Coordination through Interoperability in Distributed Processing *

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

January 1994

Abstract

In this paper Interoperability Architecture (IA) for distributed processing is proposed to provide an uniform platform for application developed and coordination over a family of languages and 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. Computations may consist of grains of different sizes, depending on the set of cooperating systems. No new language for execution control is introduced. If there is a need for an explicit coordinator, some program written in any system added to IA may serve that purpose.

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 and coordinate their cooperation, creating an integrated distributed computing system.

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 integration of various environments and coordination of applications' execution written in them.

The object of coordination of computations can be understood in many way. It may concern different levels of abstractions and granularity of concurrency. We examine the

*This work was sponsored by Polish Research Council (KBN). under grant no.: 8 S503 015 06.

means by which differnet software environments, ranging from object-oriented to logic-programming, can cooperate. By extending a given system with basic functionalities concerning communication, registering and calling services and error handling, the environments can be both clients and servers in a set of cooperating software entities. An application can be developed as separate programs or modules in any language supported by IA. It provides mechanisms which make some distributed environments: ANSA [1], Orbix [11], Linda [3], Strand [7], PCN [6] or SR [2] open for cooperation with others.

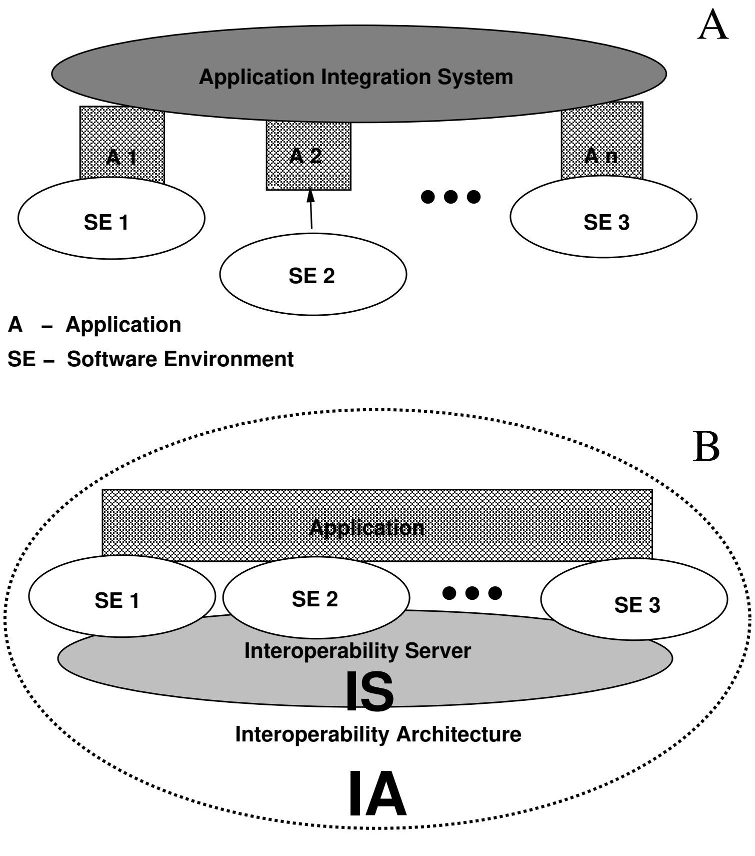

Figure 1: Application versus software environment integration

There are two levels shown in Fig. 1 where an interoperability may be implemented in 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 processing. This approach usually introduces a new coordination platform.

2. An application is implemented over a hybrid system consisting of integrated software platforms. The system is constructed from different software tools and languages and used as a common programming environment. This methodology provides coordination transparency.

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.

The scope of the paper is as follows. In Section 2 the 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 analysis 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 program written partly in Strand and partly in ANSA is described as an example of using IA and idea of coordination transparency, as proposed in previous sections. 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 objects. As it is depicted in Fig. 1A integration may concern a program like for instance an executable application (A_2) written in SR or a program with the environment's runtime support machine for it execution as for Strand and Linda (A_1 and A_n). Regardless of the programming techniques 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, as 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 describing the features of the former.

2.1 Integration of applications

The integration of applications is currently going to be very important as a main idea of many projects 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 provides an uniform approach for system integration and coordination, built of existing applications. It offers a framework for reuse of existing software components, and makes them accessible in a distributed environment. The existing and new applications may 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.

An integrated application is built of coarse 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. The coordination mechanisms are provided and enforced by an integration platform and may be very different from those used in the original versions of components.

2.2 Integration of programming environments

Integration of programming environments is better fitted to the development of a new distributed applications because they may be designed and implemented having a choice of computational and coordination models. 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 functional) is most suitable. The computational model is retained in languages making up the resulting environment.

This approach creates platform for finer grain integration, which means that functions and services written in different languages may be used transparently. The performance may benefit from using fine-grained parallelism if only component languages support it. A original 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 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.

On the other hand this solution requires initially much more implementation 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 also be taken so that the link does not interfere negatively with environments which it connects. A framework for IS generation should be provided to make the system easy to expandable. It requires thorough familiarity with all component environments, which is usually 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.

3 Design analysis of IA

Our paper concerns distributed processing so the main purpose is to integrate different distributed processing environments. We try to do it as generally as possible, so that new, not existing yet environments or the ones with which we are not familiar yet can be easily integrated into the set of cooperating entities. The curent implementation includes: ANSA, Orbix, Strand, Linda, PCN and SR. 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 the environment. In such a case some more involved mechanisms, like perpetual server-processes and incomplete-message communication [7, 4].

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 two possible variants of the IA architecture: centralized and distributed. Both of them have similar elements, because their 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 the place of these elements in IS's modules:

- Centralized (Fig. 2) – 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.

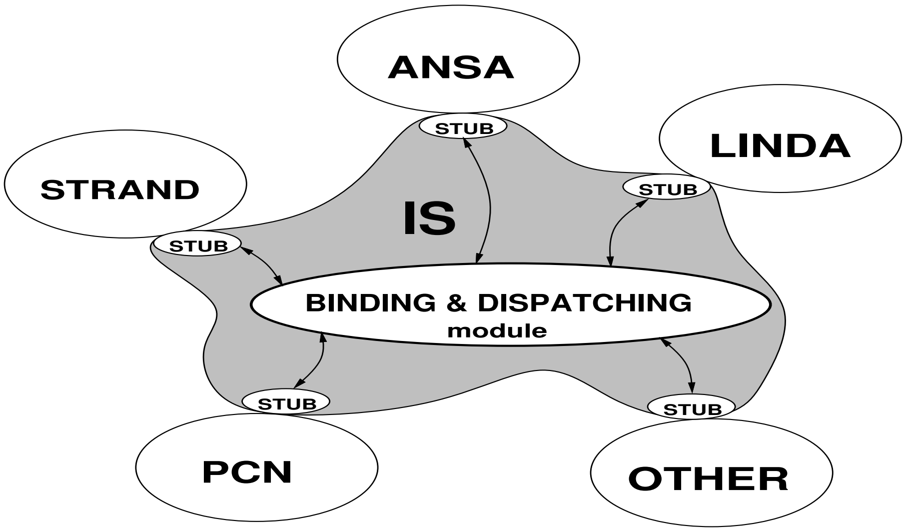

Figure 2: Centralized architecture of IA

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.

3.2 Coordination in IA

We do not introduce any new language, which we consider to be an advantage of our approach. IA preservs original computational models of integrated software environments and provides coordinations transparency. From the given environment perspective its coordination model is expanded over the services provided by foreign system.

We do not designate one of the environments participating in IA as a coordinator of computations. According to the problem at hand, the program written in any of the environments grouped in IA may coordinate others. If there is a need for an explicit coordinator (i.e. in computations concerned with load-balancing or physical simulations with barrier) one may choose the appropriate language belonging to IA as a coordinator. The choice may also be guided by some other motivation – e.g. software reuse. If there is no such need, there may be no explicit coordinator – the processes exchange messages and requests and synchronise without any supervisor. Chosing either option (i.e. with or without a coordinator) is up to the programmer and depends entirely on an application and logical functions of some process (processes) with respect to coordinating activities of others.

Internally, every environment preserves its own coordination mechanisms. If a process becomes a coordinator of computations, the coordination mechanisms from its environment are reflected in the global coordination. Thus, we have a two-level coordination: the global one, inherited from the system in which the coordinator is written and the local one, which depends on the environment and affects computations which are carried out entirely within that environment, i.e. without communication with IA.

The size of individual computational grains are not limited a priori neither from below or above – in our system, one can call both services written in Orbix[1], where there is no internal concurrency and the smallest grain is a operating system process and services defined in Strand or PCN where processes are extremely lightweight.

3.3 Design alternatives of IS

As mentioned earlier, IS is a specialized component that integrates systems grouped in IA. 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.

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.

Stub construction

Stub is responsible for all activity related to invoking external procedures and handling incoming requests, on behalf of a given environment. This part of IS can usually 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:

- Initiation of work in IA,

[1]As we have been informed, the newest version of Orbix will provide the possibility to use the C lightweight threads library, if available on the given operating system.

- 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.

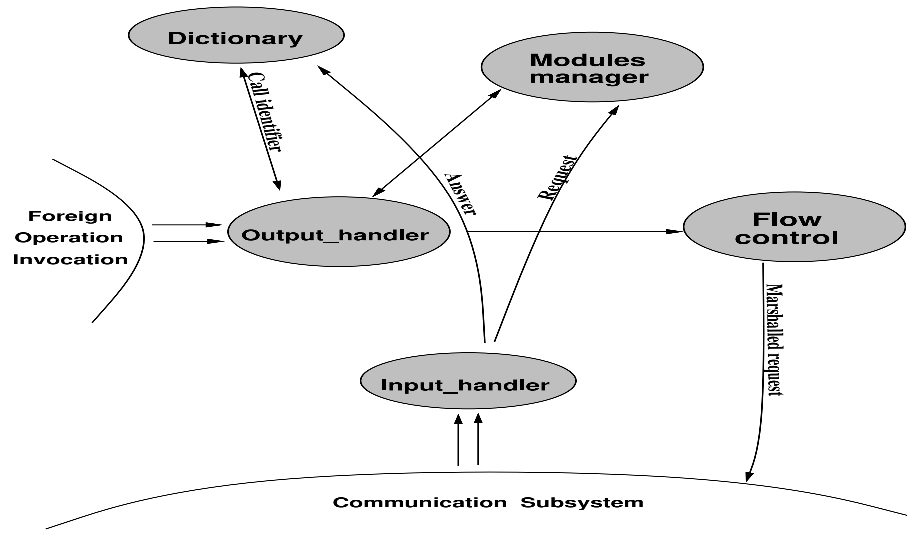

Figure 3: General structure of *stub*

Fig. 3 shows universal structure and elements of *stub*. Particular components perform the following operations:

- *Output_handler* – takes invocation of foreign operation,
- *Input_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.

In general, there exist two different ways in which a *stub* can be constructed, which depend on the features of the system added to IA. For declarative environments the *stub* can be written generally and is not changed afterwards [4]. For imperative, object-oriented, like ANSA or CORBA-compliant systems, *stub* must be prepared for each application separately [13] (except a usage of Dynamic Invocation Interface [5]), based on the information on service interfaces used by the application. Such *stub*s can be generated automatically when required.

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.

Binding in centralized architecture

For this architecture there is just one module, implemented as a separate process, which

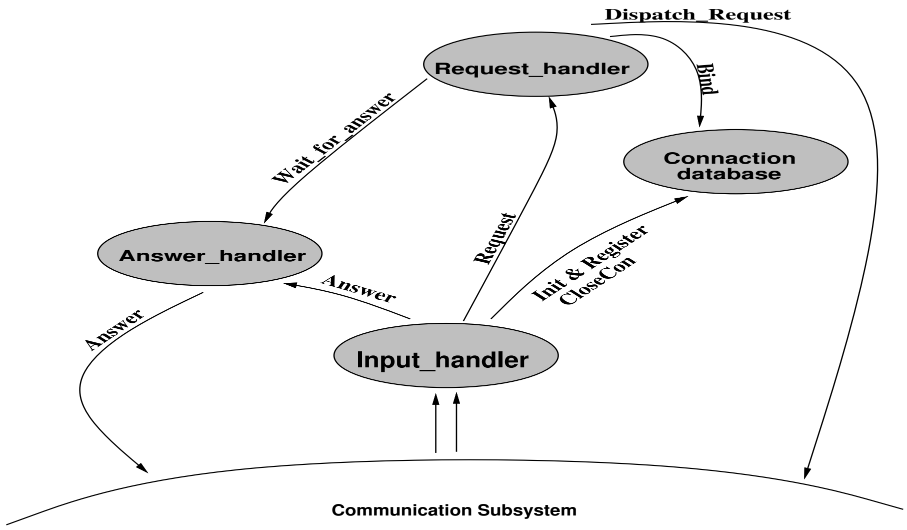

Figure 4: General structure of the binding and dispatching module.

serves as a *binder* and a *dispatcher* (*B&D_module*) for cooperating systems (see Fig. 4). First each environment, which wants to interact with the others, should register itself in this component. Also if it intends 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. In some situations the environment may crash and not be able to deregister its services before that event. 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

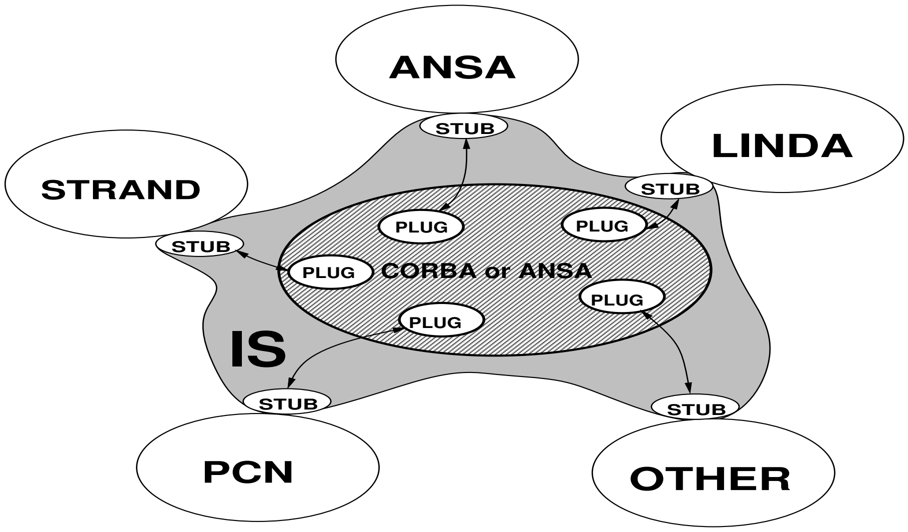

Figure 5: ANSA (CORBA) as a *medium* in distributed architecture of IA

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 (see Fig. 5). 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* [12] 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.

3.4 Chosen design principles

To sum up, we have decided to 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, having it as the target solution. We are also working on expanding the set of integrated systems.

4 Example

As said above, the system integrating several environments was built; the implementation details can be found elsewhere [4, 13]. In this section we show how the environments can cooperate and computations can be coordinated.

This example uses a model of processors farm for the calculation of a fractal bitmap. The processor farm consists of a distinguished *farmer* process, which coordinates the activities of a set of (usually identical) *worker* processes [15]. 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. 6).

4.1 Interaction of the components in the example application

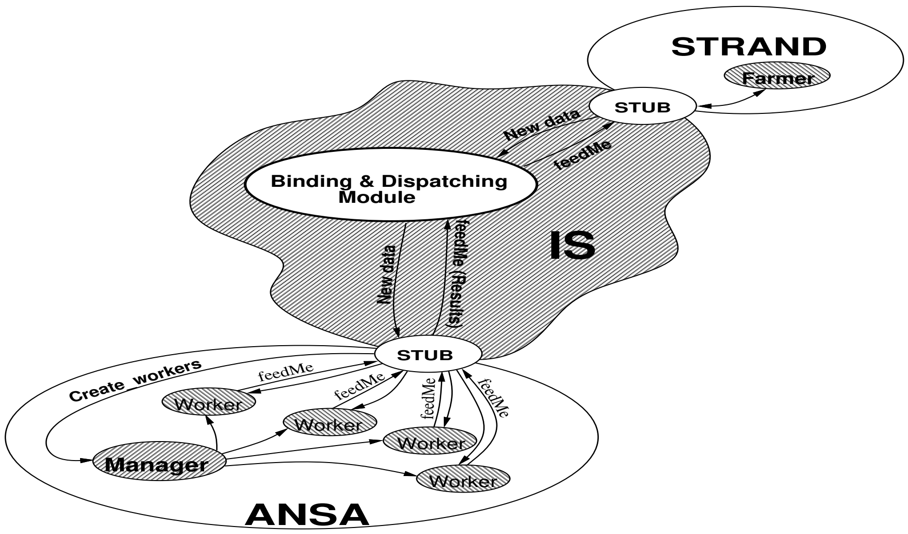

Figure 6: Coordination of IA's elements during computation of Mandelbrot fractal image

A *farmer* module is written in Strand language and is executed in this environment. It is responsible for coordination of computation of fractal bitmap, especially it organizes work of *worker*s executing 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* is to cooperate with ANSA than IS module which is responsible for binding and dispatching should be run beforehand. Services offered by ANSA (and also by Strand) and necessary for proper execution of this computation should be registered in this module. The output stream of coordination data 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. Then *worker*s start computation of first tasks and invoke operation `feedMe` after having completed them.

Description of implementation of these operations which ensures coordination of application parts executing in Strand and ANSA environments is presented in next points of this section.

4.2 Cordinating computations - Strand program

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 unknown 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 an via it with *worker*s in ANSA.

4.3 Actual computations - ANSA program

The activity of ANSA environment begins when its *stub* receives a request `StartWorkers` from Strand for the capsule `Manager`. The invocation of this operation from the `Manager`'s interface 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 Worker_InitWorker( _attr, Width, Height, Size, StartPoint)
ansa_InterfaceAttr *_attr;
ansa_Integer Width, Height, Size, StartPoint;
 {
  FractalPoints result;

  /* Import reference to Farmer interface */
```

```
! {ir_Farmer} <- traderRef$Import("Farmer",  "/", "")

  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_Farmer$feedMe(StartPoint, result)
  } while (StartPoint > 0);

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

An invocations of a operation *InitWorker* by `Manager` 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. Then they call the `feedMe` function provided by `Farmer` interface, asking for new tasks. It results in sending a request to IS, which dispatches it to Strand's *stub*. *Farmer* executing in Strand will provide new data for the computation, which are sent back to the capsule *worker*. If they receive them they continue calculation. In the opposite case they finish their work and destroy their capsules.

5 Conclusions

The proposed IA built around a family of distributed languages provides an uniform platform for hybrid distributed processing. Various computational and coordination 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. The other two issues unfortunately need careful investigation of the target language semantics and syntax. As a consquence, 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 acoordinates ANSA computations. However IS was constructed also for Orbix, PCN, Linda and SR 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] G.R. Andrews, *The distributed programming language SR – mechanisms, design and implementation.* Software–Practice and Experience, Aug vol. 18, (1982).

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

[4] G. Czajkowski, A. Uszok, K. Zieliński, *Distributed Declarative Systems as Parts of Cooperating Software Environments.* Technical Report UMM Cracow. March 1994.

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

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

[9] 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).

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

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

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

[13] A. Uszok, G. Czajkowski, K. Zieliński, *Interoperability Gateway Construction for Object-Oriented Distributed Systems.* Accepted for Nordic Workshop NWPER'94 Lund. March 1994.

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

[15] K. Zieliński, M. Gajęcki, G. Czajkowski, *Parallel Programing System for LAN Distributing Computing*, Proceedings of 14-th IEEE International Conference on Distributed Computing Systems.