. Method `frameRate` is less problematic; since the clients usually call this method only once, the clients can invoke it remotely.

An abstract solution of the problem is based on attribute sharing [MG99]. Unlike the CORBA Notification Service, this solution does not require the user to implement service-specific abstract methods, the user only have to customise the invocation semantics of the methods of interface `Station` (see Section 4). According to our abstract solution, the attribute is shared among the clients and the server in a special way: whenever the attribute is changed by the server it must be changed on all of the clients as well, but whenever the attribute is changed on a client, the attribute is left unchanged on the other clients and the server. Note that because of the one-way synchronisation (server to client) the *get* method can be removed from the interface. Having this kind of *one-way (attribute) synchronisation* based group communication infrastructure, the attribute can properly be distributed.

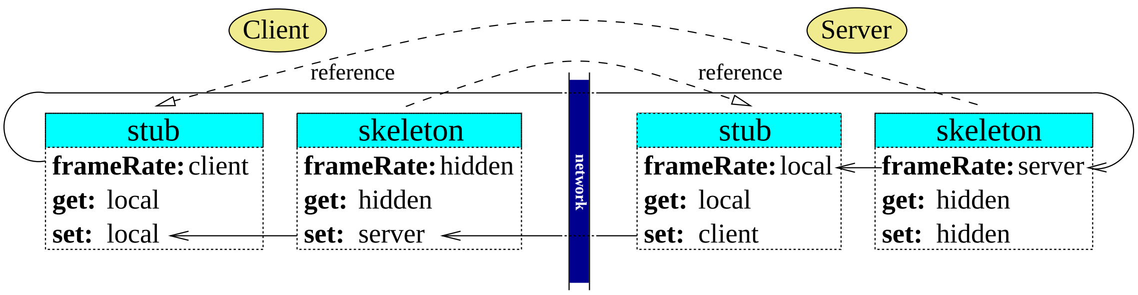


Figure 1: The Model of the Solution

The abstract solution can be implemented using the concepts of distributed object programming (see Figure 1). A skeleton and a stub have to be deployed on the client-side and on the server-side. On the

[1] Note that our approach can be used for general data multicasting; it is not limited to the context of multimedia.

client-side the remote interface of the skeleton consists of the *set* method. The remote interface of the stub consists of method `frameRate`, however it has to have a local interface consisting of the *get* and the *set* method. On the server-side the remote interface of the skeleton consists of method `frameRate`. The remote interface of the stub consist of the *set* method, the local interface the *get* method and `frameRate`. Note that the client-side stub needs the reference of the server-side skeleton, and the server-side skeleton needs the references of the client-side stubs.

This *ad hoc solution* may be implemented in many frameworks with more or less difficulties. The local interface can be created by extending the interface of the stub via inheritance, the hidden skeletons can be generated from interfaces containing the required methods. On the server-side we have to create dynamically the stubs for all clients, and the client-side stub can simply be created using the reference of the server-side skeleton.

However, the *ad hoc solution* results in almost certainly tangled design and code as most frameworks cannot be extended with reusable components implementing the generally applicable elements of the solution. In the next two sections we present a general solution to our initial problem and introduce the concept of *corpus* and a reusable *group communication* infrastructure.

2 Encapsulation of stub and skeleton functionalities by corpuses

Considering *method roles* we can find a more general approach of the solution presented in the above section: in some cases applications must contain server functionalities on the client-side or client functionalities on the server-side. These functionalities are usually implemented by stubs and skeletons managed by the user. This can result in a higher complexity compared to applications which have pure client and server components.

Stub and skeleton functionalities bound to a single object, can be encapsulated and hidden from the user. Our *corpus* class template and its instances allows the user to encapsulate these functionalities. Each method of a *corpus* can be configured in run-time to implement stub, skeleton and local method functionality by *method roles*: *local* methods are not part of the remote interface of the object; *client* methods act like stub methods, i.e. they issue RMI; *server* methods act like skeleton methods, i.e. they serve remote stub methods.

From an implementation point of view, the *corpus* is a stub which has a skeleton, i.e. the *corpus* has the capability of serving clients. The interface of the *corpus* contains the methods of the *IDL interface* specified by the user. These methods are arranged into three classes depending on the role they are playing. Some methods will be *client* method implementing remote method invocation. Other methods will be *server* methods serving clients, or *local* methods calling on their equivalent on the server object[2] directly.

Each *corpus* has to have its inverse pair on the other side (i.e. a corpus having server methods where it has client methods) or a skeleton serving its client methods. As a remote object reference determines a remotely accessible object, the reference of a *corpus* is equal to the reference of the skeleton it contains. In order to determine the remote party (corpus or skeleton), the client methods have to communicate with, a reference is passed to the constructor of the corpus, or specified later, if the remote party is not created yet[3].

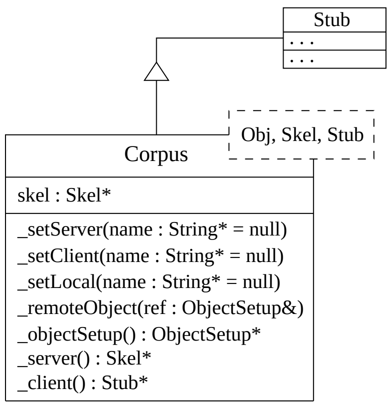


Figure 2: The Static Model of the Corpus Parametrised Class

[2]This is the object serving the client requests.
[3]Consider the situation, when for creating corpus A we need the reference of corpus B, and vice versa.

According to the above informal description the instances of the `Corpus` class template are derived from a given stub class and have a skeleton serving clients invoking methods contained by the corpus's remote interface (see Figure 2).

The Corpus is parametrised with a stub (`Stub`), a skeleton (`Skel`), and an abstract class "equivalent" of the interface the two other parameters are generated from (`Obj`). Besides the methods of the stub the corpus has its own methods as well. The constructor assigns all the stub methods the same role determined by the user, however the default role is *local*. The corpus contains three methods to set method roles: `_setServer`, `_setClient`, `_setLocal`. Using method `_remoteObject` we can set the remote object the corpus sends the invocation requests. The object reference of the corpus can be obtained by invoking method `_objectSetup`. Method `_server` returns attribute `skel`, method `_client` returns the corpus itself.[4]

In the solution for the initial problem of video multicasting, the stubs and skeletons on the client and server determine two interconnected corpuses. On the client-side, method `frameRate` plays a client role having its server party on the server-side. The `get` method is a local method both on the client- and the server-side. The server-side `set` method is a client method, served by the client-side `set` server method.

3 Semantic elements for data multicasting

The oneway synchronisation of attributes does not necessarily requires any special communication subsystem, actually many conventional infrastructures can prove to be sufficient. Considering either the TCP-based two-way or the UDP-based one-way communication, they offer the services required for an object-to-object attribute synchronisation. However, if we want to synchronise more than one client with the server, a group communication model has to be applied.

The infrastructure allowing object-to-objects oneway message passing can be built upon the multicast communication. Clients joining a multicast group receive the messages sent by the server to the group. However, IP multicasting is not supported by every Internet Protocol implementation, and the multicast messages are usually not routed. Therefore we implemented group communication using exclusively the UDP protocol, despite some of its drawbacks.

Without IP multicasting the server has to keep a record of the clients registered to receive the messages containing the new value of a given attribute. These messages (formal requests) will be sent to all of the registered clients when the server calls on the *set* method of the attribute. In order to receive such "server requests" the clients have to subscribe at the server; if a client wants to leave the group, it has to unsubscribe. Note that at this point we abandoned the high level transparency as the clients have to deal with the details of subscribing/unsubscribing for the server messages.

Figure 3 depicts the design of the simple group communication subsystem implemented for the "Station example" in ORB(M).

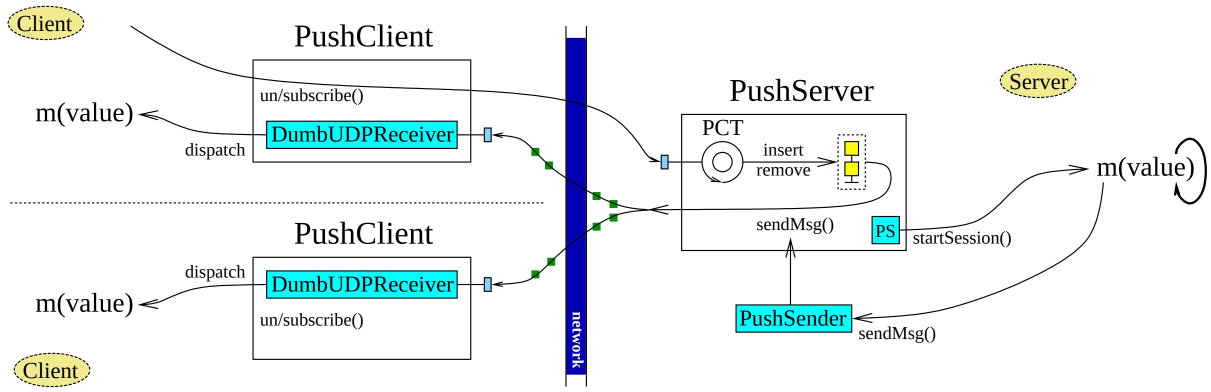


Figure 3: Group Communication Implemented by the Push Session Managers

The implementation of the new session managers and sessions is quite simple. The `PushServer` server-side session manager runs a thread (*PushControlThread - PCT*), which inserts the reference of remote methods subscribed for the push service into a store. When a remote method unsubscribes, the *PCT* removes its reference from the store. The `PushServer` creates only the `PushSender` one-way session which

[4]Note that these methods are minimal for the implementation of the basic features of the corpus determined above, and the interface can be extended with new features and services.

uses method `sendMsg()` to message sending. When method *m(value)*[5] is called, it gets the `PushSender` session and calls on `sendMsg()` in order to transmit the marshaled request message. This method forwards the message to the session manager, which sends it to all of the registered clients via UDP.

The client-side session manager called `PushClient` creates a session called `DumbUDPReceiver`, which receives the request messages sent by the server. The session dispatches the requests to the invocation semantics, which completes the remote invocation.

The developer using `PushClient` have to know the network address and the port number the `PushServer` handling the un/subscribe control messages; which can be obtained from `PushServer` itself. On the client the user can start or stop receiving request messages by calling on method `subscribe()` or `unsubscribe()`.

4 Implement the solution using custom invocation semantics

In this section we introduce how the corpus and the new session managers can be used in our prototype system to solve the problem stated in Section 1. First of all, the `Corpus` class template is instantiated passing it the abstract class (`Station`), the skeleton (`Station_skel`) and the stub class (`Station_stub`) generated from the IDL interface. This can be done like that:

```
typedef Corpus<Station, Station_skel, Station_stub> Station_corpus;
```

On the server-side the user has to instantiate the corpus class `Station_corpus`, set the roles of the methods defined in the IDL interface, create the new session manager and customise its invocation semantics. The below code snippet depicts these steps:

```
Station_impl obj;            // the server object
Station_corpus corpus(&obj); // create the corpus

corpus._setClient("frame.set"); // the set method will be a _client_ method
corpus._setServer("frameRate"); // method frameRate will be a _server_ method

... // save the object reference of the corpus

MethodSemantics ms = StubHelper::getMethodSemantics(*corpus._client(), "frame.set"); // get sem.
PushServer* smgr = new PushServer();            // create the session manager
ms.semanticElements()->setSessionManager(smgr); // replace the old session manager with smgr
StubHelper::setMethodSemantics(*corpus._client(), &ms, "frame.set"); // replace the old semantics

... // save the host:port where the session manager receives the control messages

while (true) {
  Frame f = nextFrame(); // somehow generate a new frame
  corpus.frame(f);       // assign f to attribute frame
  delete f;
}
```

As we have already mentioned the user needs an instance of the corpus class and the server object he wants to make remotely accessible. This object will serve the local invocation requests. As the corpus sets all of its methods to local, the user has to determine the client and server methods using its control methods.

The next major step is the customisation of invocation semantics used by the *set* method of attribute *frame* enabling it to use the `PushServer` session manager. First the invocation semantics of the *set* method is obtained using a helper class (`StubHelper`). Then a push server is created and the old session manager is replaced with the new one. Finally, the modified invocation semantics are assigned to the *set* method.

At this point the user can start creating the frames he wants to assign to attribute frame on the subscribed clients. Whenever `corpus.frame(f)` is called, the subscribed clients receive the UDP request containing the marshaled frame. On the client-side the user also have to create an instance the corpus class `Station_corpus`, set the roles, create the new session manager and customise the invocation semantics used by the *set* method.

[5]This is the *set* method of attribute *m*. In our example we will call the *set* method of attribute *frame*, i.e. `frame(value)`.

```
... // obtain the object reference "ref"

Station_impl obj;                 // the server object
Station_corpus corpus(&obj, ref); // create the corpus

corpus._setServer("frame.set"); // the set method will be a _server_ method
corpus._setClient("frameRate"); // method frameRate will be a _client_ method

... // obtain the control "host" : "port" of the PushServer

// get the callee invocation semantics used by the set method
MethodSemantics ms = SkeletonHelper::getMethodSemantics(*corpus._server(), "frame.set");
PushClient* smgr = new PushClient(host, port, skel_id, set_method_id); // the new PushClient
ms.semanticElements()->setSessionManager(smgr); // replace the old session manager with smgr
// replace the old invocation semantics
SkeletonHelper::setMethodSemantics(*corpus._server(), &ms, "frame.set");

smgr->subscribe();   // start receiving messages
sleep(some_time);    // wait some_time during the messages are received
smgr->unsubscribe(); // stop receiving messages

cout << "frame rate: " << corpus.frameRate() << " fps" << endl;           // _remote_ invocation
cout << "first byte of the frame: " << (int) (corpus.frame())[0] << endl; // _local_ invocation
```

The client-side operations are similar to the server-side. First object `obj` is created containig attribute `frame`. Then the corpus is created which will receive the synchronisation requests from the server and execute remote method invocation whenever method `frameRate` is called. After the corpus is created, the roles of the *set* method and `frameRate` are set.

For the customisation, the invocation semantics are obtained by using the skeleton helper. Then the `PushClient` is created, which will receive the server messages. For this the user needs the network address (`host`) and the control port (`port`) of the push server he wants to connect, and he needs the identifier of the skeleton (`skel_id`) and the *set* method (`set_method_id`) enabling the push client to dispatch the server messages.[6] Finally the old session manager is replaced and the customised semantics are assigned to the *set* method.

In order to start the synchronisation, the client subscribes to the server, then unsunscribes to stop it. Moreover, the user can get the frame rate issuing a *remote* invocation or the frame issuing a *local* invocation.

As we have seen in this section, our ORB(M) framework allows the user to implement new semantic elements and create customised invocation semantics to solve a lower level problem; in our case, multicast communication. Using customised semantics gives the freedom of the application developer to concentrate on the problem of the high level design. The effect of the customisation on the application's performance is twofold: the user can exploit the specific features of the problem and thus improve the performance, however making customisation possible obviously results some additional overhead.

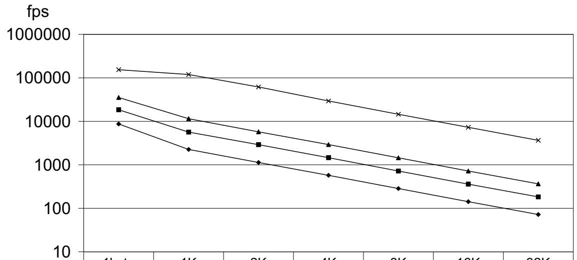
5 Performance measurements

In our tests we used the `Station` interface (see Section 1) with properly customised invocation semantics (see Section 4). In both tests we measured the execution time of 10000 invocations 10 times, left the maximum and minimum value and calculated the average execution time. The client and server computers were running Debian GNU/Linux 2.2 Release 2 (kernel version 2.2.19).

The *first* test measured the maximal framerate with different client numbers (1, 10, 20, 50) and frame sizes (1 byte, 1k, ..., 32k) by invoking on the set method of the attribute on the server side. In this test the server contained two Pentium II 350 MHz CPUs and 192M RAM, and the client was a Pentium II 266MHz computer; they were linked with a 10Mbit network.

As we have implemented a linear algorithm instead of using the IP multicasting infrastructure, the results show that the maximal frame rate (see fps) *decreases linearly* with the growth of client number and frame size (see Figure 5). K relation between the maximal frame rate and the two parameters the

[6] The host:port parameters can be obtained via a remote invocation or a common file, and the identifiers are contained by the object reference of the corpus.



	1byte	1K	2K	4K	8K	16K	32K
50	8688	2262	1130	572	285	142	72
20	18484	5663	2909	1455	720	359	183
10	35335	11481	5731	2919	1450	721	365
1	153846	119048	61728	29412	14513	7283	3663

Figure 4: Maximal frame rate with different number of clients and frame sizes

user can easily decide whether the push session managers fit to a given problem or not. Naturally, the performance could be improved by implementing IP multicast based session managers, however the linear relation between the frame rate and the frame size would presumably be preserved.

The *second* test measures the performance of the framework rather than the new semantic elements. Making measurements independent of the installed network and the current load, the client and the server were running on the same machine and communicated via the local loop-back interface. In this test we used an AMD Duron 600 MHz computer with 256M RAM. The results were compared with those we obtained from omniORB303[OMNI] using the same testing environment and method.

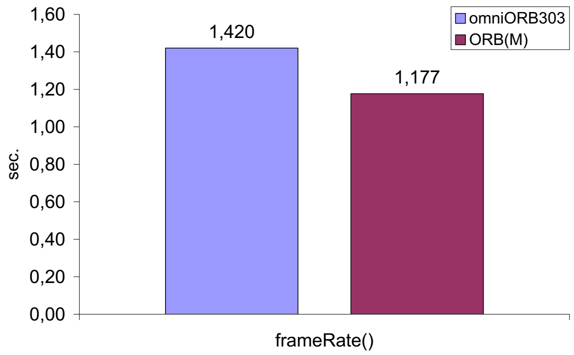


Figure 5: The delay in seconds of 10000 invocations on method `frameRate`

The results show (see Figure 5), that although the pluggable semantic elements result in additional overheads, the delays we measured in case of ORB(M) are about the *80%* of the delays of omniORB303. These reults are quite promising and imply that the overheads were successfully kept low. Although this single measurement is far from enough to draw an accurate image of the framework's performance. Moreover, the framewok is not a full-featured CORBA implementation, thus its first "complete" version will presumably result in slightly higher interaction delays.

6 Conclusions

We deem that offering the developer the freedom of "middleware programming" has significant benefits. One of the most important benefits is the *performance improvement* the users can achieve by exploiting the specialities of the problem they solve. A second benefit is the possibility of *better design and structuring*. Finally, these semantic elements are *reusable and composable*, i.e. we can create a number of invocation semantics by combining different elements.

However the freedom we offer is coupled with great responsibility. Although the default semantics generated by the *IDL* compiler are interoperable and highly transparent the developer have to pay greater attention to the customised invocation semantics. The customisation can *threaten the interoperability* by incompatible and non-standardised request and response messages or communication infrastructures. This *incompatibility* problem can be solved in the development state by interoperability testing. Changing the invocation semantics in run-time can also result in problems such as communication failures. We can avoid these problems by assigning the invocation semantics right after the object creation, or applying *static* customisation using invocation semantics definitions.

The ORB(M) framework has been built upon some of the concepts of the Composable Method System (CMS) [HOF99]. The CMS framework introduced a fully object-oriented approach where invocation semantics were themselves viewed as first class abstractions. This approach allowed the extension of JavaCMS (an implementation of CMS in Java environment) with Time-independent Invocation [HU01].

Using Meta Object Protocols (MOPs) [KRB91, FD99] the ORB could offer even more flexibility to the programmers; however their performance is considerably lower compared to the non-MOP based ORBs. The *OpenCorba* by Thomas Ledoux [LED99] is a reflexive object broker opening the invocation mechanism of the CORBA software bus. *Guaraná* by Alexandre Oliva is based on a modified version of Kaffe OpenVM^{TM} open-source Java Virtual Machine, which has been enabled for interception of operations [OB99]. The MOP of *Guaraná* introduced a *composer* meta-object remembering to our *semantic weavers* which controls the reconfigurations of its component meta-objects.

The main guiding principles behind the design of our ORB(M) framework are *flexibility* and the *adaptability*, *performance* and *simplicity*. The framework allows the programmer to design their applications using high level abstractions and enables them to exploit the specialities they have learned from the problem and the design. For exmaple, in some cases it is not necessary to use such complex protocols such as GIOP and an application-specific protocol and communication schema can serve the problem better. However handling specialised invocation semantics in run-time passes the complexity to the design. That's why we extended the interface definition language and offer a simplified, compile-time (static) customisation. The *Invocation Semantics Definitions* (ISDs) can be placed into a separated file determining the non-default semantics of the remote methods. This simplifies the application code and separates the semantic aspects [KIC96].

References

[FD99] Ira R. Forman and Scott H. Danforth, *Putting Metaclasses to Work*, Addison-Wesley, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[HOF99] Markus Hof, *Composable Message Semantics in Object-Oriented Programming Languages*, Trauner, ISBN 3 85487 118 X, 1999.

[HU01] Markus Hof, Attila Ulbert, *Time Independent Invocation in JavaCMS*, Periodica Polytechnica Ser. Electrical Engineering, Vol. 44, No. 1, pp. 1-19, 2001.

[IONA] The homepage of IONA Technologies Plc., http://www.iona.com

[KRB91] G. Kiczales, J. des Rivières and D. Bobrow, *The Art of the Metaobject Protocol*, The MIT Press, 1991.

[KIC96] G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C. Videria, C. Maeda Lopes, and A. Mendhekar, *Aspect-Oriented Programming ACM Computing Surveys*, 28(4es), 1996.

[LAZ97] A. A. Lazar, *Programming Telecommunication Networks*, IEEE Network, October 1997.

[LEA97] D. Lea, *Concurrent Programming in Java.* Addison-Wesley, 1997.

[LED99] Thomas Ledoux, *OpenCorba: a Reflective Object Broker*, 1999.

[MG99] Peter J. McCann, Gruia-Catalin Roman, *Modeling Mobile IP in Mobil UNITY*, ACM Transactions on Software Engineering and Methodology, Vol. 8, No. 2, April 1999.

[OB99] Alexandre Oliva and Luiz Eduardo Buzato, *The Design and Implementation of Guaraná*, 5^{th} USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), 1999.

[OMG] The homepage of Object Management Group, `http://www.omg.org`

[OMG98] Kimmo Raatikainen (editor), *White Paper on Wireless Access and Terminal Mobility in CORBA*, Version 2beta, OMG Telecom Domain Task Force, OMG Document: telecom/98-11-09, 1998.

[OMG00] Object Management Group, *Audio/Video Stream Specification*, version 1.0, OMG Document: orbos/00-01-03, January 3, 2000.

[OMG01a] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, version: 2.4.2, formal/01-02-27, February 27, 2001.

[OMG01b] Object Management Group, *CORBA 2.4.2 Minimum CORBA chapter*, Chapter 23, February 27, 2001.

[OMNI] The Hompage of omniORB: `http://www.uk.research.att.com/omniORB/`, AT&T Laboratories - Cambridge

[SCH98] Douglas C. Schmidt, *The design of the TAO real-time object request broker*, Computer Communications 21, pp. 294-324, 1998.

[SCH00] Douglas C. Schmidt, *Pattern-Oriented Software Architecture: Pattern for Cuncurrent and Networked Objects*, Wiley & Sons, ISBN 0-471-60695-2, 2000.

[ULB01] Attila Ulbert, *Pluggable semantic elements and invocation semantics*, ECOOP'2001, PhD Workshop, http://www.st.informatik.tu-darmstadt.de/phdws/a1.ecoop.ps.gz, Budapest, Hungary, June 2001.

[YT87] A. Yonezawa and M. Tokoro (editors). *Object-Oriented Concurrent Programming*, MIT Press, 1987.