Construction of a Generic Inter-ORB Bridge *

M. Steinder, A. Uszok, K. Zieliński
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: {gosia, uszok kz}@ics.agh.edu.pl

Abstract

This article summarizes experience in the design, implementation and testing of an inter-ORB bridge constructed according to the CORBA 2.0 standard. It proposes the framework of a request-level generic bridge enabling the implementation of these cooperation modules. A third-party CORBA 2.0 compliant ORB is used as a backbone. The article deals also with the problem of binding in a multi-ORB environment, proposing an architecture, consisting of some Object Services, which enable smooth and transparent interaction of bridges. As a case study, an implementation of the bridge for Orbix 1.3 is described. The article also discusses performance aspects of inter-ORB bridges.

Keywords

CORBA 2.0, CORBA interoperability, bridge, framework architecture

1 INTRODUCTION

With the proliferation of distributed object applications there is a growing requirement for a common platform base. This tendency should only strengthen in the near future. CORBA (OMG 93-12-43, 1993) promoted by OMG seems to gain the greatest interest in the software industry.

The CORBA Object Model (OMG 93-12-43, 1993) identifies various distribution transparencies which must be supported within each ORB environment. As it has been anticipated, none of the ORB implementations are able to address the large variety of user needs. The existence of many different ORB domains is justified also for performance, security and management reasons.

The diversity of ORBs – now evident – necessitates the introduction of means they could cooperate through; see in (OMG 94-3-1, 1994), (OMG 94-9-32, 1994) and (Uszok, 1994). Interoperability of different ORBs can be viewed as extending standard transparencies to span them. This problem was first addressed in CORBA 1.2 specification (OMG 93-12-43, 1993). Later, a general architecture for inter-ORB cooperation was put forward in the new CORBA 2.0 standard (OMG CORBA Revision 2.0, 1995) where, as one of the solutions to the interoperability problem, the notion of a bridge was introduced. The aforementioned

*This work was first sponsored by the European Union Copernicus Project no. CP940247 then by the Polish State Committee of Scientific Research (KBN) under grant no.: 8T11C01210

unit is supposed to reside at a boundary between ORBs and transparently transform requests from a source ORB to a destination one. According to the OMG strategic plans these standards should result in direct interoperability of applications built using ORBs implemented by different vendors.

The current status of CORBA 2.0 based technology may be characterized shortly as follows (OMG, CORBAnet 1996):

- There are some implementations of CORBA 2.0 standard today. However, vendors understand CORBA 2.0 compliance as the support of the IIOP protocol and the IOR reference format only.
- For example many ORBs proclaimed to be CORBA 2.0 compliant do not have TypeCode encoding as well as other complex data types implemented as stipulated in the standard (e.g. Orbix 2.1 (Orbix 2.1, 1996) or VisiBroker 2.0 (VisiBroker, 1996)).
- Due to large discrepancies between IDL compilers and the code they generate, porting an application from one ORB to another is not usually easy.

In spite of deficiences in the technology many applications have been developed using it. At the same time systems being introduced to the market are more and more CORBA 2.0 compliant. Under this circumstances further development of the existing applications could necessitate reimplementing existing components in the new technology, which, as it has been already stated might pose considerable problems. This problem may, however, be avoided if interoperability of old technology with the new one is possible. In CORBA 2.0 standard such interoperability may be achieved using a bridge. This facility also solves the problem of cooperation of CORBA 2.0 systems using different communication protocols. This article aims at presenting a framework for implementing an inter-ORB bridge and specifies services required to support its activity. By the framework a clearly defined structure of the bridge is meant enabling highly ORB-independent implementation. A third-party CORBA 2.0 compliant ORB is used as a backbone connecting bridgies. Inside the ORB a mechanism of dynamic binding across a multi-ORB environment is required for inter-ORB bridge activity. It will be shown that the mechanism may be successfully built using Trading and Lifecycle Common Object Services (CORBAservices, 1996).

The rest of this paper is structured as follows: in the next section basic assumptions in the design of the generic inter-ORB bridge are presented. In Section 3 a framework for implementing bridges is depicted. The next section describes the binding mechanism in a multi-ORB environment. In Section 5 an implementation of the bridge based on the proposed framework for Orbix 1.3 CORBA-compliant system is presented as a case study. Finally, the results of bridge performance evaluation are given with an analysis of main overheads. The article ends with conclusions.

2 BASIC ASSUMPTIONS

The presented way of addressing the problem utilizes **mediated bridging** technique. In this approach elements of an interaction are transformed at the boundary of each domain between the internal form of this domain and an agreed common form. Adapting a backbone style architecture is a standard administrative technique for managing networks. It brings about minimizing the number of bridges needed, while making the inter-ORB

cooperation match typical network organization. As such common form, according to the CORBA 2.0 standard, the IIOP protocol has been chosen and used by the **backbone ORB**.

Construction of a bridge depends on where bridge components are located: inside or outside ORB. The second option is called **request level** bridging. Request level bridges which mediate between distinct execution environments via a common protocol involve components, one in each ORB, known as *half-bridges*. Mediated request-level bridges between any two ORBs can be built without the need for information about the internal construction of these ORBs. It is the main reason why the approach has been adopted in the presented project.

A general principle of request-level bridging consists in the original request being passed by a client ORB to a special object, translating the request contents to a form understandable by the server ORB, invoking the required operation on an apparent server object and passing operation results back to the client. This object is called *InterORB_Proxy*.

Request-level bridges may be: interface-specific or generic. Interface-specific bridges support predetermined IDL interfaces only, and are built using IDL-compiler generated stubs and skeletons. Generic bridges are capable of bridging requests to all server objects regardless of IDL definitions of their interfaces using the Interface Repository, Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI). In this project a **generic** request-level bridge has been constructed.

Bridges should support an arbitrary number of *InterORB_Proxies*, which may be created as ordinary objects using the Basic Object Adapter (BOA) and the DSI. The multiproxy bridge requires internal concurrency of the server process provided by the multithreaded environment. The construction of the multiproxy bridge is also the fundamental subject of the paper.

3 BRIDGE FRAMEWORK ARCHITECTURE

The aim of this section is to present a general approach to constructing bridges. It was recognized that due to large discrepancies between ORB systems it is impossible to implement a bridge capable of cooperating with all of them. Instead, the framework has been created facilitating the implementation of bridges for all CORBA 2.0 compliant systems. Inside the framework, ORB dependent and ORB independent parts have been distinguished. The former must be implemented for each system separately because they rely on intra-ORB functionalities. The latter only use standard CORBA interfaces therefore and thus may be implemented once for all of the systems.

The concept of the bridge framework is presented in Figure 1. It shows the basic components of the bridge, necessary to relay a request from one ORB (*CLIENT ENVIRONMENT*) to another (*SERVER ENVIRONMENT*). In the real bridge, simultaneously translating requests in both directions, a number of *InterORB_Proxies* is multiplied.

3.1 InterORB_Proxy

The *InterORB_Proxy* object constitutes core of the bridge framework. It uses standard CORBA interfaces to translate request from the client's ORB to the server's ORB. It utilizes *dynamic implementation* which is a part of client's Dynamic Skeleton Interface.

The main *InterORB_Proxy*'s functionality is performed by the *invoke* method of the *DynamicImplementation* class. It creates a new request performing all necessary mappings and uses server's Dynamic Invocation Interface to forward it.

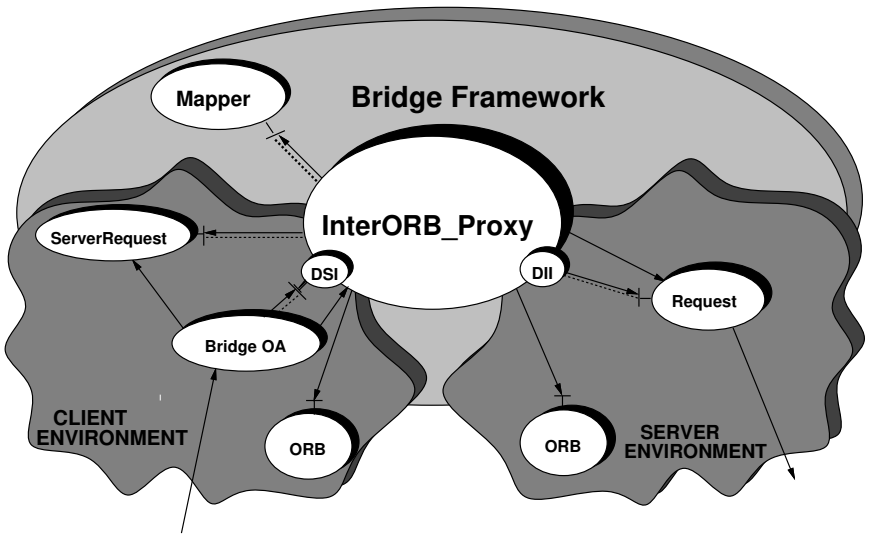

Figure 1 A Concept of the Bridge Framework Architecture

The *InterORB_Proxy* is implemented as a template parameterized by the names of CORBA modules belonging to adjacent ORBs (Figure 1). These names are in fact bridge constants whose values are determined at compilation time.

```
template <class CORBA_client, class CORBA_server>
class InterORB_Proxy : public virtual CORBA_client::DynamicImplementation {
  // It is a stringified object reference of a partner representing
  // a remote server
   CORBA_server::ObjectRef Peer;

  // Reference of the Mapper responsible for translating Objects, Typecodes,
  // Principals, Contexts and ServiceContexts from a CORBA_client to
  // CORBA_server representations
   Mapper * my_Mapper;

   public:

   InterORB_Proxy (CORBA_client::ORB *, CORBA_server::ORB *,
                   CORBA_server::ObjectRef target,
                   CORBA_client::ReferenceData & ref,
                   Mapper*) : CORBA_client::DynamicImplementation(ref);

   ~InterORB_Proxy ();
   void invoke (CORBA_client::ServerRequest *&, CORBA_client::Environment &);
};
```

It is worth noting that although the *InterORB_Proxy* possesses some attributes which constitute its state (`PeerRef`, `my_Mapper`) they are never modified while *invoke()* is being performed. The *InterORB_Proxy* does not remember requests being passed through it. Thus, it may be considered stateless. Therefore, in the multi-threaded version of the bridge many threads are allowed to run within the same *InterORB_Proxy* simultaneously.

The proposed approach based on *InterORB_Proxy* implementation as a C++ template is elegant from software engineering point of view but introduces a very serious practical problem. The two libraries that correspond to the client and server ORBs respectively have to be linked into one program. In practice all existing CORBA implementation use the same name of CORBA module that causes name conflicts during linking and loading phase. This problem can be easily solved when source code is available. A more general solution requires the allocation of unique prefixes of CORBA module provided by different vendors. The allocation process could be, for instance, controlled by the OMG.

3.2 InterORB_Proxy creation and managment

InterORB_Proxies in different systems may be created in different ways which impact on a level of transparency perceived by a client. One solution assumes an implicit *InterORB_Proxy* creation and leaves a client program without any changes. Within this approach *InterORB_Proxy* is created by the Bridge Object Adapter (Bridge OA) obtained modifying a usual Object Adapter of the system. When a foreign object reference appears inside a bridge this new Object Adapter has to enable the creation of dynamic object – an *InterORB_Proxy* – to encapsulate it if such an encapsulating entity does not yet exist.

The other solution assumes explicit *InterORB_Proxies* creation. It may be performed by a client or management tool call to the BridgeManager responsible for managing objects inside a bridge. The BridgeManager provides the following interface:

```
interface IOPFactory {

  InterORB_Proxy createIOP(in OctetSeq key); /* create Proxy      */
  oneway remIOP(in InterORB_Proxy proxy);    /* remove this Proxy */
};

interface BridgeManager : IOPFactory, SecurityManager, CacheManager,
                          MapperManager, PersistenceManager,
                          ConcurrencyManager {
  oneway terminate();           /* kill yourself  */
  short ping();                 /* are you active */
};
```

The `IOPFactory` interface enables the dynamic creation and destruction of *InterORB_Proxies*. The other interfaces provide functionality for managing different bridge components, such as caches, mapper, security, etc.

It is necessary to point out that not all available CORBA implementations enable the implicit *InterORB_Proxy* creation. The explicit approach is possible in any system.

In both cases *InterORB_Proxy* creation is performed by a call to its constructor. The

constructor invokes an Object Adapter of the client ORB to register the *InterORB_Proxy* in it. There are several parameters which must be specified at this time: references to ORB objects of two CORBA systems it connects (the *InterORB_Proxy* will use some of their functionalities) and reference to an object that will help it to translate certain data e.g. object references, contexts etc. (*Mapper*).

When a call to such a created *InterORB_Proxy* is recognized the ServerRequest is built and *InterORB_Proxy*'s *invoke* method is activated regardless of what operation has been demanded in the request.

3.3 Mapper

In addition to forwarding requests, mapping objects, defined in CORBA, is the main functionality of the bridge. This task is entirely ORB dependent and is performed by a dedicated object – *Mapper*. The *Mapper* is equipped with *map()* methods – one for each object to be mapped. It reads data from a source object, creates and fills in a target object. There is one or more *Mappers* in the bridge. However, each *InterORB_Proxy* is associated with exactly one of them.

In general, the *Mapper* is expected to translate CORBA objects between two arbitrary ORBs. Nevertheless, in the architecture with the ORB backbone only mapping between ORB and ORB backbone and vice versa is necessary.

3.4 Bridge concurrency model

A classical model of object oriented processing approved by CORBA 1.2 and retain by CORBA 2.0 specification does not address the problem of internal parallelism of objects. It is, however, a very important issue for bridges, when object references are used as operation parameters and new proxies have to be created for them. The mechanisms based on activation policy proposed by the CORBA standard exploits inter-server parallelism and is too heavy for mapping object references. Thus, to create an efficient bridge it is necessary to exploit the parallel execution of inter-ORB service invocations within its process.

It has been mentioned in the previous subsection that an *InterORB_Proxy* may host by many threads performing the translation process of different requests directed to the same service. This feature of the *InterORB_Proxy* implies different intra-bridge parallelism models. Referring to (Schmidt, 1996) the following four models should be discussed:

Thread per Request – A new thread is created to continue dispatching of each request.

Pool of threads – Each request is placed into a common linked list of pending requests. Then, a pre-allocated pool of threads competes to empty the list continuing dispatching of each request in the list.

Thread per object – Each request is placed into a linked list of pending requests associated with the target application object. Objects have dedicated threads servicing requests.

Thread per connection – Each connection has a dedicated thread servicing pending requests. A new thread is created/destroyed any time a new connection is established/disconnected.

The choice of the most suitable model of parallelism for a bridge is a matter of a trade-off between overheads of creation and destruction of a new thread and handling request lists as well as obtained level of parallelism. The request translation process performed by the *InterORB_Proxy* is not very time consuming – it takes at most a few milliseconds. On the other hand the CORBA processing model promotes a standard client-server semantics, based on RPC style of remote invocation. After a request is issued by a client, client's thread or process hangs and waits for a reply. From the networking point of view a client-server communication needs a new connection to be established when the first request is issued by the client to the server. This connection is dedicated to the particular client-server pair of threads (processes) used by subsequent request invocations. This connection is kept alive as long as both threads exist. It means that in CORBA based applications parallelism provided by *thread per connection* model is the most suitable because it allows avoiding the sophisticated buffering of pending requests within bridge and eliminating context switching among threads. The successive requests on the given connection are processed by the same thread.

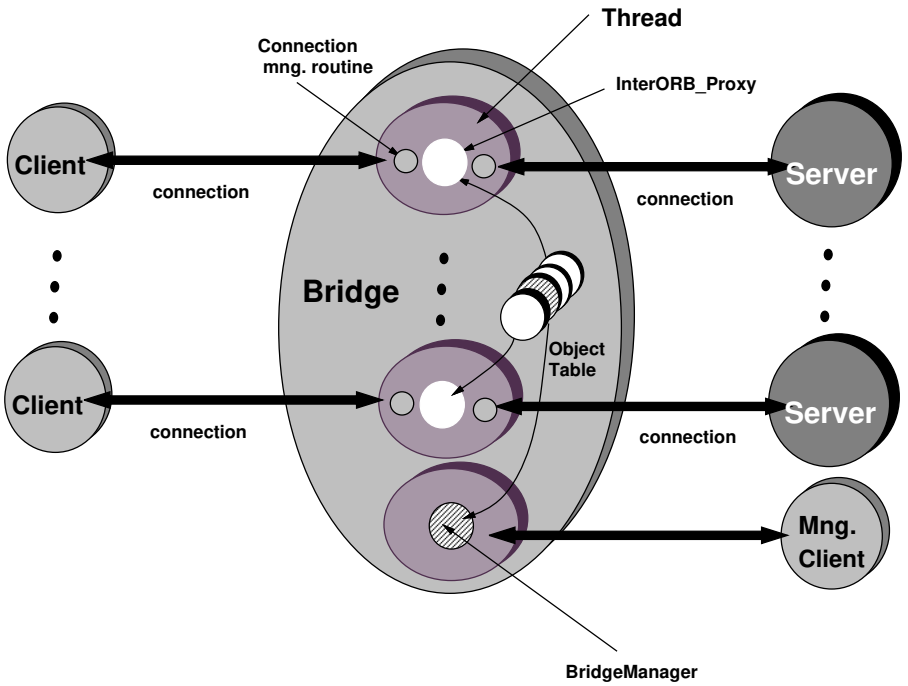

Figure 2 Intra–bridge parallelism model

The other models may introduce some limitations e.g. when a pool of threads is too small or the overhead of thread activation and deactivation is too costly. The thread-per-object model makes sense when a server is single-threaded, but this model introduces extra complexity related to handling request lists for each object. Taking into account the presented consideration the thread-per-connection model has been chosen for the implementation of the multithreaded multiproxy bridge (Figure 2). The number of existing threads depends on a number of active connections within the bridge.

4 BINDING IN MULTI–ORB ENVIRONMENT

In all ORB environments in order to faclitate the cooperation of clients and servers there have to be a way of identifying service objects. This identification is performed with the use of object references, which have to be given to clients explicitly or obtained by means of a special service. The first solution is cumbersome and not dynamic, because an object reference has to be known before a client is started. Obviously it is not very flexible for a distributed system. So, practically, all analyzed ORB implementations are equipped with dynamic binding mechanism.

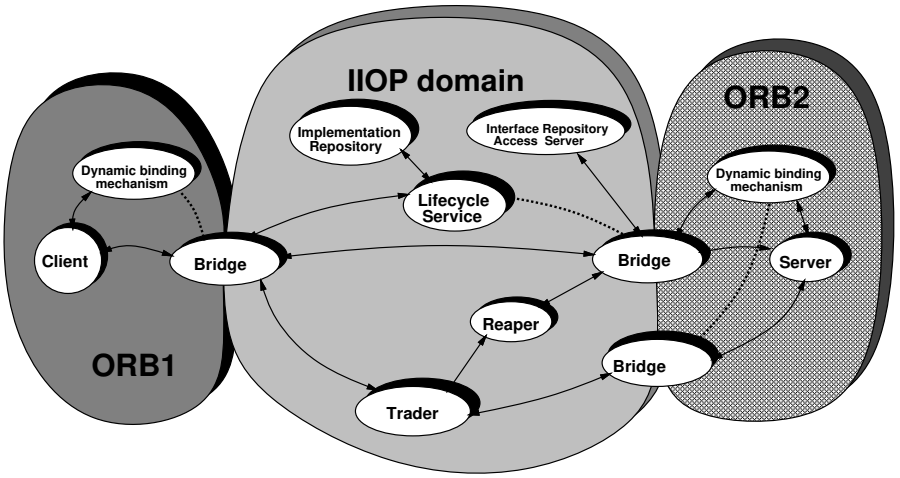

Figure 3 IIOP domain infrastructure for inter-ORB cooperation

Bridges which act as clients in IIOP domain may also be bootstrapped manually with references of appropriate servers. However, this solution has many drawbacks enumerated above. Thus, beside this approach also dynamic binding of bridges has been made possible. In order to elicit this behaviour special services has been introduced into IIOP domain. These services may be provided by using Trading and Lifecycle Common Object Services (CORBAservices, 1996).

4.1 IIOP-domain reference types

In order to enable cooperation of bridges and mapping object references a special format of the IOR object reference has been proposed. This special format refers to the *object_key* field of the IOR since it is the only field that can be manipulated. The used layout of the field is as follows:

```
struct Property{
          string propertyName;
          string propertyValue;
       };
```

```
typedef sequence<Property> PropertySeq;

struct BridgeObjectKey{
          unsigned char refType;
          PropertySeq properties;
          OctetSeq key;
       };
```

The structure is encoded in the field as an sequence of octets. This structure contains: type of the reference – two kinds of object references has been defined: *eager* or *lazy* (`refType`), a set of properties – a set of values particular for each domain, required by a dynamic binding mechanism of this domain (properties) and a stringified object reference of a server (most often it is *InterORB_Proxy*) (`key`). The *eager* object reference points out to an active object, the *lazy* one denotes an object which is not active. Information included in the *eager* reference is sufficient to contact the service. Information contained by the *lazy* reference allows activating the object and obtaining its *eager* reference. The two kinds of object references in IIOP domain have been devised because for services from connected domains it should be possible to postpone the process of *InterORB_Proxy* creation to represent them.

Both *eager* and *lazy* references are built according to the same rule. The type and number of properties included in them depends on particular ORBs, properties may describe such parameters as a name of a server, a name of a domain to which the server belongs, host name on which the server is running, a name of a particular instance of the server object, information about Interface Repository containing interface definition of the service, etc. When defining a mapping for object references one has to determine what information is necessary in a given ORB to obtain a server object reference using dynamic binding mechanism, and include it in a properties list. Regardless of the ORB being considered two properties are obligatory: a domain name and a server name. The server name is the minimum amount of information necessary to describe a service in mind to the Trader. The domain name is needed to launch a proper bridge and determine the origin of the reference encapsulated inside the IOR. The rest of the properties is meaningful only in particular domains and interpreted by mapping functions defined for them. The `key` field is not empty only inside the *eager* IOR. However, the complete list of properties has been retained in the *eager* reference to enable converting *eager* references to *lazy* ones. This specific format of the *object_key* field in IOR is significant only for bridges, other objects in IIOP domain do not interpret it. They use these specific references as ordinary ones. Similarly, bridges may cooperate with objects in the IIOP domain, which do not possess this special form of reference, in this case they regard these references as references of ordinary IIOP objects.

4.2 Trader

A bridge refers to services in IIOP and other domains using their names. To obtain mapping of these names into object references of services in IIOP domain or bridges for services in other domains a Trader service is employed in the ORB backbone. References of all services that are to be available to clients in IIOP domain have to be registered in it either in their *lazy* or *eager* form. Since the *lazy* reference contains only a name

of a service it represents and some set of properties and their values there has to be a way of mapping it to a reference of a target object in the other domain. Therefore in the proposed architecture it is assumed that systems connected via the ORB backbone possess at least the most primitive mechanisms of dynamic binding. Trader does not distinguish between *eager* and *lazy* references. Therefore it is not possible to change state of object references from eager to lazy and vice versa inside Trader or force it to return only a particular kind of object references. So, a change of the bridge's state must be notified to Trader explicitly by deregistration of the old reference and registration of the new one. A *lazy* object reference is kept in Trader as long as there is no *eager* counterpart. It is unregistered by the *InterORB_Proxy* when it is created and replaced with an appropriate *eager* reference. When the *InterORB_Proxy* is destructed its eager reference is unregistered and replaced with the appropriate *lazy* reference.

The Trader used in the project derives from ANSA system (ANSAware 4.01, 1992). Its interfaces and functionality are sufficient to obtain the expected behavior of the bridge. On the other hand bridges are not strictly dependent on the Trader interfaces and may easily be adjusted to OMG Trader (OMG Trader, 1996).

4.3 Lifecycle Service

The usage of the Lifecycle Service within the ORB backbone is rather straightforward. It requires the existence of a factory object whose main activity consists in creating bridges and *InterORB_Proxies* requested by clients. A client must be already equipped with a *lazy* reference of a required service; in this case the factory is able to create its *eager* correspondent. According to the employed activation policy the factory may launch a separate bridge (in unshared activation mode) or it may try to create an *InterORB_Proxy* in an existing bridge (shared mode). In the second case it has first to obtain a reference of a BridgeManager object inside the bridge by contacting the Trader. Obviously, if a required bridge does not exist it has to be activated.

The factory is supported by Implementation Repository, which stores a mapping from names of domains connected by a bridge (one of which is always IIOP) to information about: the name of the executable for this bridge, its location in the file system (a path name), parameter list, activation mode (shared or unshared).

4.4 Bridge cooperation scheme

The first object created inside a bridge is a BridgeManager which represents this bridge. As soon as the BridgeManager is created it is registered with the Trader. When a bridge is to establish a connection from some service in non-IIOP domain to some other service via the ORB backbone it first contacts the Trader to obtain a reference of a required service. Only services registered in the Trader may be made available to the bridge client. In the case of a service implemented in non-IIOP domain the Trader has to contain registration of *InterORB_Proxy* representing this service in the ORB backbone. If such a reference is returned in an *eager* form it may be used without any further actions. It may happen, however, that the Trader returns a *lazy* reference. Then, the Lifecycle Service is contacted, which activates a bridge (if necessary) and creates an approperiate *InterORB_Proxy* inside it. This *InterORB_Proxy* is registered with Trader with its *eager* reference and have its *lazy*

reference deregistered. The bridge which requested the creation of a new *InterORB_Proxy* in the IIOP domain has now to create an *InterORB_Proxy* in the other domain.

All *InterORB_Proxies* and the BridgeManager active within a particular bridge and registered within Trader have to be unregistered and have their registration replaced with *lazy* references when a bridge ceases to be active. For testing activity of bridges a Reaper process is used. It closely collaborates with the Trader from which it periodically obtains references of all registered BridgeManagers. Then it tries to invoke `ping` operation on each of them. In the case of failure it deregisteres the reference of the inactive BridgeManager. All *eager* references of *InterORB_Proxies* supported by the bridge represented inside Trader by these inactive BridgeManager have to be unregistered and substituted with their *lazy* counterparts. *InterORB_Proxies* belonging to these bridges are distinguished by the appropriate context space within the Trader.

4.5 Mapping object references

In CORBA-based environments object references may be operation parameters and results. However, they are only understandable within the scope of one technological domain. Therefore, when crossing technological borders object references originating from one ORB have to be associated with some other reference of the other ORB. This process is called mapping. When the IIOP protocol is used as the backbone protocol for each domain a mapping has to be defined for object references between this domain, the IIOP domain and vice versa.

Eager mapping of object references to the IIOP domain

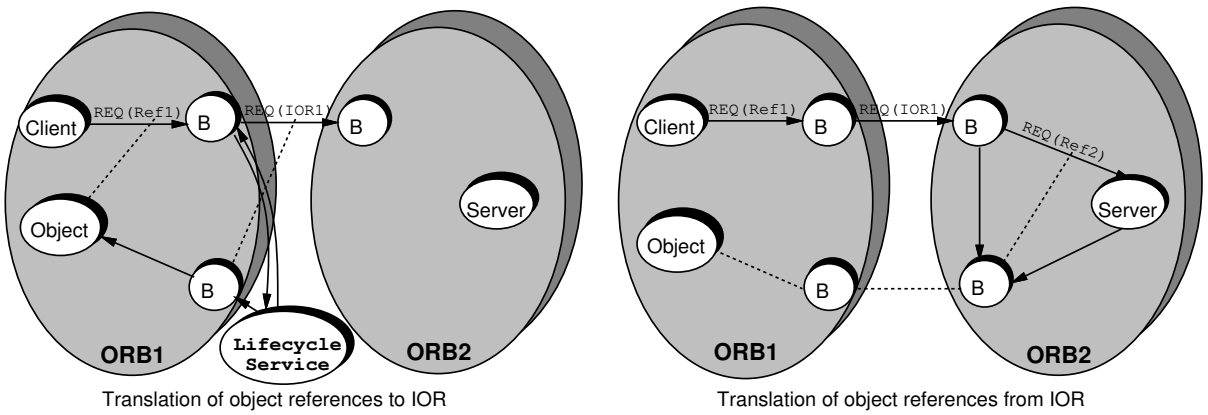

Figure 4 Eager mapping of object references on IIOP domain borders in case of unshared bridges

In this approach a new *InterORB_Proxy* is immediately created that will allow contacting the object pointed by mapped reference inside client's ORB (Figure 4). *InterORB_Proxy* may be created in the same bridge or in a separate server. Its reference replaces the reference specified in parameter list. On the other side of the IIOP domain this reference is mapped again to the format understandable to the recipient.

Lazy mapping of object references to the IIOP domain

In this approach creation of the *InterORB_Proxy* is postponed until an invocation on it is made. Instead of its reference a special – *lazy* format of reference is created and written in the parameter list. When on the other side of the IIOP domain a bridge directed to the object represented by the mapped reference is created it contacts Lifecycle Service and induces creation of new *InterORB_Proxy*.

Mapping object references from the IIOP domain

A request arriving from the IIOP domain to the server ORB may contain IOR references which have to be mapped to the server's ORB proprietary form. The bridge has to create a new *InterORB_Proxy* which encapsulates the reference. It performs this task using mechanism valid in this ORB. The *InterORB_Proxy* object possesses a reference specific for this domain which replaces the IOR in the request. However, it may happen that the received IOR is the encapsulation of some object reference of the domain. In such a situation, when the IOR is *eager* a stringified object reference of the server is retrieved form the IOR *ObjectKey* field. On the other hand, when it is a *lazy* reference a properties list contained by the *ObjectKey* field is used to contact some dynamic binding mechanism available in this ORB.

5 CASE STUDY FOR ORBIX

The proposed framework has been applied to implementation of a bridge for one of the most representative CORBA compliant systems – Orbix 1.3 from IONA Technologies Ltd (Orbix 1.3, 1995). The system does not use the CORBA 2.0 IIOP protocol (facilitating interoperability with other ORBs). This justifies the construction of a bridge for the system.

The system is built with a sufficient conformance to the CORBA 1.2 standard. However, it is required by the InterORB_Proxy template that systems it joins together be CORBA 2.0 complaint. Therefore it was necessary to add a wrapping layer above Orbix 1.3 system, which hides all inconsistencies of the system with CORBA 2.0.

The main deficiency of Orbix 1.3 was the unexistence of the DSI module. The assumption behind the presented architecture is that *InterORB_Proxy* is a generic object able to represent any interface. In CORBA 2.0 systems this may be achieved by inheritance from *DynamicImplementation* class, a component of DSI. The *DynamicImplementation* class inherits from *CORBA::Object* and has the feature of being able to be registered in the *ObjectTable* with an arbitrary interface name. The DSI implementation requires writing a *ServerRequest* class. The *ServerRequest* class has been implemented according to its specification in the CORBA 2.0 C++ mapping. To make it possible to handle operation involving setting or reading attribute values another operation has been added:

```
AttributeDef_ptr attr_def();
```

For *ServerRequest* marshaling and unmarshaling functions based on Typecode interpretation have been implemented. In order to retrieve data from the incoming request a stream-like interface of the Orbix *Request* object is used. After this changes sufficient compliance of Orbix 1.3 with CORBA 2.0 has been achieved.

The IIOP part of the bridge has been built using the InterORB-Engine provided by SunSoft, which, however, required many extensions:

- Support for multi-object server,
- Multi-thread implementation,
- A wrapping layer over the original systems, which equips it with CORBA 2.0 interfaces,
- Increasing modularity and object-orientation of the code, improving its memory management and efficiency.

It is required by the *InterORB_Proxy* that systems it connects have different names of CORBA modules. The usage of the InterORB-Engine has been justified by possibility of changing its source code. This made it feasible to rename its CORBA module and guarantee its uniqueness.

In the case of an Orbix bridge it is possible to create *InterORB_Proxies* in both explicit and implicit way. The first approach consists in implementing BridgeManager object responsible for creating *InterORB_Proxies* on explicit clients requests. With the second approach the Orbix specific *Loader* mechanism has been used, which takes care of all references referred to by clients, which do not yet exist, and automatically creates *InterORB_Proxies* for them.

The minimum functionality of the Mapper object necessary to achieve cooperation of Orbix 1.3 and the IIOP domain was mapping Object References and TypeCode. A process of mapping Object References form Orbix to IIOP does not possess any specific features and is as already described in Section 4. IOR object references are mapped by encapsulating them as a *marker* field inside Orbix references. Mapping TypeCodes necessitated implementing translation procedure from string-like representation in Orbix to the CDR encapsulation in the IIOP domain and vice versa.

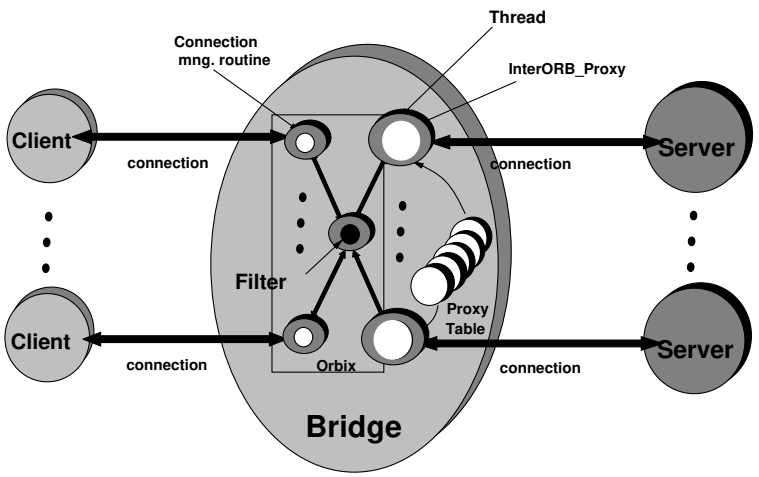

Figure 5 Orbix bridge concurrency model

When implementing the proposed multithreaded model of the bridge for Orbix (Fig. 5) several problems have been encountered resulting from the internal concurrency model of

this system. In the Orbix model separate threads are created for each connection, however all of them meet in the same *Filter* thread. No earlier than at this point is a programmer's intervention possible. Thus, it is necessary to demultiplex all incoming request within a new set of threads created for connections.

The implemented bridge acts in Orbix as an ordinary server. It is registered in the Implementation Repository and launched by the Orbix demon. However, it may also be activated manually as a persistent shared server. The Orbix client may reach the bridge functionality by calling *bind* method of an invoked interface. The *bind* directed to the bridge differs from the ordinary one only in that a bridge name has to be specified as the server name in *bind* parameter list:

```
Ex_interface *my_interface;
my_interface = Ex_interface::_bind(":Bridge", host_name, IT_X);
```

Thanks to the *Loader* mechanism the contacted bridge is able to handle requests directed to the requested interface. This minor change required to the client code does not spoil transparency of the bridge during subsequent request invocations.

6 BRIDGE PERFORMANCE STUDY

In this section the performance evaluation of the previously described bridge for Orbix is presented. This evaluation has been performed on the basis of a number of tests aimed at estimating the overall overhead introduced by the bridge and distinguishing its components, which contribute to this cost.

A testing environment consisted of a client written in Orbix 1.3MT, a bridge and a server implemented in VisiBroker for C++ 2.0 (Visigenic, 1996) communicating with the bridge using IIOP protocol. The client and the bridge as well as the Interface Repository were run on Sun Ultra 1 workstation and the server on Sun Sparcstation 10/514. The evaluation does not include connection establishment stage, so it should be expected that *InterORB_Proxy* creation time is longer.

A testing procedure consisted in invoking operations taking as parameters more and more sophisticated data – *long*, *string* and data presented below:

```
enum EnumType {first, second, third};

struct FixedStruct{
   short       s;
   long        l;
   float       d;
   EnumType    e;
};

struct VariableStruct{
   FixedStruct f;
   char        c;
   string      s;
};
```

Type of data	Client invocation [ms]	Invocation Bridge – server – Bridge [ms]
long	20.77	19.61
string	21.16	20.01
FixedStruct	25.23	24.04
VariableStruct	28.60	27.36
UnionType	30.69	29.40
VariableSequence	33.20	31.81
ObjectRef	38.33	35.97

Table 1 Invocation time of different kinds of operations

```
union UnionType switch (long){
   case -1: short s;
   case  0: long l;
   case 10: VariableStruct v;
};

typedef sequence<VariableStruct> VariableSequence;
```

These parameters have been passed in `in`, `out` and `inout` modes as well as a return value, according to the following function template:

```
T operation_name(in T inArg, out T outArg, inout T inoutArg);
```

The server functionality was simple: it copied the `in` argument to the `inout` argument, the `inout` argument to the `out` argument and returned the `in` argument.

Table 1 presents the average execution times of the described above operations – calculated from 1000 subsequent invocations. The second column presents operations execution time registered by the client and the third one times taken by operations between entering the bridge and leaving it, including server invocation time.

It may be noted that the complexity of data is a factor that contributes to the bridge overhead. Because the obtained times were long, Table 2 presents how different components of the bridge contributed to the times presented in the second column of Table 1. These components are:

- DSI *ServerRequest::param* function – TypeCode driven unmarshalling and marshaling data,
- *ServerRequest::op_def* – Interface Repository access,
- ORB::create_operation_list – *NVList* creation,
- Mapper – translation of TypeCodes and Object References as well as copying data,
- DII – TypeCode driven marshaling of data,
- *Request::invoke* – DII and server invocation.

As a conclusion derived from the presented performance evaluation it may be concluded

Type of data	DSI [ms]	IR [ms]	NVList creation [ms]	Mapper + data copying [ms]	Request::invoke [ms]
long	0.39	13.81	2.07	0.14	2.73
string	0.45	14.02	2.11	0.17	2.87
FixedStruct	0.86	15.87	2.98	1.60	3.19
VariableStruct	1.08	17.68	3.18	2.19	4.12
UnionType	1.27	17.45	3.46	3.46	4.56
VariableSequence	1.32	16.86	3.70	2.76	6.25
ObjectRef	0.96	15.13	2.08	3.88	13.36

Table 2 Time spend in different parts of the bridge during invocations of different kinds of operations

Type of data	Old invocation time [ms]	New invocation time [ms]	IR time [ms]	Bonus [%]
long	20.77	8.40	1.44	60
string	21.16	8.51	1.37	60
FixedStruct	25.23	10.87	1.51	57
VariableStruct	28.60	12.34	1.42	57
UnionType	30.69	14.89	1.65	51
VariableSequence	33.20	17.70	1.36	47
ObjectRef	38.33	24.52	1.32	36

Table 3 Cache impact on bridge efficiency

that the greatest contributor to the bridge overheads is access to the Interface Repository, which takes $40 - 70$ per cent of overall time spent in the bridge. Performance may be greatly increased by an employment of a cache. Such a cache for IR data has been implemented. Its usage allows to significantly decrease execution time of IR operations. IR time has been once again measured with use of cache. Table 3 presents invocation times for different operation provided that cache is already initialised.

When very complex data are being passed as operation parameters also time spent for translating TypeCodes, copying, marshaling and unmarshaling data is getting more and more significant. As it was presented in (Zieliński, 1996) time taken by these manipulation functions grows exponentially with the increase of a complexity of data type represented by the Orbix TypeCode. By complexity the number and depth of nests in data type is meant. The situation may be improved if interpretation of TypeCode in marshaling and unmarshaling of data is avoided. This may be possible if TypeCode interpretation trace is cached in the form of basic functions calls, and used in subsequent DSI and DII invocation of the same operation.

Cache in both cases performed well when the same operation is invoked many times. The single invocation of the operation always is subject to the cost presented in this section.

7 CONCLUSIONS

In this paper a general framework for inter-ORB bridge construction has been presented. The framework was based on the CORBA 2.0 standard and classified as a mediated, request level, generic bridge according to this document. As an intermediary protocol IIOP was chosen. It constitutes a new backbone domain equipped with all CORBA features which are necessary to implement a request level bridge.

A core of the bridge – the *InterORB_Proxy* is ORB independent because it uses only standard interfaces. *InterORB_Proxy* is invented for the ideal case: when connected systems are completely CORBA 2.0 compliant and symbols they define are unique. In the case they are not a wrapping layer must be constructed.

The other crucial bridge component – Mapper could not be implemented as a framework, because for its functionality Mapper exploits lots of ORB specific features. Thus, only Mapper interface has been specified to enable its usage inside *InterORB_Proxy*, however, its methods have to be implemented for each system separately.

Binding in multi-ORB environment may be effectively solved by using Trading and Lifecycle services. It requires the proper structure of information represented by the entry stored in a Trader data base. They may be divided into two categories called *lazy* and *eager* object references respectively.

The notion of *lazy* and *eager* reference plays also a very important role in reference mapping process. This concept has great potential in inter-ORB operation invocation speedup when references are passed as parameters.

The first measurement of the bridge performance proved that the mechanism of inter-ORB activity is costly. Analyzing time consumed by different bridge components show that the most significant contributor to the bridge overheads is obtaining information necessary to dynamically translate request invocations. This factor was reduced by applying cache.

The architecture of a bridge presented in this article may be effectively used when the systems it is applied to are implemented in a proper standard way and provides sufficiently good performance.

8 REFERENCES

CORBA 1.2 Revision Draft (1993) OMG Report 93-12-43, Object Management Group (OMG) Inc.

CORBA Revision 2.0 (1995) OMG Inc.

CORBAnet – the ORB Interoperability Showcase (1996) OMG Inc., *http://corbanet.dstc.edu.au/*

CORBAservices: Common Object Services Specification (1996) OMG Inc.

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

A. Uszok, G. Czajkowski, K. Zieliński, *Interoperability Gateway Construction for Object Oriented Distributed Systems* (1994) Proceedings of 6th Nordic Workshop on Programming Environment Research

Orbix 1.3 Programmers Guide (1995) IONA Technologies Ltd.

Orbix 2.1 Programmers Guide (1996) IONA Technologies Ltd.
ANSAware 4.0 — Application Programmer's Manual (1992) APM Ltd. Cambridge
VisiBroker for C++ 2.0 User Manual (1996) Visigenic Software Inc.
D. Schmidt, S. Vinoski, *Object Interconnections* (1996) June SIGS C++ Report
Z. Yang, A. Vogel, *Achieving Interoperability between CORBA and DCE Applications Using Bridges* (1996) Proceedings of IFIP/IEEE ICDCP'96, Drezden, Germany, Chapman & Hall
J. Sventek, *Revised submission for the Trader service* (1996) OMG Inc., TC Document 96-07-08
K. Zielinski, M. Steinder, A. Uszok, *A Framework for Inter-ORB Request Level Bridge Construction* (1996) Proceedings of IFIP/IEEE ICDCP'96, Drezden, Germany, Chapman & Hall
M. Steinder, A. Uszok, K. Zielinski, *Crossing Technological Domains Using the Inter-ORB Request Level Bridge - Preliminary Performance Study* (1996) Proceedings of the Workshop Trends in Distributed Systems, Aachen, Germany, LNCS 1161, Springer-Verlag

9 BIOGRAPHIES

Małgorzata Steinder graduated from the Computer Science Department, University of Mining and Metallurgy in 1994. Her MSc thesis concerned mathematical models describing performance in the ATM network. Currently, she works as a research and teaching assistant in the Computer Science Department, UMM Cracow. She is interested in interoperability and trading in distributed processing. She is working on her PhD thesis.

Andrzej Uszok graduated from the Computer Science Department, University of Mining and Metallurgy in 1993. His MSc thesis concerned load-balancing in the ANSA system. Currently, he works as a research and teaching assistant in the Computer Science Department, UMM Cracow. He is an author of 15 articles in the area of distributed systems. He has just finished his PhD thesis: *Transparent Interoperability in Distributed Processing.*

Krzysztof Zieliński is a professor of computer science at the Computer Science Department, University of Mining and Metallurgy. He spent two years in the Cambridge Olivetti Research Lab in the years 1988-1990, where he worked on the first prototype of the ATM network. Now, he is the main designer of the Cracow ATM MAN. He is also the Technical Manager of the Copernicus **TOCOOS** project. He is an author of about 100 publications.