Crossing Technological Domains Using the Inter-ORB Request Level Bridge – Preliminary Performance Study *

K. Zieliński, A. Uszok, M. Steinder

Institute of Computer Science, University of Mining & Metallurgy
Al. Mickiewicza 30, Cracow, Poland
tel: +48 (12) 17 39 82, fax: +48 (12) 33 89 07,
e-mail: {kz, uszok, gosia}@ics.agh.edu.pl

Abstract. This paper evaluates an overhead introduced by components that build a framework for inter-ORB request level bridge. It shows that TypeCode processing complexity is the most influential factor that determines efficiency of almost each generic bridge. This has been experimentally proved by evaluation of the components of the implemented half-bridge between Orbix 1.31 and IIOP domain.

1 Introduction

Existence of many different ORB domains is justified for performance, security and management reasons. Even in a context of CORBA 2.0 emerging standard this diversity of application requirements leads to many technological domains distinguished by various data representations, communication protocols, addressing and cooperation schemes.
Providing interoperability across technological domains need a bridging mechanism which resides at boundaries between domains and transforms requests expressed in terms of one domain's model into a model of a destination domain. This activity introduces an overhead which depends on many factors. Their identification and experimental evaluation is crucial for applicability of this techniques and efficient inter-ORB bridge design.
This paper concentrates on request-level generic transparent bridging defined by UNO specification as the most general approach. Implementation of this bridging technique is supported by components such as DSI, DII, and IR.
The aim of this paper is to evaluate an overhead introduced by Inter-ORB bridge components that build the framework for inter-ORB request level bridging proposed in [8]. The half-bridge components are analyzed separately to identify parts which have the greatest influence on an overall bridge performance. An advantage of this approach is that the obtained results can be also used for other applications that require DSI or DII usage.
A generic half-bridge requires dynamic interpretation of request arguments of various data types. This process is controlled by the TypeCode. Organization of the presented study is based on the thesis that TypeCode processing complexity is the most influential factor that determines efficiency of almost every generic bridge. This claim has been experimentally proved by evaluation of the components of the implemented half-bridge between Orbix 1.3.1 and IIOP domain.

* This work was sponsored by the European Commission under the COPERNICUS project TOCOOS no. CP940247. (http://galaxy.agh.edu.pl/research/cs/TOCOOS/COPERNICUS.html)

The rest of this paper is structured as follows: in the next section basic components of the framework for inter-ORB request level bridge have been described and their activity has been analyzed. Then, in Section 3 implementation of selected components, which are responsible for dynamic interpretation of arguments of a forwarded requests and, thus, introduce the greatest overhead, has been presented in more detail. This section focuses on TypeCode translation process, DSI and DII functionality, and organization of IR access. The implemented components of the half-bridge are very complex, thus, the extensive testing have had to be carried out. A methodology of this process is presented in the next section. In Section 5 results of preliminary performance studies of the investigated components have been presented and discussed. The paper is ended with conclusions.

2 Half-bridge functional model

Half-bridge functional model is determined by its location in a cooperation environment, on domain borders. Its task is to receive request addressed to a remote server from the local client, translate it into server's format and forward it to the server.
To perform this task a half-bridge must be able to: initialize itself, handle and understand client's request, translate objects defined in CORBA model, and construct a new request. The initialization process has been investigated in [8] and does not influence the overall half-bridge performance, so the last three activities will be elaborated here. The framework architecture of the half-bridge which carries out these activities is presented in Figure 1.

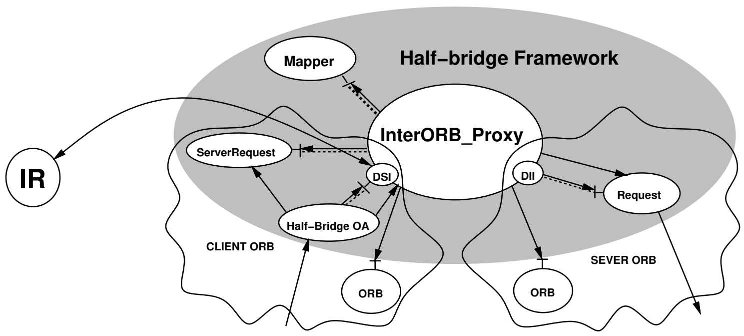

Fig. 1. Half-Bridge Framework Architecture

2.1 Incoming Request handling

A request to an object in CORBA compliant systems is taken over by an object adapter, the same which is used to activate the object. It uses Dynamic Skeleton Interface (DSI)

to interpret and serve a request.
Dynamic Skeleton Interface is a CORBA 2.0 mechanism, therefore most of currently available ORB systems do not possess its implementation. This necessitates extension of these systems with DSI.
DSI provides the same functionality as the type-specific IDL generated skeletons. However, it allows an object implementation to receive any operation request on any available CORBA object without requiring compile-time knowledge of the type of the object it is implementing. The main component of this information is a set of TypeCodes, which describe operation arguments. This information is stored in IR (Interface Repository) from which it could be retrieved in run-time [5].
DSI is a server-side analogue of Dynamic Invocation Interface (DII). All requests on a particular object are performed by having the ORB invoke the same up-call routine – *invoke* method of the *DynamicImplementation* object. This function need an access to operation data including its name, parameters' types and values as well as its result, which are offered through an abstract ServerRequest object, which possesses functionality for retrieving data from a request buffer and writing return values into the request buffer. These activities are controlled by TypeCodes.

2.2 Mapping objects defined in CORBA model

CORBA 1.2 standard left some parts of the system undefined because the then state of the art did not allow standardization or some of the elements were intentionally left opaque to allow their specialization for different uses. These deficiencies in the CORBA definition allow vendors of CORBA compliant systems to specify different extensions to the same interfaces to make them usable. To allow two different ORBs to cooperate a mapping from one ORB to another and vice versa must be defined for Objects, TypeCodes, Principals, Contexts and ServiceContext. TypeCode mapping together with Object References mapping is most common and introduces the greatest overhead. The need for this kind of mapping occurs when a TypeCode is an argument of operation invocation. However, it is more common that an operation definition has to be translated when crossing domain's border. It always involves TypeCodes mapping, which describes data types of operation arguments. In the case of a half-bridge built around ORB backbone only mapping from cooperating environments to this ORB backbone and vice versa is needed. Mechanisms responsible for performing this mapping may take necessary information from bootstrapping or from external protocols.

2.3 Forwarding Request to Server

A translated request is forwarded to the server using DII. Its activity is once again navigated by TypeCodes. The new NValue list has to be created which will contain arguments of the request in a server domain. Putting these arguments into the list is controlled by their TypeCodes. This operation is symetrical to the DSI activity.

3 Sources of overhead

When considering a half-bridge implementation (in this case all presented analyses are based on a half-bridge working between Orbix 1.3.1 [9] and IIOP domain) it is easy to recognize that the most complex computations are related to marshaling and unmarshaling messages, extracting and inserting parameters from/to ServerRequest and translating CORBA Objects. Processing all mentioned operations requires TypeCode accessibility and interpretation. This remark leads to the conclusion that a half-bridge overhead is very much depended on efficiency of the following functions:

- TypeCode format translation,
- Extraction and insertion of parameters from/to ServerRequest performed by DSI,
- Processing Interface Repository operations which are necessary to provide information for DSI and DII.

Implementation of software components that perform these processing will be described in more details. Another half-bridge activity which introduces significant overhead is Object References translation, however, it has been already evaluated in [8].

3.1 TypeCode translation

TypeCode is a value which represents type of data. The CORBA 1.2 standard defined only an abstract structure of the TypeCode, which consists of a "kind" field, and a set of parameters appropriate for that kind. It also specifies an interface enabling manipulating the TypeCode. However, it does not describe representation of this abstract structure, thus, existing CORBA 1.2 compliant environments possess their own proprietary internal representation of TypeCodes. The new CORBA 2.0 [4] standard defines TypeCode representation which should be used in IIOP messages. This led to the problem of translating TypeCode between proprietary formats when constructing the half-bridge as for instance in the case of Orbix 1.3.1 and IIOP.

The Orbix 1.3.1 TypeCode is represented as a string, containing a description of the given type. In this string the first character describes the TypeCode kind, e.g. type **long** is represented as simple "l". For complex types apart from the kind field the string stores their names, member fields and member types (following these rules if members are also complex types), etc.

On the other hand the UNO (CORBA 2) encoding of a TypeCode [4] is the `TCKind` enum value, followed by zero or more parameter values. For simple data types only the `kind` value is filled and the parameter list remains empty. It is significant only for complex types such as struct, union etc. Their encoding is done according to the CDR specification. Any new component of this buffer has to start from the position whose index is a multiple of four.

The two TypeCode formats between which the translation has to be carried out have completely different structures. Mapping could be done directly only for simple data types. Mapping for a TypeCode representing complex data types will involve complicated analysis of a recursive structure.

Here the description of two translation functions `Orbix2UNO` and `UNO2Orbix` with a stress on an overhead sources in their implementations is presented. They both act in a similar way: they take a TypeCode value in one encoding and construct its counterpart in the second encoding.

Translation from Orbix to UNO TypeCode encoding. The main problem encountered was evaluating the size of the created UNO TypeCode. The size of a memory area necessary to hold TypeCode is not known before the end of the whole analysis, thus it is impossible to allocate memory for the whole UNO TypeCode at the beginning of translation. Therefore it should be allocated piece by piece when required in time

of translation. Memory for incoming octets is allocated by an object of a special class `Buffer` dynamically in small arrays stored in a list. This solution requires transformation of information stored in the object of the class `Buffer` into one array of octets, which is a part of the UNO TypeCode encoding, when the translation is finished.
Another problem was mapping TypeCodes being parts of another TypeCode. In case of the TypeCodes with a complex parameter list, encoded as a sequence of octets, it is important to know the length of the inner TypeCode, because this value should precede elements of the sequence. The problem was solved by recursive calls of the internal function `Inner_Orbix2UNO()` which returns length of the TypeCode being mapped. Each of those recursive calls works on the same `_buffer` defined in function `Orbix2UNO()`. This function interprets a contents of the Orbix TypeCode and fills `_buffer` with information about structure of the corresponding UNO TypeCode. The interpretation is based on the system of cases, each corresponding to one of simple or complex types. In case of complex types function `Inner_Orbix2UNO()` calls itself for interpretation of subtypes (for instance types of elements of array, or types of fields in structure and so on).

Translation from UNO to Orbix TypeCode encoding. Inside the function `UNO2Orbix` – first the UNO TypeCode is changed to the tree form, where every node contains attributes of an element of a structure (type, name, pointers to possible members when it is a complex structure). Many times jumps back have to be done in tree's structure (i.e. when "qualified" structure name has to be created – **struct1::struct2**, or structure declaration has to be repeated in the Orbix TypeCode twice).
The length of the Orbix TypeCode is calculated using the created `tree`. Then, such a number of bytes is allocated in memory. After that the tree is translated to the Orbix TypeCode encoding by recursively analyzing its structure. If a currently analyzed node represents a simple data type, it is translated directly to a corresponding Orbix value for this TypeCode. If, however, the TypeCode for a complex data types is encountered in the node, first, the beginning of this structure is produced, next, all its members (or fields i.e. for an enumerated type) are processed recursively, and finally the tail of the structure (as necessary in Orbix format) is created.

3.2 Dynamic Invocation Skeleton for Orbix

DynamicImplementation interface, whose implementation constitutes DSI, is defined in IDL as follows:

```
interface DynamicImplementation {
       void invoke(inout CORBA2_IONA::ServerRequest sreq);
};
```

The Orbix 1.3 IDL compiler constructs three C++ classes based on this interface, which were modified as follows in order to implement DSI for Orbix:

- `DynamicImplementation` – All methods of this class were removed.
- `DynamicImplementationBOAImpl` – It is a base class for all objects that wish to implement DIR. Its second constructor has been added to allow registration of the class with an arbitrary interface name. The first parameter of this constructor denotes an interface identifier of the implemented object.

– `DynamicImplementation_dispatch` – It implements dispatch method of the `DynamicImplementationBOAImpl` derived classes. When a request directed to one of the objects of a class that inherits from `DynamicImplementationBOAImpl` arrives the `dispatch` method of `DynamicImplementation_dispatch` class is called. The body of this method has been modified in such a way that it accepts all operation names, creates a `ServerRequest` object, invokes `invoke` method of the specified object and finally writes all inout, out and return values into the received from the client Request object. This method does not require any changes when being used in users programs.

A programmer who uses DSI has to implement the `invoke` method on his own. This method gets as a parameter a ServerRequest object. The object instance on which a request is made, can retrieve all information necessary for handling request from this object and modify the inout parameters, out parameters, and result. The object implementation also reports exceptions through this object.

ServerRequest implementation. ServerRequest object implementation in Orbix wraps up the Orbix `Request` object. Its main function – `params()` is used to unmarshal/marshal income parameters and returned values using internal functions `extract`, and `insert` respectively. Function `extract` reads parameters placed in an Orbix Request and places them in the NVList structure. On the other hand the function `insert` reads values from NVList and puts them into the Request. For thier activity it has been necessary to implement functions which determine size of data represented by a given TypeCode and read data of an arbitrary complexity from the Request. Both functions are based on TypeCode interpretation. To analyse TypeCodes function `tc_traverse()` has been implemented. Analysis of TypeCode is not complicated when simple data types are in use. In that case we have only to determine which of these types is processed. Then the proper code is executed, which deals with it. Problem gets a bit more intricate when complex data types are used (for example structures, unions, structures of unions and so on). In that case the function must be called recursively for each subtype of the examined data structure. The interior of function `tc_traverse()` consists of one multi-conditional switch, which checks the kind of type represented by TypeCode and, next, executes required sequence of operations for this type.

Function `tc_traverse()` does not operate on Orbix Request data – these actions such as writing and reading Orbix Request are all performed by functions, pointers to which are passed to the function `tc_traverse()`. On the other hand these functions do not operate on TypeCode – its analysis is made by `tc_traverse()`. For reading a value from Request function `fill_mem` was defined Operations performed by this function are very easy for simple data types – the function just extracts data from Request and writes it into the area pointed by its argument. The problem gets more complicated when operating on complex data, then it is necessary to call function `tc_traverse()` which has `fill_mem()` as its argument. In this case `tc_traverse()` is called in order to modify parameter containing TypeCode of the retrieved from the Request parameter, so that it points to a proper part of Typecode and to call function `fill_mem()` when necessary. Reading data from Request was simplified by stream operator >> , which was defined for all simple types of data.

For writing a value into Request function `fill_req` was defined. Writing data to Request is made with the use of stream operator >> defined for all simple data types. Problem of writing complex data into Request was solved in the same way as it was shown for function `fill_mem()`. In case of a complex TypeCode function `fill_req()`

calls `tc_traverse()` with itself as an argument of this function.

3.3 Interface Repository access

DSI and DII require an access to the Interface Repository (IR). Orbix environment provides IR implementation that makes it possible to build client and server parts of the half-bridge in this environment. Unfortunately, when the reported research started there was no IR for the IIOP domain available. To solve this problem a service specific halfbridge called IR Access Server (IRAS) has been constructed. It is a service specific bridge because its functionality is limited to translating requests addressed to the IR objects only.
The IIOP client generates requests to IR according to its interface specification. These requests are captured by IRAS that represents IR in IIOP-domain, transformed into Orbix format, and sent by the Orbix IR client to Orbix IR server. Results of these calls are sent back following the same route.

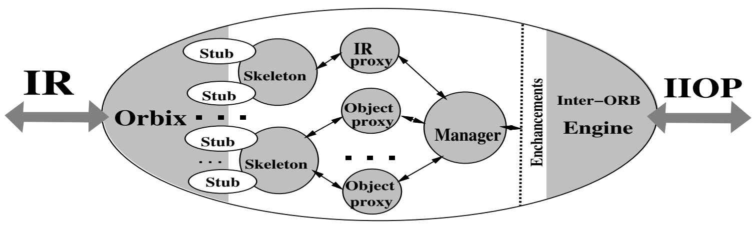

Fig. 2. Internal structure of IRAS

The IRAS internal structure is shown in Figure 2. It consists of:

1. Interface Repository Proxy Object in IIOP domain, which represents an access to Orbix Interface Repository from IIOP part of IRAS.
2. IR Objects Proxies for IIOP domain (below called OrbixEncapsulators) which encapsulate Orbix objects entries returned from Orbix Interface Repository to use them in IIOP part of IRAS.
3. IIOP Skeletons, which handle incoming requests. They are specific for individual object types (interfaces) and are pointed by appropriate Proxy Object.
4. Main Manager which manages all above components, translates requests from IIOP into Orbix format and vice versa and communicates with all requesting objects.
5. Enhancements of Inter-ORB Engine to provide management (creation and destruction) of more than one object, such as creation and destruction of objects and two level dispatching of requests — the first, choosing a target object, the second, executing appropriate skeleton function.
6. Inter-ORB Engine part responsible for utilization of IIOP standard libraries.

The IRAS activity may be described as follows. First, request generated by IIOP client is received by Inter-ORB Engine Object Adaptor (OA) and dispatched by Main Manager

to the interface skeleton function pointed by the Proxy Object the request is referred to. The suitable Object is located according to Object Key in `Objects Table`. If this Object does not exist it is created by the OA and put into the `Objects Table`. The interface skeleton function retrieves a name of the requested operation from the IIOP request and on this basis passes the request to a proper implementation skeleton function. This function recovers operation parameters and performs a standard Orbix operation call on a suitable IR Object. The last step activates suitable Orbix Stub function. On return, if the objects references are passed back, the object proxies are created and registered in the object table. Next, the result is put into the IIOP request and send back to the IIOP Client.
IIOP Client Stubs for every operation of interface supported by objects in Orbix Interface Repository are implemented to provide access to IR for any process in IIOP domain. All operations provided by Orbix IR interfaces have been made accessible for IIOP clients. IRAS activity does not involve the DSI and DII mechanisms and is based on statically linked stubs. The most complex operation performed by IRAS is a TypeCode translation process whose implementation is based on the `UNO2Orbix` and `Orbix2UNO` functions described in the previous section.

4 Components testing

When the components were implemented, there was a need to check if they worked correctly as well as to evaluate their performance. Though, they were tested on some simple interfaces, which implied simple TypeCodes, it did not guarantee that they would work correctly in other more complex cases. A solution was to construct a tool that generates random IDL structures according to the IDL syntax and semantics, and uses its output as a source of a random operation definition and TypeCodes.

4.1 TypeCode mapping functions testing

The whole testing scenario is presented in Figure 3 and may be described as follows:

1. Generate a module using Random Interface Generator.
2. Compile received code with the Orbix IDL compiler.
3. Extract TypeCodes from files generated by the IDL compiler.
4. Translate Orbix TypeCode to IIOP format using `Orbix2UNO` function.
5. Translate the obtained UNO TypeCode back to Orbix format using the `UNO2Orbix` function.
6. Check if the Orbix TypeCode produced by the `UNO2Orbix` function is identical to this generated by the Orbix 1.3.1 IDL compiler.

4.2 DSI testing

DSI tester is an environment that enables checking on correctness and efficiency of DSI mechanisms used by server encoding operations sent by client. It consists of programs *client*, *server* and random-interface generator. Client calls chosen operations with data as arguments and server "echoes" this data back to client. In order to find out if DSI works properly client has to check if data sent by him does not differ from data echoed by server. DSI performance is estimated on both sides: in client program – by measuring lapse of time during invocation of operation, in server program – by measuring how much time server takes to extracting data from request) and inserting data to request.

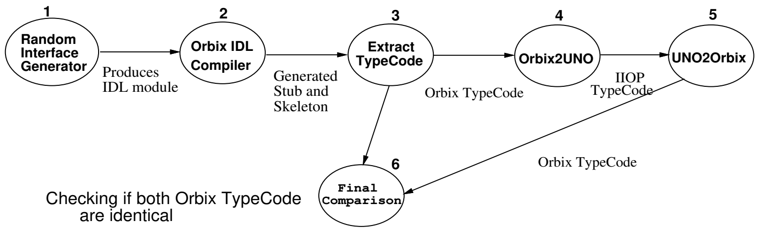

Fig. 3. Testing scenario of the TypeCode mapping functions

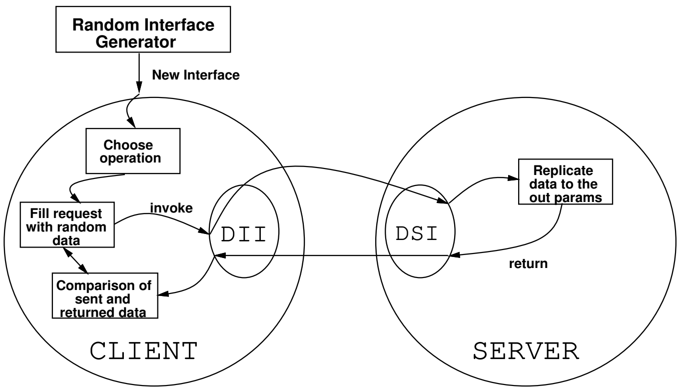

Fig. 4. Concept of DSI tester

5 Performance study

All performance results presented in this section were obtained on Sun SparcStation 10/514 and Sun SparcStation 5 machines with **SunOS 5.4** operating system. During the evaluation process, every test was repeated 5000 times in a loop for the given operation or TypeCode and the time was measured with the system `time` tool. The presented results are the average values calculated using the collected data.

5.1 TypeCode mapping functions performance

Some performance evaluation tests have been carried out to estimate time consumed by translation of TypeCode of a given complexity.
First, functions with simple "linear" structured data types (with growing number of "long" fields in a structure) were measured. In this case the time consumed also grows

linearly, although the `Orbix2UNO` function is considerably faster than `UNO2Orbix`. The reason is probably that `UNO2Orbix` uses more complicated tree-shaped internal structure for translating TypeCode. Next, the evaluation was done for nested structures with growing number of nests in structure, as follows:

```
struct st1 { ....
                  struct stn {
                               long l;
             ....
                  }field;
};
```

The obtained times of translation are shown in Figure 5 and 6.

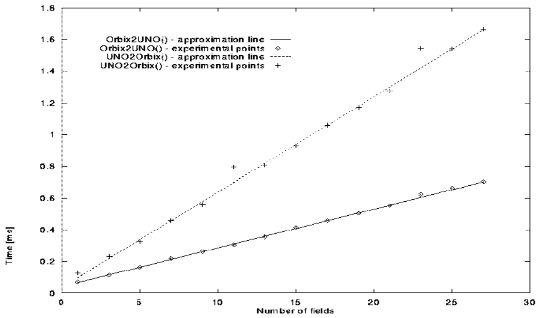

Fig. 5. Time consumed by TypeCode translation function for linear structures

Consumed time grows in the second case exponentially! This could be caused by an enormous increase of length of the Orbix TypeCode with the grow of data type complexity shown in Figure 7. It could be seen that for 200 bytes long IDL specification (what relates to 7th level of nesting) the Orbix TypeCode is nearly 7000 bytes. This forces both functions to make much work with simple reading and writing bytes and in consequence the consumed time grows faster than linearly.

5.2 DSI performance

Some performance evaluation tests have also been carried out to estimate time consumed by DSI operation versus complexity of their arguments. The tests have been performed for the invocation of operations – `extract` and `insert`, which respectively retrieve and

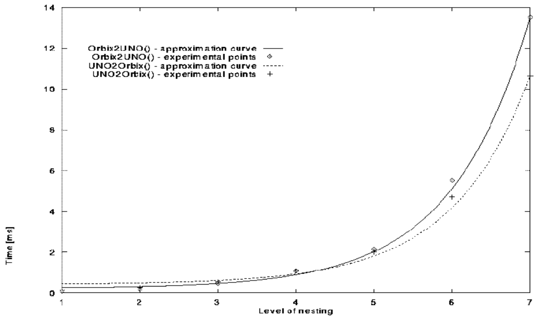

Fig. 6. Time consumed by TypeCode translation function for nested structures

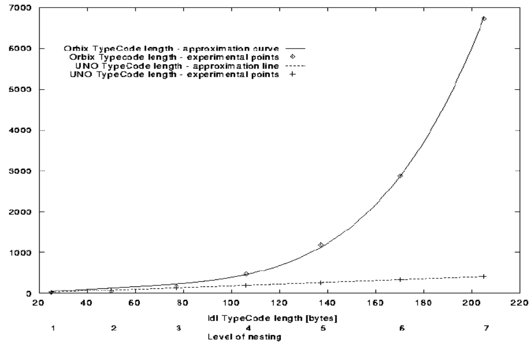

Fig. 7. Increase in TypeCode length vs. data type complexit

put data in the request. The obtained times are shown in Figure 8 and 9. In the first example only linear structure were subject to `params()` and `insert()` methods. In the second case the structure was nested as presented in the previous point.

5.3 IRAS performance

To evaluate IRAS overhead the difference in access time to IR in two situations has been measured: the first is, when the Orbix client performs the operation and the second then the same operation is called by the IIOP client via IRAS. This experiment has been

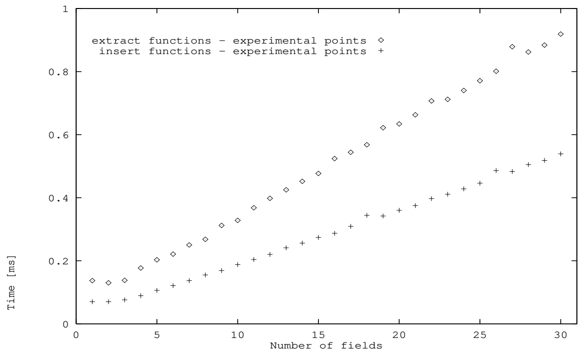

Fig. 8. Time consumed by DSI functions `extract` and `insert` for linear structures

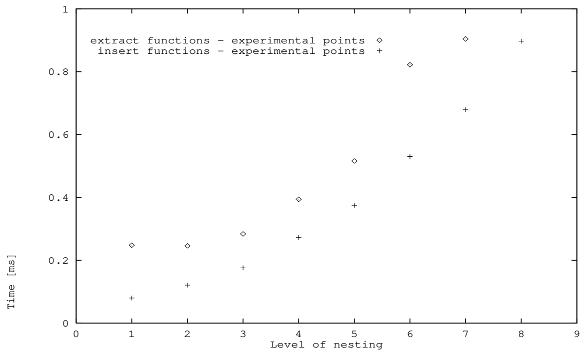

Fig. 9. Time consumed by DSI functions `extract` and `insert` for nested structures

repeated for every IR interface operation. The obtained results for `describe_operation` and `lookup_name` method are shown in Figure 10 and 11.

The presented results illustrate a general phenomenon that has been observed: IRAS introduces substantial overhead only for operations that need TypeCode translation. The process of creating proxies for Orbix IR objects in the IRAS introduces approximately only 4 [ms] delay.

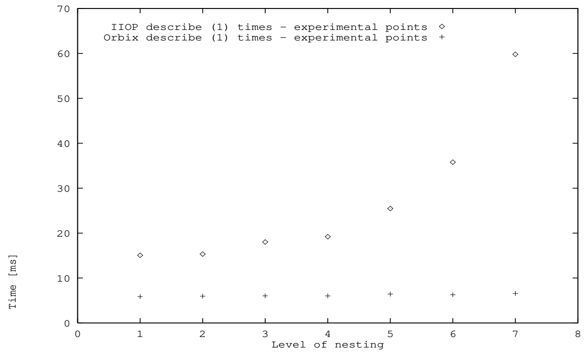

Fig. 10. Average access time to IR – `describe_operation` method

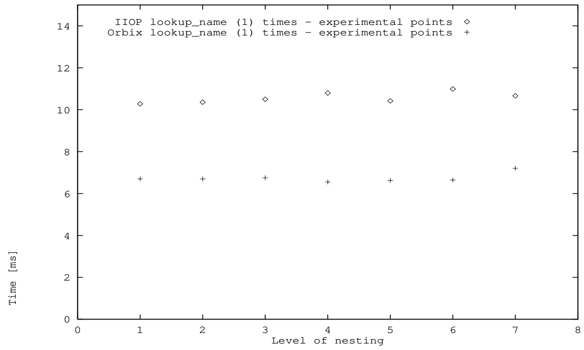

Fig. 11. Average access time to IR – `lookup_name` method

6 Conclusions

In the paper sources of overhead, when technological domains between CORBA-compilant systems are crossed, using the bridge, have been identified and evaluated. Despite of the fact that the presented results concern Orbix 1.3.1 and IIOP domain interoperability the conclusion presented below has a general meaning, as considered environments are representative for CORBA compliant systems.

The complexity of TypeCode processing has a substantial influence on DSI activity, CORBA Objects translation process and IR access time. The TypeCode encoding should

not lead to substantial growth of occupied memory with comparison to a number of bytes used for textual representation of data structures in IDL. It is evident from the obtained results that growth of the TypeCode representation length results in increase of the processing time.
The TypeCode translation process is more time consuming than extracting and inserting data from `ServerRequest`. The DSI functions `extract` and `insert` activities are in fact controlled by the TypeCode so its length influences the processing time.
The results obtained from the study of the service specific bridge such as IRAS also prove that the TypeCode translation process introduces the most substantial overhead. The IR interface operations that do not require TypeCode translation are performed very efficiently. The process of proxy objects creation does not introduce a substantial delay.
The concept of bridge presented in this article applies not only to the interoperability between CORBA 1.2 and CORBA 2 systems. It can be very valuable as the mechanism enabling division of a CORBA-compilant system into sub-domains and serving as an administrative, security and firewall tool. It also can be used for interworking between non-CORBA compiliant systems like DCE or ANSAware.

7 REFERENCES

1. *CORBA 1.2 Revision Draft* (1993) OMG Report 93-12-43, Object Management Group (OMG) Inc.
2. W. Harrison, *The Importance of Using Object References as Identifiers of Objects – Comparison of CORBA Object References* (1994) IBM Watson, TR
3. *ORB Interoperability. Joint SunSoft / Iona Submission to the ORB 2.0 Task Force Initialization & Interoperability Request for Proposals* (1994) OMG Inc., TC Document 94-3-1
4. *CORBA V2.0* (1995) OMG Inc.
5. *Interface Repository* (1994) OMG Inc., TC Document 94-11-7
6. *ORB Initialization Specification* (1995) OMG Inc., TC Document 94-9-46
7. A. Uszok, G. Czajkowski, K. Zieliński, *Interoperability Gateway Construction for Object Oriented Distributed Systems*, (1994) Proceedings of 6th Nordic Workshop on Programming Environment Research,
8. M. Steinder, A. Uszok, K. Zieliński, *A Framework for Inter-ORB Request Level Bridge Construction*, (1996) Proceedings of IFIP/IEEE International Conference on Distributed Platforms, Chapman & Hall,
9. *Orbix Programmers Guide* (1995) IONA Technologies Ltd.,

This article was processed using the LaTeX macro package with LLNCS style