Interoperability in Open Distributed Processing

K.Zieliński, G.Czajkowski, A.Uszok
UMM Institute of Computer Science
TR 94-1

18.01.94

Abstract

In this paper Interoperability Architecture (IA) for distributed processing is proposed to provide an uniform platform for application developed over a family of languages or software tools. The basic properties of IA and its functional model are described. As a building IA component Interoperability Server (IS) is specified in a form of generic module which may be customized for different languages and software tools. A short example of using implemented IA, parts of which are Strand and ANSA is given at the end of the paper.

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, 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, such as imperative, functional or logic programming.

An object-oriented approach provides a natural framework for distributed systems integration by viewing distributed components as objects that can communicate with each other using messages addressed to well-defined interfaces. These components have their own locally-defined procedures that they call when message are sent to them. Object-orientation addresses the key demands of the system integration such as [12]:

- Emphasis on interfaces allows components in the system to be implemented differently, as long as they adhere to a known interface.
- Components can change independently of others, provided that their interfaces are maintained.

- The message-passing style of classical object systems maps well to the interapplication communication.

The goal of this paper is to present Interoperability Architecture (IA) 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. Our concept grows up from object-oriented approach but our effort is to allow interoperability among diverse distributed systems, not only object-oriented, without creating extra complexity. IA provides functionality addressing the problem of cooperation of various software environments or integration of software components. Thay can be developed as separate programs in any language supported by IA. IA provides mechanisms which make some distributed environments like Linda [3, 9], Strand [8, 17] or PCN [6, 7] open what means that a new application program may cooperate with programs developed earlier, after a minimum amount of changes. It is a meaning of the open system notion as it is understood in this paper.

There are two levels shown in Fig. 1 where an interoperability may be implemented in open distributed processing system:

1. A distributed processing system is built via integration of existing applications. In general these applications may be written in different languages and thus may represent different models of distributed processing.

2. A system is created as an application implemented over integrated software platforms, which is constructed from different software tools and languages and used in a uniform way as a common programming environment.

The level on which the interoperability is provided has an effect on overall system features and is a subject of detailed study in this paper. The proposed IA is oriented on the software platforms integration. It is rather in contrast to the previous works [11] devoted mainly to applications integration and limited rather to different object models cooperation.

The scope of the paper is as follows. In Section 2 influence of the integration level choice on system properties (in terms advantages and disadvantages) is discussed. Next in Section 3 design alternatives of IA are considered. This analisis led to the design assumptions specification and the general functional model of IS. Distributed and centralized implementations of IA are shortly compared. IS is considered as a basic IA component and specifes in a form of generic module that may be customized according to the requirements. In Section 4 a simple program written partly in Strand and partly in ANSA [1, 2] is described as an example of using IA, as proposed in previous section. The paper is ended with conclusions.

2 Influence of the level of interoperability on IA characteristics

The interoperability of systems may be achieved on various levels. It may concern a whole applications integration, software modules which may be separately compiled and statically linked to create one application or cooperation of dynamically linked distributed

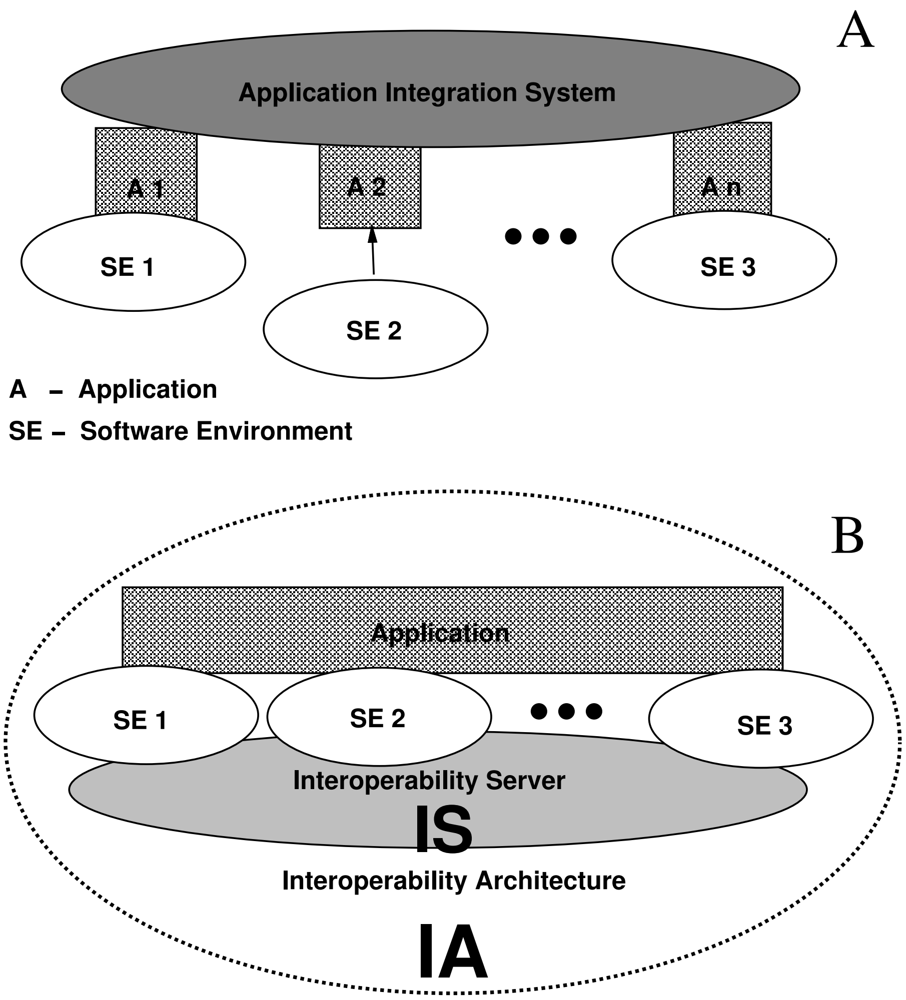

Figure 1: Application versus software environment integration

objects. As it is depicted in Fig. 1A integration may concern a program like for instance an application (see A2) written in SR or a program with an abstract machine for it execution as for Strand and Linda (see A2 and A3). Despite of the programming technique used for the system integration a key point is to understand the difference between applications integration and interoperability on the level of software tools used for their implementation, that provided by the proposed IA shown in Fig 1B. This difference is a motivation of our study and together with a general assumption about object oriented approach determines a basic IA property. To contrast IA with application integration we start from the former characteristic.

2.1 Integration of applications

The integration of applications is currently going to be very important as a main idea of many projects, such as DOE [4] or OLE [13], concentrating around OMG CORBA [5] model . The key point of these efforts is to provide a uniform tool for integration different applications by standardization of interfaces between them and providing mechanisms for an object oriented style of services invocation. In this approach a programmer must define an interface, which is specified in an interface definition language (IDL), and write a server process which implements an access to an existing application. An interface is made publicly accessible and acts as a gateway to the application. This methodology has the following advantages:

- It provides an uniform approach for system integration, built of existing applications.
- It provides a framework for reuse of existing software components, and makes them accessible in a distributed environment.
- The existing and new applications my be integrated to create one system, hiding differences in computational models, data representation and etc., behind interfaces.
- Different implementations of the same service may be used to provide a desired function accessible via unchanged interface. Hence an old implementation may be transparently substituted by a new one.
- Heterogeneous system may be effectively used, and different hardware and software architectures may be exploited.
- A distributed application may be built by composition of sequential ones.

This methodology has also same disadvantages:

- An integrated application is built of rather courser grain modules wrapped by objects communicating via interfaces. It is justified by the fact that the provided mechanism of integration is rather heavy.
- An original computational model of given application can not be expanded transparently across a system, rather capsules that isolate different models are created.
- This approach is rather limited to object oriented distributed processing.

2.2 Integration of programming environments

This approach is rather better fitted to the development of a new distributed applications because it may be designed and implemented having a given computational model in mind. A programmer gets an access to various programming techniques, processing models and may use them for the application development. A process of creating an application can be split into parts dealing separately with problem domains in which using a particular approach (e.g. object-oriented or logic programming) is most suitable. The computational model is retained in languages making up the resulting environment. This approach has a number of advantages:

- It creates platform for finer grain integration, which means that functions and services written in different languages may be used transparently. It is in contrast to integration of whole applications.
- A native computational model of the distributed application may be preserved across a system and enhanced by the components implemented using different software tools, particularly the most suitable for the given function. Application can be tailored exactly as it is desired.
- It provides support for software reuse, because the object wrapping techniques may be used without limitation.
- It solves out in a uniform way a problem of hybrid application implementation that combine different computational models. This leads to the elimination of error prone *ad hoc* designed interfaces between application written in different languages.
- It enables full control over developed application.
- The performance may benefit from using fine-grained parallelism if only component languages support it.
- It provides mechanisms to open a closed software environment that is it makes a cooperation possible via dynamic binding of the programs written in the same or different languages.
- A considerable amount of existing software (in source code) can be changed a little and used, making easy computations that were so far cumbersome and difficult to accomplish, e.g. symbolic processing in C.

The disadvantages of this level integration are as follows:

- It requires initially much more work because the link between different environments should be established. Because it is the crucial part of the system, it must be designed extremely carefully. Special care must be taken so that the link does not interfere negatively with environments which it connects.
- It requires thorough familiarity with all component environments, which is sometimes time consuming to acquire. Necessity to take different approaches while developing applications may become a major problem at creating large programs – a cooperation of experts in different programming paradigms may be necessary.
- It causes multiplication of bugs and exceptions which exist in cooperating environments.
- The flexible and efficient implementation needs advance software technology such as dynamic invocation interface [5], etc.
- A framework for IS generation should be provided to make the system easy expandable.

The specified general properties of integration at this level are in a natural way inherited by IA. The key disadvantages have in fact their source in the most difficult implementation problems of IS.

3 Design analysis of IA

Our paper concerns distributed processing so the main purpose is to integrate different distributed processing environments. We concentrate on Strand, Linda, PCN and ANSA integration. We try to do it as generally as possible, so that new, not existing yet environments can be easily integrated into the set of cooperating entities. There are three things in a given environment which might be affected by our work:

- The way programs are written,
- Compiling,
- Starting an application/environment.

We would like to interfere as little as possible with these items; however, the fact that the language is a component of some bigger structure will not be always invisible to the user. In other words, it is not always possible to achieve transparency in these subjects.

It would be desirable that the way remote, foreign procedure be called in exactly the same way native code procedure is called. However, it can not always be achieved. For example, in languages where data and program are equivalent (e.g. logic programming) it is possible to call any procedure, for example typed in from the console. If such a procedure is not found in a program, run-time system reports an error. However, without tinkering with the system code, it is impossible to have it check if the called procedure is or is not provided by some other environment. In such a case some more involved mechanisms, like perpetual server-processes and incomplete-message communication mechanisms, rather than new process creation, must be used.

The process of compiling may be affected in systems like ANSA, where it is necessary to know the names of all called procedures at compile-time (usually, such languages have C as their syntax base). Automatic generation of such information requires some preprocessing.

When starting an application, some configuration file will have to be supplied, allowing bootstrap in a distributed system. Some other parameters and names of auxiliary processes, helping in communication, may also have to be given.

3.1 Architecture

We can discuss several possible variants of the architecture of the system that interests us. All these variants should have similar elements, because their main functionality is the same – to link together, in the same manner, different programming environments. These functional elements are:

- Data type translation module,

- Service invocation mechanisms,
- Binding and dispatching mechanisms,
- Communication modules,
- Exception and error protocol.

The differences between these variants lie in placement of these elements in IS's modules. We consider following approaches depicted in Fig. 2 and Fig. 3:

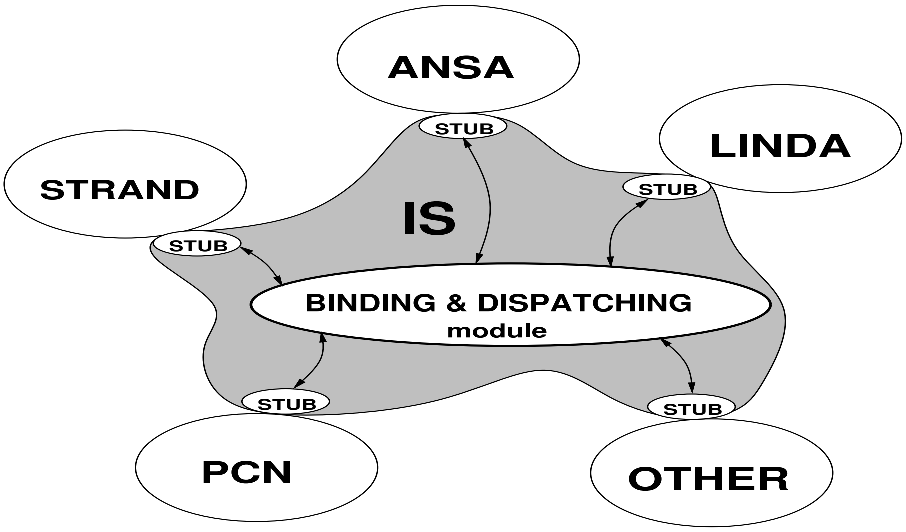

Figure 2: Centralized architecture of IA

- Centralized – there exist one process which serves as a *binder* and *dispatcher* for all cooperating environments and module or process (not exactly one) (*Stub*) for each of these systems which marshals parameters, invokes external procedures, receives responses and maintains connection.
- Distributed – there is no central process which keeps global information of linked environments, every environment has to try on his own to bind to other systems which can provide external services.

When using the centralized approach we gain easiness of binding which can be done on the basis of global information. On the other hand we face all the usual disadvantages of centralized system: single point of failure and performance bottleneck.

Distributed architecture offers more flexibility and possible performance increase. Binding, naming and location become considerable problems, together with lack of accurate global information.

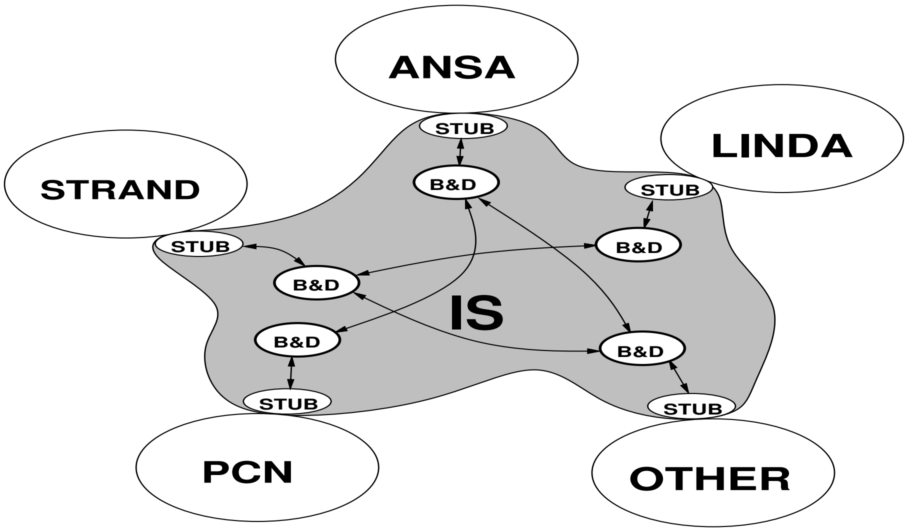

Figure 3: Distributed architecture of IA

3.2 Design alternatives of IS

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. Below we investigate the crucial design aspects of IS.

Communication protocol design

For us the only reasonable choice concerning data transmission protocol was TCP/IP versus UDP [16]. Features favoring UDP were: preserving boundaries between messages and speed. On the other hand, while a message that is delivered using UDP is (almost) never distorted, messages do sometimes get lost or duplicated. Reliable use of UDP requires setting up an acknowledgment/time-out protocol. Although TCP does not possess this disadvantages, we found that for our purposes speed is the most crucial feature (a considerable amount of time is used by type converting and marshaling procedures) and choose UDP. However, working on inclusion some environments (like SCA Linda [10]) into IA, we have encountered some major problems with establishing time-outs, which is necessary when using UDP with acknowledgment protocol, because timer signal cannot be caught by the user program in these systems. Such a problem may be solved by setting several other processes (e. g. ordinary UNIX processes) handling the communication. This solution is difficult and not elegant. We have decided to leave open the possibility to use TCP, but we have based data transmission on UDP wherever possible.

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 Prolog? Or, similarly, in some implementations of Linda, 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.

Concurrency and multithreading in the *stub* implementation

If we are considering systems which are not concurrent or parallel then we can use synchronous remote procedure calls (RPC) when invoking external services. This approach does not require introducing any concurrency to the implementation of *stubs*. But we assume that any programming environment can be a member of IA, probably with internal multithreading. Such systems like ANSA, SR and others which are distributed and parallel but also some logic programming environment like Strand, PCN etc. which evaluate their expressions in concurrent way force us to use asynchronous RPC. But if we construct *stub* for such environment then we usually can use multithreading mechanism provided by this system like for instance in ANSA or in other case we can use interrupt-driven methods to achieve it.

Stub construction

Stub is responsible for all activity related to invoking external procedures. This part of IS usually can be written in the language offered by the environment, actually at least some parts of it have to be written in this language. It results from the fact that

this component of IS should call foreign services in the most transparent manner for the environment for which works. In general these actions are:

- Choosing and take over invocations of external procedures from an environment,
- Translating them and their arguments into universal form,
- Attaching unique identifier to this invocation,
- Putting record with this identifier into the list of calls waiting for an answer,
- Sending such prepared form of invocation to the binding service,

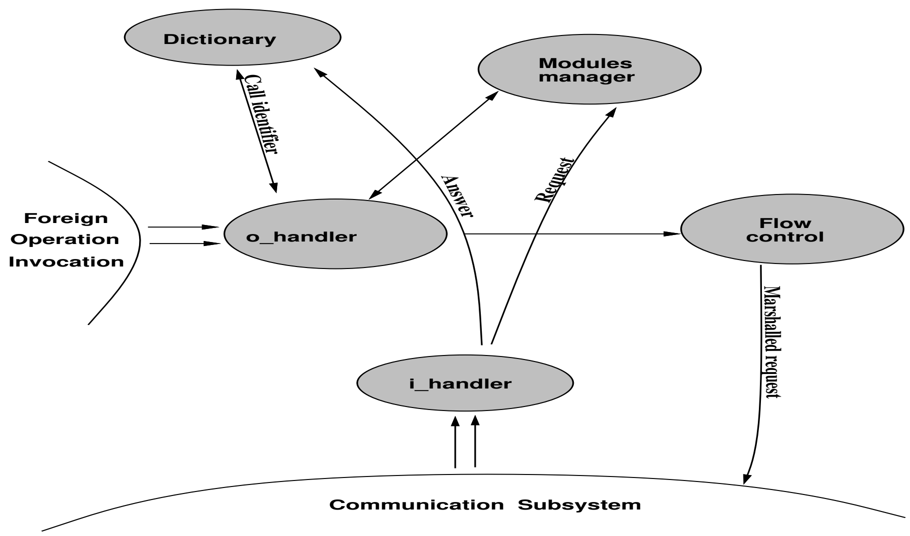

Figure 4: General structure of *stub*

- Receiving answers and errors from others environments.

Fig. 4 shows universal structure and elements of *stub*. Particular components perform the following operations:

- *O_handler* – takes invocation of foreign operation,
- *I_handler* – receives messages from other environments,
- *Flow_control* – manages the sending of messages to the outer world,
- *Dictionary* – maintains the database of calls waiting for a response.
- *Modules_manager* – keeps information of exported services and modules which support them.

For some environments, like Strand or PCN, the *stub* can be written generally and is not changed afterwards. For some others, like ANSA or CORBA, the stub must be prepared for each application separately, based on the information on service interfaces used by the application. Such stubs can be generated automatically.

Binding service

It is one of the most difficult problems which should be solved during design of IS. If linked environments are to cooperate calling procedures of each other, they should be able first to find and establish communication with a required system and then invoke appropriate procedure. As we have noticed earlier there is a significant difference in possible solutions to this problem between the centralized and distributed architectures of IA. We consider:

- Binding in centralized architecture,
  For this architecture there is just one module, implemented as a separate process,

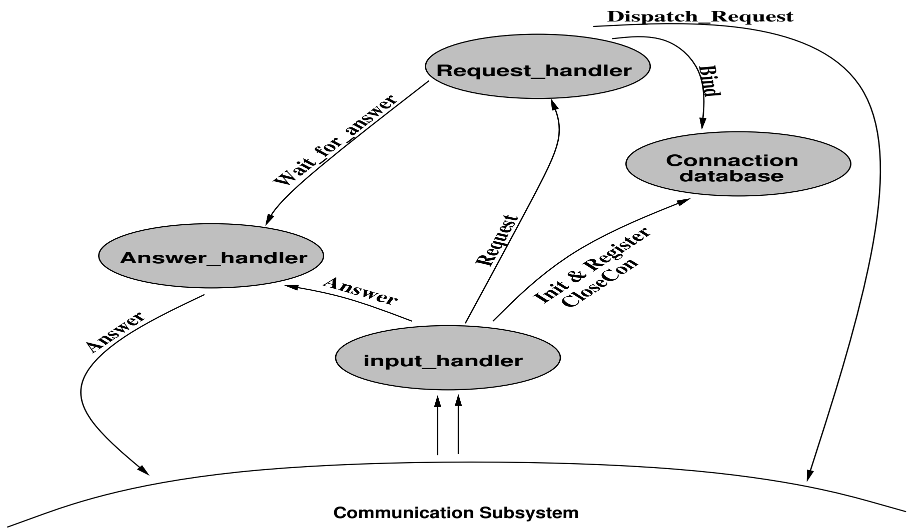

Figure 5: General structure of the binding and dispatching module.

  which serves as a *binder* and a *dispatcher* (*B&D_module*) for cooperating systems (see Fig. 5). First each environment, which wants to interact with the others, should register itself in this component. Also if it likes to offer some kind of services to its partners it should inform the *B&D_module* about it. Afterwards it can invoke external procedures asking a binding process to find an appropriate environment according to the interface's or system's name and to dispatch procedure call to the chosen one. Invocation result or exception code will be returned to the caller. If system has registered any services interfaces in the *B&D_module* it can expect that some other systems will invoke its procedures as well. Result of such call should be simply sent to the *B&D_module* which knows what to do with it. When the system does not want to cooperate any longer it sends *deregister* information to the *B&D_module*. This register/deregister scheme allows to avoid a lot of errors in the activity of IA. Of course in some situations the environment may be not able to deregister itself for some reason. That is why *keep-alive* protocol is kept between *B&D_module* and systems. If a system does not respond for a certain period of time it is deregistered automatically.

- Binding in distributed architecture,
  Solving the binding and dispatching problem in this architecture is more difficult than in previous situation. If interacting systems are to find themselves some kind of location service has to be implemented what can be difficult and clumsy. That is the reason why we propose the use of existing distributed environments, like ANSA or CORBA which can serve as a binding and dispatching medium for IA

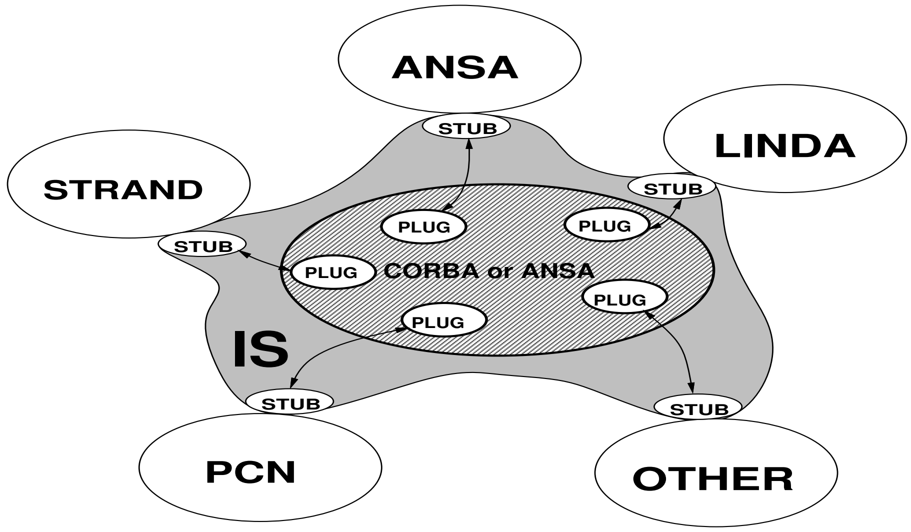

Figure 6: ANSA (CORBA) as a *medium* in distributed architecture of IA

(see Fig. 6). They will be seen as a *glue* which keeps together elements of IA. To achieve this we should design component which will represent given system in the medium environment. We called this module *plug*, the activity of which is limited to simple information passing between the *stub* and medium. The medium environment is responsible for binding and dispatching of request calls to the *plug* of the system which offers the required service. A *plug* construction can be universal.

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* [18] 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.

3.3 Chosen design principles

To sum up, we have decided to base communication protocol on UDP, supply only basic data types, that is integers, real numbers, strings and their sequences, make the foreign procedure call mechanism as similar as possible to the model of a calling environment, provide error protocol together with connection control.

We do not think that any of the discussed architectures of IA is better than the other one; both the centralized and distributed architectures have their advantages and disadvantages. At this moment we have implemented centralized variant but we are working on decentralized IA very intensively, having it as the target solution. We are also working on expanding the set of integrated systems — now we have ANSA, Strand, PCN and CORBA. We plan to add Linda soon; also, SR is considered.

4 Examples

As said above, the system containing several environments was implemented. Although, in our opinion, interesting, implementation details are not covered in this section because we are more interested in showing how the environments can cooperate. Here we will show an example program in which Strand process asks IA to compute an Ackermann number. A server providing that kind of service is a running ANSA program. In turn, ANSA, like Strand, is a part of IA. The example chosen is extremely simple – we rather try to concentrate on how the foreign call is accomplished.

4.1 Startup

First of all, before any interaction between systems making up IA takes place, IS must be started, what includes: *B&D_module* and *stubs* for ANSA and Strand. An ANSA *stub* is a separate UNIX process, written in ANSA and working in its environment. It must be started separately, while a *stub* for Strand is implemented by means of two caught signal handlers, and is started as a call to Strand process *start_work*. Of course, before any Strand process may request computing of an Ackermann number, this service must be registered and an appropriate ANSA capsule must be running.

4.2 A Strand program

In order to make Strand part of IA, user must specify a bootstrap file (details will not be given here) and call *start_work(IStream, OStream, Status)*. All requests to the IA servers are put into *OStream*, so from the Strand user point of view, communication with entities in IA is done using a well known logic programming technique called *incomplete* *messages*; *IStream* contains information from the IA processes, what allows IA processes to maintain connection with Strand processes, creating thus Strand servers that possess a state. *Status* informs of errors.

An example call to a foreign Ackermann routine could be written like this:

```
compute(P1, P2, Res) :-
  start_work(_, OStream, _),
  OStream := [ { ackermann, P1, P2, Res } | _ ].
```

After putting a tuple { *ackermann, P1, P2, Res* } to the *OStream*, it is handled by auxiliary Strand processes. First, it is intercepted by an *o_handler* process, which is responsible for dealing with all outgoing calls. An unique number *UN* , identifying the call, is assigned to it and a tuple { *UN, Res* } is stored in a *dictionary* process, waiting for a call to return. Then, a tuple containing all arguments but the last one of the original tuple and with the addition of *UN*, that is { *UN, ackermann, P1, P2* } , let us call it *X*, is created. When *X* is fully ground, that is, there is no variable in it, *X* is transferred to the Strand process called *flow_control*. When all messages previously sent are acknowledged, *flow_control* calls a C routine that marshals *X* and sends it to IS. All the actions that happen from the point of delivery of marshalled call name and its parameters until a result is returned to Strand are described in the next section. When the call returns to Strand, the unique call identifier and the result are extracted from the answer message, and a tuple { *UN1, Res1* } is send to *dictionary*. Using a dictionary entry labeled with *UN1*, a *Res* variable is assigned *Res1*.

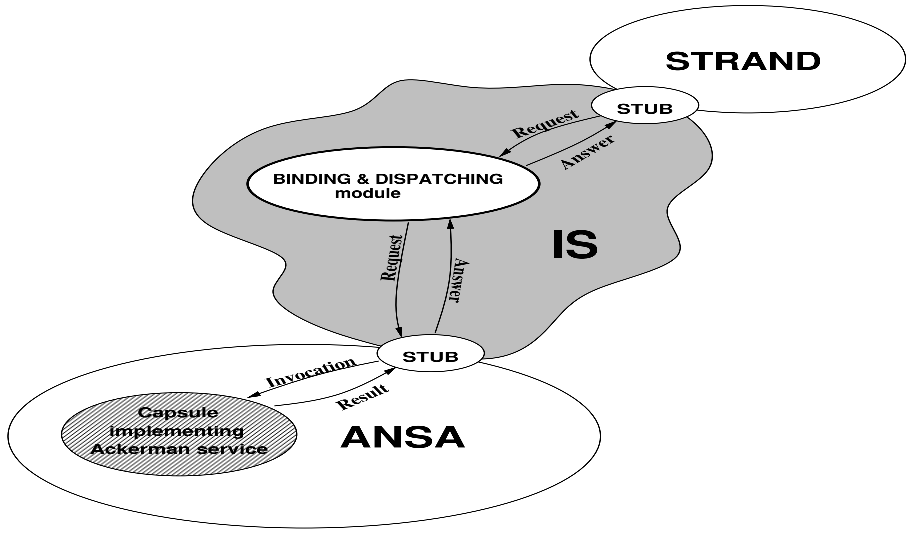

Figure 7: Invocation pattern.

4.3 *B&D_module* activities

When a request packet is obtained by *B&D_module*, it is routed to the appropriate IA environment. That environment is located using information obtained from environments (registration). Similarly, a message containing a marshalled answer is transferred from the server to the caller by means of *B&D_module*. In our case ANSA environment has registered procedure which calculates Ackermann numbers. When *B&D_module* receives message from Strand containing request to compute such a number it will find ANSA as a provider of the service. According to the information about the location of ANSA's *stub*, part of *B&D_module* responsible for dispatching forwards a Strand's request to this *stub*.

4.4 ANSA program

After all the activities described above ANSA *stub* which registered the *ackermann* service in *B&D_module* receives a message to perform requested operation. ANSA's language is based on C and its *stub* should be separately generated for each sets of offered services. It is possible that there are more active *stubs* in ANSA system which export different procedures. The part of each *stub* is generated by a prepocessor *dispatcher* procedure which chooses right ANSA method according to a demarshalled operation's name. Below we show a simple instance of such *dispatcher* procedure.

```
int Invoke_Call(_attr)
ansa_InterfaceAttr *_attr;
 {
   ....

   /* Choose a method */
   if(!strcmp(argl[0]->values.string.string_val, "ackermann")){
    /* Call ANSA operation */
!    {argl[0]->values.int_val} <- ir_Ackermann$ackermann(argl[1]->values.int_val)

    /* Marshal result */
    buflen = datatype_encode(argl[0], rnumber, (char)0, &buf);

    /* Send answer to B&C_module */
    c_send(buf, buflen, mn, 0);
   }
   else
    (void)fprintf(stderr, "BCmodule error: Unknown operation name\n");

  return SuccessfulInvocation;
 }
```

The called ANSA method is supported by a capsule running in ANSA environment. It can be located on any node of the network. ANSA is responsible for finding this capsule and invoking appropriate operation. Result returned by this call is then marshalled and send back to *B&D_module*.

The path traversed by a Strand request to the ANSA service provider and the route of the reply message from ANSA back to Strand is shown in the simplified form on Fig. 7.

5 Conclusions

The proposed IA built around a family of distributed languages provides an uniform platform for hybrid distributed processing. Various computational models may cooperate effectively within one application implemented in most suitable environment. IA is constructed within object-oriented framework, what is observed in functionality represented by IS, but the interoperability model it provides may be applied to any software environment. Hence for instance a logic programming application may cooperate with a computational modules implemented as objects written in ANSA.

The most difficult implementation problems of IS are: binding in distributed environment, data translation between various languages, providing of services and invocation transparency. The first problem may be solved out using the rather general mechanisms, provided already for instance by CORBA or ANSA. Another two issues unfortunately need careful investigation of the target language semantics and syntax. So it is really difficult to propose a general solution.

The identified functional components of IS and investigated implementation alternatives provide a general framework for it construction. This framework may be used for extension of IA with new systems. As an example of using IA a simple program in which Strand and ANSA interact was presented. However IS was constructed also for PCN, Linda and ORBIX [14, 15] what proved that the assumed model of interoperability IA provides is general. So far IS framework has been described informally. The next step would be a construction of a software tool for IA extension with a given language by automatic IS code generation.

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] N. Carriero, D. Gelernter, *How to write parallel programs: A guide to the perplexed.* ACM Computer Surveys, September (1989).

[4] J.D. Clark, *Window Programmer's Guide to OLE/DDE.* Prentice-Hall (1992).

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

[6] I. Foster, S. Tuecke *Parallel Programming with PCN* Argonne National Laboratories, 1992.

[7] I. Foster, R. Olson, S. Tuecke *Productive Parallel Programming: The PCN Approach* Scientific Programming, Vol.1 (1992).

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

[9] D. Gelernter, *Parallelism to the People, or, Experience with Programminig in Linda.*

[10] *SCA Linda ver 2.4.6 – Reference Manual* (1991).

[11] 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).

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

[13]

[14] *Orbix – A technical Overview.* IONA Technologies Ltd., (1993).

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

[16] W. R. Stevens, *UNIX Network Programming.* Prentice Hall (1990).

[17] *Strand'88 User Manual Buckingham Release.* Strand Software Technologies Limited, (1990).

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

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