Interoperability Gateway Construction for Object-Oriented Distributed Systems *

A. Uszok, G. Czajkowski, K. Zieliński
Institute of Computer Science
University of Mining and Metallurgy
al. Mickiewicza 30
Kraków, Poland
e-mail: {uszok, grzes, kz}@ics.agh.edu.pl

March 1994

The paper addresses the problem of making object-oriented distributed imperative systems such as ANSA or Orbix (implementation of CORBA standard) parts of some bigger environment for writing and execution of distributed applications. This enables programs in these systems to call services provided by other languages, possibly with different computational models, like for instance declarative or functional. On the other hand, application in ANSA or Orbix may export certain services, thus becoming a server in a set of cooperating systems.

In this paper we present such hybrid system called Interoperability Architecture (IA), parts of which are ANSA and Orbix. We try to concentrate on building specialized modules – `Gateway`*s which translate requests from the environment to an universal form of exchanged messages in IA. We investigate some requirements and constraints in construction of such a module and propose general structure of it. This structure enables its automatic generation on the basis of specified interfaces of used operations. As a example of* `Gateway` *we present the construction of this module for ANSA and some hints for building it for Orbix. We also present example of application executing in IA which in this case consists of ANSA and Strand.*

1 Introduction

Today's information systems consist of a vast network of distributed computing and information resources. With growing number of such systems, there is an increasing need for technology to integrate them, creating an integrated distributed computing system.

While global interoperability may be achieved by the the universal adoption of a particular computational model [4], it is likely that no one model will achieve universal acceptance. Hence, the best achievable interoperability solution is to create mappings between the features of the various models and provide uniform platform for different computing paradigms [6, 11], such as imperative, functional or logic programming.

In this paper we would like to present our view on how distributed object-oriented imperative systems might become members of cooperating environments. After introducing a notion of Interoperability Architecture, details of which are described in [12], we present the construction

*This work was partially sponsored by KBN under grant no.: 8 S503 015 06

of *Gateway* modules for imperative object-oriented systems like ANSA [1, 2] or Orbix [8, 9] which allow these systems to become active parts of IA.

1.1 Overview of Interoperability Architecture

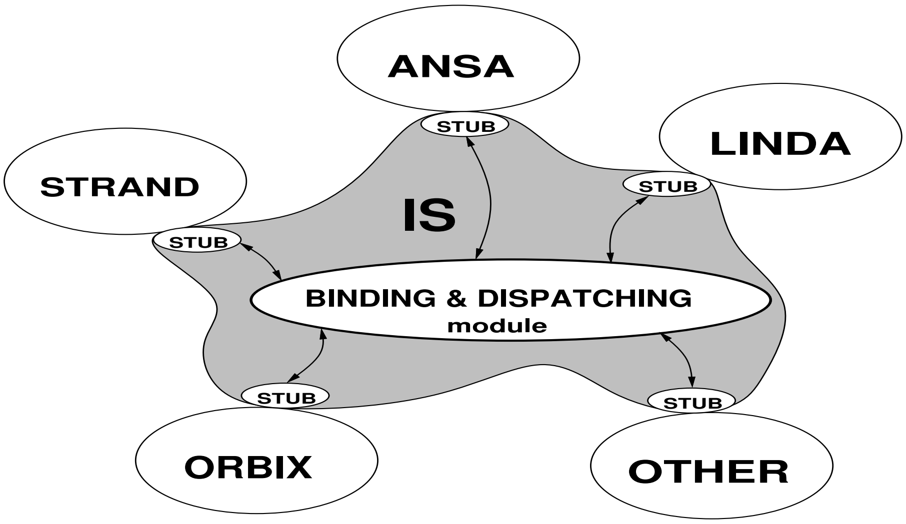

Figure 1: Interoperability Architecture (centralized version) with Strand, Orbix, ANSA and Linda environments

Interoperability Architecture (called IA) is built around a family of distributed languages and software tools integrated by the specialized component called Interoperability Server (IS). IA offers an uniform platform for the application software development. The main effort is to allow interoperability among diverse distributed systems without creating extra complexity. IA provides functionality addressing the problem of cooperation of various software environments or integration of software components. They can be developed as separate programs in any language supported by IA. It provides mechanisms which make some distributed environments open what means that a new application program may cooperate with programs developed earlier, possibly after a minimum amount of changes.

Similarly, we try to design IS under the condition that it should preserve the computational models of linked environments and be transparent for a programmer wherever it is possible. The very important design issue concerned data types.

Below, we briefly describe our generalized solutions concerning data types and error handling for all systems in IA, regardless of their computational model. Others aspects of entering object-oriented imperative system in IA which are different with regard to declarative, logic-programming or functional systems are covered in the next sections.

Data representation and translation

At the risk of being boring, we notice that different languages have different data types. When linking N environments, the question is: should we make it possible for all languages involved to use all data types they have and translate them as necessary or intersect the sets of all possible types and supply operations on these types only. The second approach has two strongly tempting features, which made us decide on taking it: it is easy to program and fast in use. Besides, it frees us from coping with some problematic questions, e.g: how is the pointer to be represented in Strand [5]? Or, similarly, in some implementations of Linda [3], tuple space is an ordinary data structure; one can have an array of tuple spaces. How is that to translate ?

We have decided to use types that are common to all languages we currently use (it is difficult to imagine a language that would not provide that): integers, reals, strings and sequences of these types. Encoding and decoding of these types are fast operations and we think all types of process conversation/interaction can be expressed using them, although sometimes with a little bit more difficulty as when using more abstract types.

Error and exception handling

There can be different reasons for errors and exceptions in IA. Most of them origin from the fact that computations are carried out in a distributed system, from which the well-known disadvantages are inherited.

Let us assume that procedure P_1 in an environment E_1 calls some service P_2 supplied by the system E_2. First of all, E_2 may crash. To detect such situations, the server linking environments maintains a *keep–alive* protocol. If E_2 does not respond for a certain period of time, it is considered to be down. If E_2 has crashed part way through the call P_2, the appropriate error code is returned to P_1. If E_2 is rebooted later, the calls that were prematurely ended because of the crash are not automatically called once again. It is the responsibility of the caller to take care of getting results right (possibly calling the required service more than once) in the presence of crashes. Thus, IA offers *at most once* [10] semantics.

If E_2 is functioning correctly, the possible source of errors may be the called service, P_2. It may be deregistered (or not yet registered) at the moment of call, it may expect different number and types of arguments that supplied or it may return result of type different from the one caller expects (this errors may be common when programming using typeless languages with strongly typed languages). All of these errors are reported to the caller, using error reporting mechanism available in a given environment.

Finally, we would like to point out that if the called procedure does not return for a very long time but its environment has not crashed it is assumed that everything is fine, that is, the computations run the way they should. We do not put an upper time limit on a call execution.

2 Object-oriented systems in IA

In order to make object-oriented systems parts of IA in the most suitable way we should first consider a model of computation offered by them and both how this model effects possible solution of this problem and how the solution would affect the computational model.

An ordinary application in an object-oriented system consist of some number of communicating independent objects. They cooperate with each other performing a task and may have an access to another objects being publicly accessible servers or even members of another application. Interaction between objects is supported by `stub` module linked to the object. This module is generated according to interfaces describing operations made available to others by the object and operations used by it. An interface to the object includes a description of operations with their names, lists of arguments and types of return results. `Stub`'s procedures handle invocation of such remote operations as: marshalling (packing of arguments/results into buffer for transmission), unmarshalling and communication between objects. This approach allows to separate implementation and interface and to have different versions of an implementation for the interface. There usually is a process or processes which act as a repository in which exported functions are entered by objects, and through which their localization can be obtained by others. They serve as a point of rendezvous for interacting objects in the distributed environment.

As we pointed out earlier one of the main design issues of IA is to achieve transparency and to preserve the original model of computation wherever possible. To fulfill these requirements the following functionality should be provided:

- Services offered in a object-oriented environment by other systems working in IA should be seen as ordinary ones exported by other objects. Also operations contributed by objects to the parts of the application which work in other IA's systems should be exported in the usual way.
- Some mechanisms should be available which will enable objects to communicate with IS.

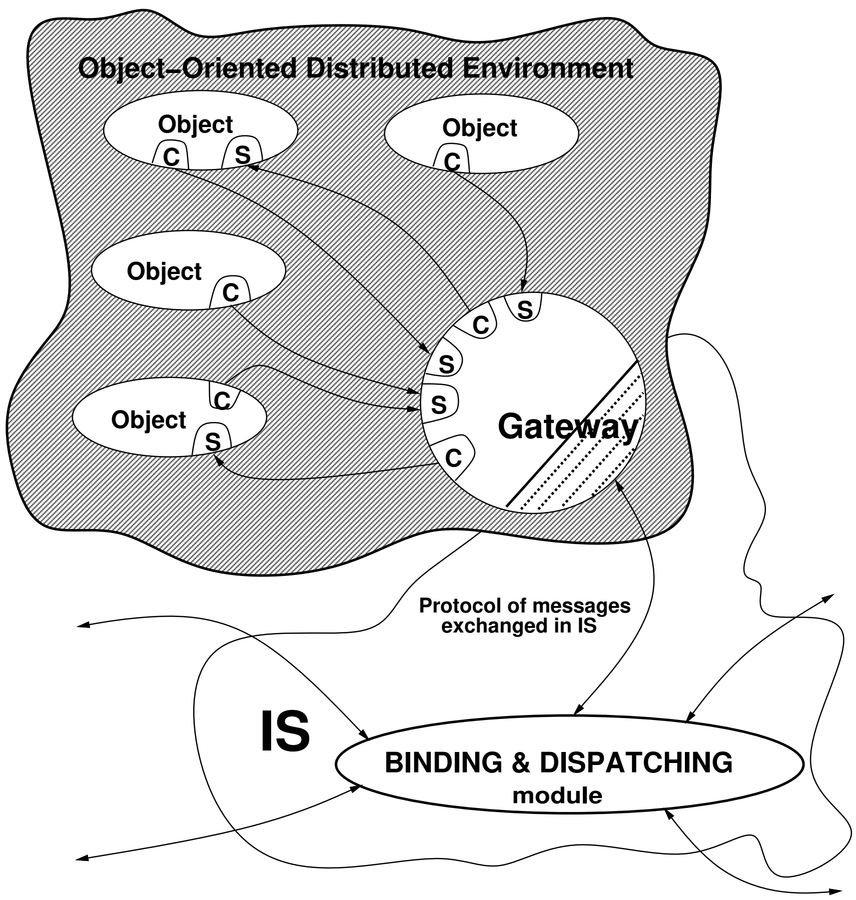

Figure 2: *Gateway* module in Object-Oriented Distributed Environment

An introduction of the special object which will represent IA in the environment can solves these problems. This object should look like an ordinary one for all objects and mechanisms of the environment. It should register in the environment's repository services provided by other

IA systems and export from it services needed by other environments. On the other hand it should provide mechanisms for communication with IS. Its main functionality is to translate messages from the internal environment representation to the IS protocol. It explains why this module was called *Gateways*.

According to the presented model of computation, interfaces of object's type with which application's objects would like to cooperate should be known at the moment of writing the application (except for situation in Orbix when DII – Dynamic Invocation Interface is used). Then operations exported by other IA systems which are used by an environment should be grouped in interfaces. They should be described in the standard Interface Definition Language (IDL). These interfaces should be supported by some objects existing in the environment.

We thought about general construction of *Gateway* which will allow to represent and export any kind of interfaces. But after careful examining of computational model of these environments (which is imperative) we concluded that achieving that goal is impossible without deep changes into internal implementation of these environments. In contrast to declarative systems they do not allow to invoke any operation from the program without its previous declaration and generation of appropriate code. Application should be consolidated with *stub*s, which have been mentioned before (except for situation in Orbix when DII is used). If *Gateway* is to invoke some services on behalf of other systems then it should include `stub` in *client*'s version (`C` on the fig. 2) generated on the basis of an appropriate interface. If it is to offer some services provided by other environment than it should also include `stub` in the *server*'s version (`S` on the fig. 2). That is why *Gateway* should be generated separately for each application what will allow a generation and linking up those `stub`s.

Gateway is an ordinary object of the given environment. Thus, no program executing in this environment (except for *Gateway*) is aware of the fact that services offered by it are defined in others languages and systems. The original computational model of the environment has been left unchanged.

In the next section a functionality, a structure and a way of construction of the *Gateway* object will be presented.

3 Construction of *Gateway*

The task of *Gateway* is to translate requests and messages between the environment and IS and to represent services of IS systems. Several interesting issues are addressed:

- Initiation of work in IA,
- Maintaining communication links with IS,
- Registering in IS offered services,
- Translating requests, their parameters and return values,
- Identifying the requested operation and invoking of it,
- Invocation of remote services,
- Triggering of the exceptions.

The proper work of *Gateway* is crucial for an execution of the application. Next we will investigate how the elements enumerated above can be accomplished.

3.1 Main requirements in the *Gateway* construction

Gateway module is an intermediary element between two systems. As such it should in the least possible way limit their cooperation and communication. Main conditions which it should fulfilled are:

- **Reliability** – it can not introduce any kind of error in an execution of distributed application. What can cause for instance its termination because of some internal error in its code. When doing marshalling, transmission and unmarshalling – at each of this level the correctness of received data should be checked. In the case of an error it should try to force retransmission of the message and decode it again. When irreversible error occurs a proper exception action should be to return to the environment as the exception of this system.
- **Speed** – it can not delay execution of application by slow transmission and marshalling of requests so operations doing these should be written and profiled especially carefully.
- **Concurrency** – *Gateway* should be able to server arriving asynchronous requests concurrently.
- **Security** – systems connected by IA execute in distributed, multiusers environment and some services provided by them may be not be available for all users. Passing by *Gateway* authentication and security information will be the big advantage. Of course not many distributed object-oriented systems of today have an authentication service. For instance ANSA doesn't have it. Orbix can be regarded as one of the first exceptions. When application of such systems will be growing, security issues will gain considerable meaning.
- **Automatic generation** – it should be possible to generate the particular clone of *Gateway* (that is, the *Gateway* for a particular application) automatically using the information specified in interfaces.

Some of these issues (reliability, speed, concurrency) may be addressed by using mulitithreading, if it is available in the environment (like, for example, in ANSA). Each request may be assigned a separate thread dedicated to service it.

3.2 *Gateway*'s architecture

As has been proved earlier it is impossible to construct *Gateway* generally, independently of applications for which it serves. But it is perfectly feasible to determine the structure and main components of this module which are responsible for performing particular tasks (see fig. 3).

We will distinguish constant and changeable parts of the module. Constant parts lie on the IS side of *Gateway* and are responsible for communication, marshalling and unmarshalling request from IS protocol. The IS protocol is a message-passing protocol, in which every type of possible information (e.g. request to register a service or return of the value) is represented. Most of messages contain invocation parameters (see section on data types in IA), which are in the simple network format. These constant parts of *Gateway* are: *I/O_handler*, *Request_handler* and *Answer_handler*.

Changeable part are responsible for contact with the environment. They service calls from other IA environments and act as an intermediary when the environment makes a call to some IA service. These changeable parts are: *Client_Stub* and *Server_Stub*.

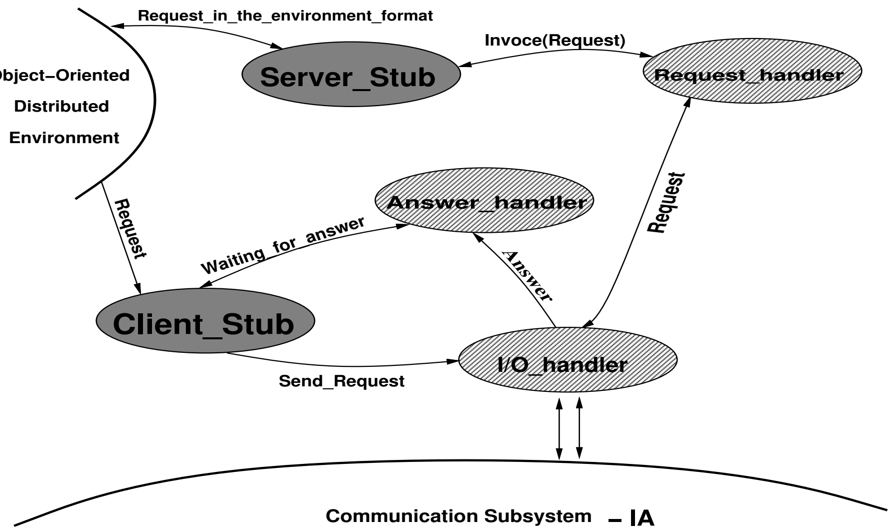

Figure 3: General Structure of Interoperability *Gateway*

3.3 Activity of *Gateway*

When *Gateway* starts operating it first tries to connect to IS. It is the responsibility of *I/O_handler* as any other aspects of communication. It exchanges messages with IS, maintains *keep-alive* (described earlier) and acknowledgment/time-out protocols. There are also services liable for providing some kind of authentication during an access of remote servers. After establishing the link *Gateway* registers in IS services offered by the environment. Next, the module begins waiting for requests.

A request received from IS by *I/O_handler* is handled by *Request_handler*. New thread is spawned in which a function from *Server_Stub* is executed. It takes over responsibility for serving this request. *Server_Stub* will be described in the next section.

On the other hand requests coming from the environment's objects are handled by *Client_Stub* (its structures will be described later) and after marshalling into buffer are sent to *I/O_handler* for transmission. Unique number is bound to it which is registered in *answer_handler*. Answers received by *Gateway* are sent on to it; further, the thread waiting for that particular answer chooses it according to this number.

3.4 Automatic generation of *Gateway*

A manual writing of *Gateway* each time when it is needed for a new application would be extremely inconvenient for the user. Due to the determining of the structure of this module and distinguishing of both universal (i.e. constant) and changeable parts (different patterns on the figure 3) it is possible to generate the *Gateway* automatically. The preprocessor which we provide takes as an input a set of interfaces written in IDL (they define exported and imported operations) and produces *Server_Stub* and *Client_Stub* procedures . Their code is produced in such a way that a compiler will include the appropriate calls in the executable code, which can be linked with universal parts of the module.

Such a solution hides complexity of internal structure of *Gateway* from the programmer and allows him to use it in a transparent manner. He or she should only specify interfaces used by this module but they are anyway written in standard IDL language of the environment. It also allows fast reconfiguration of the application when interfaces are changed.

Next, a way in which the changeable parts are generated and their structures together with *Gateway* example code for ANSA will be presented.

3.5 Code of *Server_Stub* part

Server_Stub unmarshalls messages received from IS and according to the requested operation name chooses an procedure to invoke. After return of this call it sends the return value to IS. All environments discussed in this paper are imperative; because of that there should be a code which contains calls to all procedures supported by that *Gateway*. The presence of this code will force the compiler to link in the appropriate *stub*s.

In ANSA, *Server_Stub* is implemented as a separate interface of the *Gateway* capsule with one operation *Invoke_Call*. It is responsible for creating a new thread which will further serve the request. *Invoke_Call* uses universal function *decode_request* which returns pointer to a list of decoded arguments of the invocation with first element containing name of the called procedure. The returned result is marshalled using universal function *encode_return_value* and finally send on to *I/O_handler*.

A simple example of an interface in ANSA IDL language is shown:

```
Manager : INTERFACE =

BEGIN
-- Operations
   StartWorkers : OPERATION [
                       Width : INTEGER;
                       Heigth: INTEGER;
                       Size  : INTEGER]
                 RETURNS [INTEGER];
END.
```

The code presented below is a simplified version of the code generated by the preprocessor on the basis of `Manager` interface.

```
/* Declaration of usage of the interface 'Manager',  .. (possible others) */
! USE Manager
! DECLARE { ir_Manager } : Manager CLIENT
. . .

int Invoke_Call(_attr, Pointer_to_Buffer)
ansa_InterfaceAttr *_attr;
ansa_Cardinal Pointer_to_Buffer;
 {
   . . .
   /* Decode request's buffer from IS */
   if(decode_request((char *)Pointer_to_Buffer, &argc, &argl, &rnumber) == -1)
    trigger_exception(Buffer_Internal_Error);

   /* Choose method */
   if(!strcmp(argl[0]->values.string.string_val, "Manager_StartWorkers")){
```

```
    /* Invoke function which export references to a capsule of
        the 'Manager' type */
    export_references("Manager", ir_Manager);

    /* Call the 'StartWorkers'  function of 'Manager' interface */
!   {argl[0]->values.int_val} <- \
            ir_Manager$StartWorkers(argl[1]->values.int_val, \
                 argl[2]->values.int_val, argl[3]->values.int_val)

     /* Type of return value */
     argl[0]->type = NET_INT;

    /* Encode return value */
    buflen = encode_return_value(argl[0], rnumber, &buf);

    /* Send back this buffer to IS*/
    c_send(buf, buflen);
   }
   else
    /* Code for calls of other interfaces operations */
   . . . . .
   . . . . .
   else
    trigger_exception(Unknown_Operation_Name);

  return SuccessfulInvocation;
 }
```

3.6 Code of *Client_Stub* part

A *Client_Stub* is a collection of operations which implementat services from interfaces imported by the *Gateway* capsule on behalf of other systems. They look like an ordinary procedure implementations but inside they wrap sending requests to IS. They marshall name and arguments of the call into a buffer, send it to *I/O_handler* and wait for an answer.

Below an interface in ANSA IDL language is shown, the implementation of which in *Gateway* will be presented next:

```
StrandFarmer : INTERFACE =

BEGIN
-- Operations
   feedMe : OPERATION [
                      StartPoint    : INTEGER;
                      IndexInBuffer: INTEGER]
          RETURNS [INTEGER];
END.
```

The implementation of this operation uses universal functions: *encode_request* to marshall a request to IS and *decode_answer_integer* to unmarshall returned answer. These functions are provided by specially written library with marshall/unmarshall functions.

Code presented below is an implementation of the `feedMe` operation. It is a simplified version of the code generated by the preprocessor on the basis of a definition of this procedure in the `StrandFarmer` interface:

```
int StrandFarmer_feedMe(_attr, StartPoint, IndexInBuffer, result)
 ansa_InterfaceAttr *_attr;
 ansa_Integer StartPoint;
 ansa_Integer IndexInBuffer;
 ansa_Integer *result;
 {
   . . .
  /* Create list 'argl' with arguments to operation 'feedMe':
      StartPoint and IndexInBuffer */
   . . .

  /* Marshall request to IS */
  ret_var_number = generate_new_ret_var_number();
  buflen = encode_request(argl, 3, ret_var_number, &buffer);

  /* Put it into answer_handler database with pointer to unmarshall
          function for return value */
  wait = newWA(result, ret_var_num, decode_answer_integer);

  /* Send message */
  NB = c_send(buffer, buflen);

  /* Was a sending of this message ok ? */
  if (!buflen)
   return UnsuccessfulInvocation;
  else{
   /* Wait for IS's answer */
   while(!wait->answered)
    /* Switch thread */
    instruct_Pause();

   /* Received result */
   if (wait->error){
    /* Delete element in answer_handler database */
    deleteWA(wait);
    return UnsuccessfulInvocation;
   }
   else{
    /* Delete element in answer_handler database */
    deleteWA(wait);
    return SuccessfulInvocation;
   }
  }
 }
```

3.7 Hints in a *Gateway* construction for Orbix

Orbix is an implementation of the CORBA standard. Besides, its computational model is very similar to that of ANSA. Almost all ideas presented in the ANSA implementation of *Gateway* can be applied. One exception may be in the usage of DII – Dynamic Invocation Interface what can allow to generalize a construction of *Gateway*. Any Orbix function from the interfaces available in the environment at the given moment can be called from *Gateway* using DII. Accordingly, a construction of *Server_Stub* should be changed. After this other environments can invoke any available service from Orbix but they should specify interface which includes required operation. If this interface and operation exist then it will be called using DII mechanism (which is quite complicated).

Such changes free a user from specification of interfaces offered by Orbix to others IA environments what allows a more flexible usage of *Gateway*. But it also decreases user's control over the application.

4 Example

This example uses a model of processors farm for the calculation of a fractal bitmap. In the next paragraphs the interaction of application's components which execute in different environments (in this case ANSA and Strand) and their implementation is described (Fig. 4).

4.1 Short overview of Strand

Strand [5] is a distributed, declarative, committed-choice logic-programming language and system. A basic abstraction of a Strand computational model is a *process pool* that is, a dynamically changing set of concurrent processes. They communicate via shared variables and data structures. If there are more than one clause matching a process to be reduced – only one of them is chosen, in a nondeterministic way.

4.2 Interaction of the components in the example application

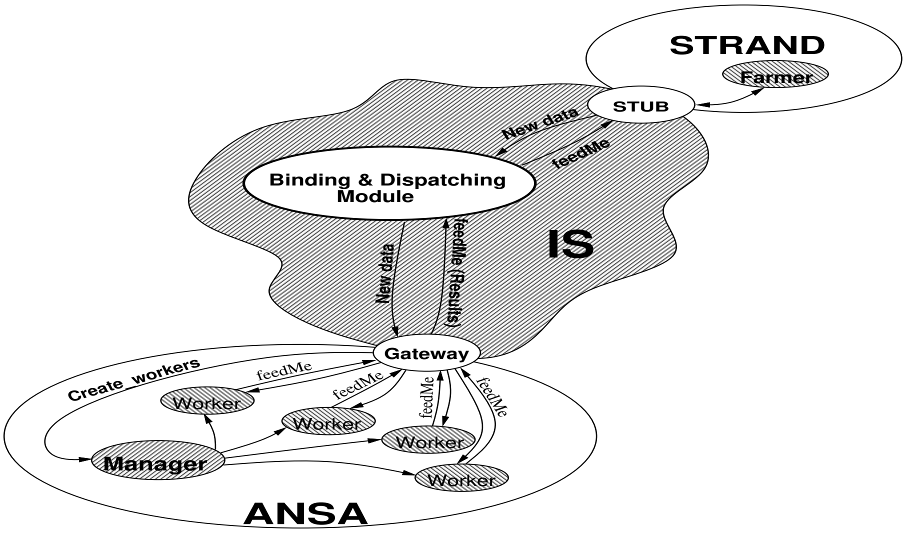

Figure 4: Cooperation of IA's elements during computation of Mandelbrot fractal image

A *farmer* module is written in Strand language and is executed in this environment. It is responsible for: initialization of the computation, sending tasks to the *workers* and receiving from them the calculated parts of a fractal bitmap. After receiving all parts of this bitmap *farmer* writes a complete bitmap to a file. *Worker* processes are written in C and executed in ANSA.

Farmer process communicates with the ANSA environment using two streams. In one of them it puts its requests which are carried out by ANSA processes. From the second stream *farmer* retrieves return values from the requests. Any kind of request can be put into the output stream. In this example these are: creation of the processes *worker*, computation of a task and destruction of *worker* processes.

If *farmer* written in Strand is to cooperate with ANSA than IS module which is responsible for binding and dispatching should be run beforehand. Services offered by ANSA and necessary for proper execution of this computation should be registered in this module. In this case it is a function `StartWorkers` provided by the special ANSA capsule `Manager`.

In order to ensure cooperation between ANSA's capsules and Strand's program services provided by the Strand system should also be registered in IS module. In this example this is an operation `feedMe` which returns a computed part of bitmap and asks about new input data. The operation `feedMe` belongs to the interface `StrandFarmer` which is provided in the ANSA environment by constructed for it *Gateway*.

The output stream from *farmer* is processed by IS which directs it to the appropriate ANSA's capsules. In particular if IS receives request to create *worker*s than it invokes the operation `StartWorkers` provided by the capsule `Manager`. This capsule uses standard ANSA mechanisms to create *worker*s on all nodes belonging to a system.

Finally *worker*s start computation of first tasks and invoke operation `feedMe` after having completed them.

Description of implementation of these operations which ensures cooperation between Strand and ANSA environments is presented in next points of this section.

4.3 Strand programs

Here we shortly describe Strand programs. First of them starts the fractal service. Initially, *init_ia* is called which is assumed to bootstrap Strand system in IA (line 1 in the listing below). The process *fractal_compute* is registered (line 2). It can be called from other IA systems and it starts computation of a fractal bitmap. Another service provided by Strand – `feedMe` and registered (line 3), will be used by IA processes-workers, which carry out the actual bitmap computing. All *fractal_compute* and `feedMe` requests are routed on the IStream to the *compute* process (line 4). Finally, *cap* process is started (line 5) which captures all bad requests.

```
start :-
  ia:init_ia(IStream, OStream, _),                  % (1)
  OStream := [ { 2, fractal_compute },              % (2)
               { 2, feedMe }],                      % (3)
  compute([], IStream, OStream, UnknownRequests),   % (4)
  ia:cap(UnknownRequests).                          % (5)
```

The first argument of *compute* is a synchronizing parameter. Second is a list of incoming requests. *OStream* is used to init ANSA procedures that compute bitmap. Finally, all requests that *compute* is not prepared to handle are sent for further processing on the *UnknownRequests* stream.

```
compute([], [ fractal_compute(Weight, Height, FName) | Rest ],
                                        OStream, UnknownRequests) :-
  make_fname(Weight, Height, FName), % FName == "data" ++ itoa(W) ++ itoa(H)
  OStream := [ startworkers(W, H, 1000, Workers) | OStream1 ],
```

```
  CurPoint is Workers * 1000,
  Total is Weight * Height,
  farmer(Rest, IStream1, { FName, FileName, Done }, CurPoint, Total, Workers, []),
  compute(Done, IStream1, OStream1, UnknownRequests).
compute([], [ H | Rest ], OStream, UnknownRequests) :- otherwise |
  UnknownRequests := [ H | UnknownRequests1 ],
  compute([], Rest, OStream, UnknownRequests1).
```

When process *compute* receives request *fractal_compute* than it prepares initial variables and create new process *farmer*. It is responsible for managing a computation of the requested fractal bitmap because of that it extensively communicates with IS. Below code of the *farmer* procedure is shown:

```
farmer(IStream, IStream1, X, CurPoint, Total, Workers, Result) :-
  CurPoint >= Total |
  finish_job(IStream, IStream1, X, Workers, Result).
farmer([ feedMe(StartPoint, ComputedPack, Work) | IStream ], IStream1, X,
  CurPoint, Total, Workers, Result) :-
  CurPoint < Total |
  Work := CurPoint,
  CurPoint1 is CurPoint + 1000,
  Result1 := [ { StartPoint, ComputedPack } | Result ],
  farmer(IStream, IStream1, X, CurPoint1, Total, Workers, Result1).
farmer([ H | IStream ], IStream1, X, CP, T, W, Res) :-
  otherwise |
  IStream1 := [ H | IStream2 ],
  farmer(IStream, IStream2, X, CP, T, W, Res).
```

4.4 ANSA *worker*

The activity of ANSA environment begins when its *Gateway* receives a request `StartWorkers` from Strand for the capsule `Manager`. A part of the *Gateway*'s code generated by the preprocessor on the basis of `Manager`'s interface is responsible for invoking the appropriate function of this interface. This fragment of the *Gateway*'s code was presented before.

The invocation of the operation `StartWorkers` from the `Manager`'s interface by *Gateway* results in finding this capsule by ANSA and execution of this function. It creates capsules *worker* on all nodes working in the ANSA environment.

```
int Manager_StartWorkers(_attr, Width, Heigth, Size, result)
 ansa_InterfaceAttr *_attr;
 ansa_Integer Width, Heigth, Size;
 ansa_Integer *result;
 {
  int host_nb, i;
  char propbuff[60];

  /* Scan Trader database looking for Factory services
           existing on all system's nodes */
! {Lookup_Res} <- traderRef$Lookup("Factory", "/", "", Lookup_All)

  /* Create 'worker's */
  host_nb = Lookup_Res.u.u_S_S.length;
```

```
  for(i=0; i<host_nb; i++){
   /* Import reference to Factory service */
   (void)strcpy(propbuff, "Node=='");
   (void)strcat(propbuff, name);
   (void)strcat(propbuff, "'");
!  {FacRef} <- traderRef$Import("Factory", "/", propbuff)

   /* Create capsule */
!  {CapRef, Cid} <- FacRef$Instantiate \
         ("/home/tulip/uszok/strand/fw/", "worker", "", "", nullRef, 0)

   /* Create object in capsule */
!  {ObjRef, Res} <- CapRef$Instantiate(nullRef, nullRef,\
                          "Worker", "", "")
   /* Initiate capsule 'worker' */
!  {} <- Res.data[0]$InitWorker(Width, Heigth, Size, Size*i)
  }

  *result = host_nb;
  return SuccessfulInvocation;
 }
```

An invocations of a operation *InitWorker* starts execution of capsules *worker*. They compute (using the function *c_work*) a part of the fractal image until their input data has been exhausted. Than they call the `feedMe` function provided by `StrandFarmer` interface, asking for new tasks. If they receive them they continue calculation. In the opposite case they finish their work and destroy their capsules.

```
int Worker_InitWorker( _attr, Width, Height, Size, StartPoint)
ansa_InterfaceAttr *_attr;
ansa_Integer Width, Height, Size, StartPoint;
 {
  FractalPoints result;

  /* Import reference to StrandFarmer interface */
! {ir_StrandFarmer} <- traderRef$Import("StrandFarmer",  "/", "")

  do{
   /* Compute fragment of fractal image */
   result.length = Size;
   c_work(Width, Height, &(result.length), StartPoint, &(result.data));

   /* Send calculated part of fractal image and ask for new data */
!  {StartPoint} <- ir_Strand_farmer$feedMe(StartPoint, result)
  } while (StartPoint > 0);

  /* Finish work */
  Destroy_Worker_Object(Global_State);
  exit(0);
 }
```

An invocation of the `feedMe` function from the `StrandFarmer` interface results in sending a request to IS, which dispatches it to Strand's *Stub*. New data for the computation of the fractal image, which is a return value is sent back to the capsule *worker*. After receiving it *worker* can continue execution. In the other case it terminates.

5 Conclusions.

In our opinion, the value of the sum of several different distributed software environments can be increased by providing them with constructs enabling cooperation without the loss of the original computational model. The proposed *Interoperability Architecture* addresses this challenge. In this paper we have concentrated on adding object-oriented distributed systems like ANSA (which is already well-known) or Orbix (which is gaining popularity) to IA. A solution presented uses a special module, called throught the paper *Gateway*, which translates and routes requests between the environment and IA. The paper proposes general architecture which enable automatic generation of *Gateway* for an application. The introduction of this module changes in the minimal level the original computational model of the environments.

References

[1] *ANSAware 4.0 — Application Programmer's Manual.* APM Ltd. Cambridge (1992).

[2] *ANSAware 4.0 — System Programmer's Manual.* APM Ltd. Cambridge (1992).

[3] S. Ahuja, N. Carriero, D. Gelernter, V. Krishnaswamy, *Matching Language and Hardware for Parallel Computation in the Linda Machine*, IEEE Transactions on Computers, Aug vol. 37, (1988).

[4] *Draft Common Object Request Broker Architecture Revision 1.1.* OMG Report 91-12-1, OMG Inc., (1991).

[5] I. Foster, S.Taylor, *Strand - New Concepts in Parallel Programming* Prentice Hall (1990).

[6] R. Hayes, S. W. Manweiler, R. D. Schlichting, *A Simple System for Construction Distributed, Mixed–language Programs.* The University of Arizona TR 87–1b, (1987).

[7] F. Nayeri, B. Hurwitz, *Dispatching Experiments in a Distributed System.* GTE Laboratories Incorporated (1993).

[8] *Orbix – A Technical Overview.* IONA Technologies Ltd., (1993).

[9] *The Orbix Architecture.* IONA Technologies Ltd., (1993).

[10] A. S. Tanenebaum, *Modern Operating Systems*, Prentice-Hall (1992).

[11] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, P. Tarr, *Specification Level Interoperability.* Com. of ACM, May vol. 34 (1991).

[12] K. Zieliński, G. Czajkowski, A. Uszok, *Interoperability in Open Distributed Processing.* Technical Report, Institut of Computer Science UMM Cracow (1994).