Interprocess data transfer in ATLAS, a platform for distributed applications

M. Fairén and À. Vinacua
Department of Software
Institute of Robotics and Industrial Informatics
U.P.C.
Diagonal 647, 8ena planta
E08028 Barcelona, Spain
{mfairen,alvar}@turing.upc.es

Abstract

The ATLAS platform strives to make several useful but technically involved mechanisms available to the programmer building applications over it with the least possible effort. These mechanisms include network distribution of cooperating processes, a powerful macro language, a journaling system and fault tolerance in the presence of network failures or node crashes. In this paper we discuss the techniques used in ATLAS to implement data transfer over a network between different machines with the least hassle to the programmer.

Keywords: Network data transfer, distributed applications, XDR

1 Introduction

ATLAS is a software development platform designed and implemented in our Department with a twofold objective:

- To transparently offer a collection of services to applications developed over it and
- To facilitate the integration and reuse of different components developed at the Department.

Among the services provided under the first of these is included the possibility of breaking up the application into modules that can be distributed between nodes of a LAN. This gives rise to many different problems that need to be addressed[1], and the emphasis in ATLAS's design is to relief the programmer

from most of them. We have made great efforts to make the use of the platform as effortless and transparent as possible.

In ATLAS each module is an independent process running on any of the nodes (that provide it), and therefore the transfer of data between different architectures needs to be addressed.

Further, the data needs to be —at least very often— accessible to ATLAS itself, which includes a programming language to define user-machine dialogues or otherwise interconnect the different modules.

The problem of actually transfering the data robustly has long since been solved. Indeed we just rely on XDR[2] for that purpose. What we discuss here are the mechanisms adopted to attain these data sharing with the maximum of transparency for the user, who needs not be aware of XDR, and indeed almost needs not be aware of our interchange method.

This paper is structured as follows: section 2 describes briefly the design of ATLAS, to set the framework for the ensuing discussion. Section 3 introduces our data structures (called `Variables`) used to wrap user data in each process. These offer access methods used by the interpreter of the command language, and also encapsulate methods to encode and decode XDR streams transparently. Section 4 discusses further the extent to which all this mechanism is invisible to the programmer of an ATLAS application, and section 5 gives some closing conclusions.

2 The ATLAS architecture

The ATLAS architecture is represented in figure 1, where the ovals denote processes and the arrows represent communications between them. It is a centralized architecture where the process `distr` acts as the master process and is the center of each ATLAS application. This architecture allows an intelligent distribution to be managed, i.e. the `distr` process decides the processes distribution dynamically depending on availability, load and aptitude of each host in the network to run each application process. The communications between processes have been implemented with sockets and using the ACE_wrappers library [3] to build the communications drivers.

Figure 1 shows a typical ATLAS application. The processes depicted with a thick line represent the main components of ATLAS. All the others are regarded equally by the system, and they don't need to know about each other.

Of these three main processes, the most crucial one is the master process `distr`. This is the process that the user starts up to invoke the application. It acts as a communications center for the duration of the execution, and provides some essential services to all other modules. It is also responsible of the *journaling* mechanism, the *fault-tolerance* of the system and the central mechanism to assign an input datum, provided by an input system, to the corresponding request, normally issued by another process.

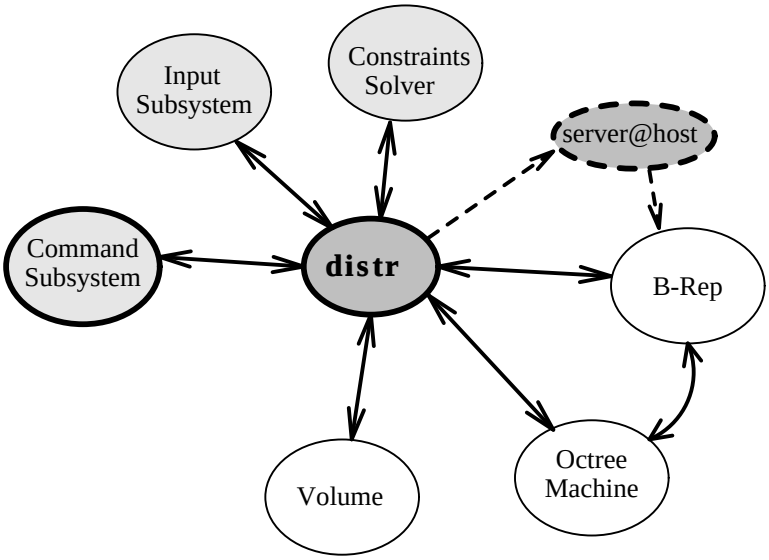

Figure 1: A sample execution of an ATLAS application.

The process `server` is a daemon that runs on all hosts configured to run ATLAS applications in the network. A user can select a specific list of hosts via variables in his environment, or else ATLAS attempts to use all resources configured by the administrator (depending on their load). Each time a new process needs to be loaded and connected to the rest of the application, `distr` connects to the `server` on the chosen target machine and requests that such a process be started for him. `server` then forks a copy of itself, makes the appropriate verifications, loads the adequate environment and execs the desired process. Figure 1 only shows one such `server` for legibility, although one such server will be running on every node available to ATLAS.

The third of the ATLAS main processes is the `command subsystem` which guides the application behavior by interpreting programs and instructions written in a language (ATL) designed for ATLAS and described in [4]. ATL is a powerful control language which allows the developer to describe his application and aspects of its user interface and also allows the final user to introduce his own macros. ATL is a modular language where a module is a file written in this language which can be:

- a description of the interface of the corresponding ATLAS process, including also the definition of commands adequate to that process,
- a definition of some commands useful for the application, but without being directly related with any ATLAS process definition; e.g. commands defining the interaction among several processes.

Atlas uses a remote procedure calling paradigm, where calls are controlled by the command subsystem who executes commands that can invoque extern routines (calls to another process being executed in the application).

An extern routine call usually will have input parameters, and it can also receive input/output parameters and return results. In all of these cases, except when the routine has neither parameters nor result, it is necessary to pass data from one process to another, therefore data must travel through the network and must be readable by different architectures with the same meaning. To solve this problem Atlas communications use the standard XDR representation to exchange data over the network. The parameter passing convention used in Atlas is a "copy-in copy-out" convention, as opposed to applications based on CORBA (see [5]), for example. In this way, objects are managed by the user applications directly, and data is shared only between applications that agree on their type. We have found this scheme to better support the applications we intend to develop, and to favor the portability of modules from one application to another, as they are more loosely coupled. Although type agreement is required, the modules become totally encapsulated and independent. The only mechanism where a "sort-of-pass-by-reference" paradigm is used is the global identifiers mechanism (defined in [6]) where the global identifiers acts as a kind of reference to the concrete datum.

3 Wrapper structure for data and types

In order to make the ATL language more flexible and allow the user to manage structured type variables in his module commands, ATL language accepts the definition of a restricted set of types. This set of types includes the ATL basic types (integer, real, boolean and string) and any construction over them using any order of records or arrays; i.e. a correct type can be: (1) a basic type, (2) a record with a finite number of fields each one of a correct type, or (3) an array with a finite number of components of a correct type. Figure 2 shows an example with some correct type definitions in ATL language.

As is usual in imperative languages with this constructs, the ATL compiler can operate upon components of these variables as if they themselves were variables (of appropriate types), making it possible to use them as in the example shown in figure 3.

The wrapper structure designed in Atlas for data and types is based on the management needed for these kind of data. This management can be divided in two important requirements: (1) making the command subsystem able to access component values of these sort of variables and (2) encapsulating these data robustly allowing it to travel from one process to another.

The Atlas variables are represented internally by a tree structure guided by a compact type definition. This compact definition comes from the ATL type definition and is initialized by the ATL compiler, who is aware of the complete

```
#deftype point STRUCT
                 x -> real;
                 y -> real;
                 z -> real;
               ENDSTRUCT
#deftype face VECTOR [3] OF STRUCT
                              p1 -> point;
                              p2 -> point;
                              ident -> integer;
                            ENDSTRUCT
#deftype pyramid STRUCT
                   name -> string;
                   base -> face;
                   sides -> VECTOR [3] OF face;
                 ENDSTRUCT
```

Figure 2: Example of some correct type definitions in ATL.

```
point p1,p2,p3;
   ...
pyramid pyram1,pyram2;
face base;
base = build_face(p1,p2,p3);
build_pyramid(base,GETDATA("Input fourth point"),"name1",pyram1);
build_pyramid(pyram1.sides[1],
              build_point(base[2].p2.x+100,base[2].p2.y+50,base[2].p2.z),
              "name2",pyram2);
   ...
```

Figure 3: What can be done with variables in ATL.

type composition. As an example the compact type definition for the `pyramid` type in figure 2 would be the following string:

```
"S(name string,
   base V[3]S(p1 S(x real,y real,z real),
              p2 S(x real,y real,z real),
              ident integer),
   sides V[3]V[3]S(p1 S(x real,y real,z real),
                   p2 S(x real,y real,z real),
                   ident integer))"
```

The main interface of C++ class for types is shown in figure 4. This class contains the name of the type (used by the ATL compiler) and its compact definition describing all components of the type. The methods shown in the figure are those needed to manage the access to type components, and some of them are only used for record types because their elements are accessed by name (field names).

The C++ class representing a variable is shown in figure 5. This class encloses a very powerful management of variables

- allows ATLAS to be undisturbed by the variable type (ATLAS sees them as `Variables` and does not care about their internal type),

```
class Type {
    String tipus,deftipus;
  public:
      ...
    int Components ();          // Returns the number of components.
    int Accedir (char *cami);   // Returns the component index from
                                // its name (only for structures).
    char Codi ();               // Returns a code showing the node
                                // type. It is useful when we need
                                // to make explicit castings.
    Type TipComp (int index);   // Returns the component type from
                                // its index.
    String NomCamp (int index); // Returns the component name from
                                // its index (only for structures).
      ...
};
```

Figure 4: Interface for class `Type`.

```
class Variable {
    String nom;
    Type tipus;
    node *arbre;
    XDR reprxdr;
    char *mem;
    int posdada,xdrlong;
  public:
    Variable () {}
    Variable (String t, String n);
    Variable (Type t, String n);
    Variable (const Variable & v);
    Variable (char *m, int lng);
    void crea_arbre ();
    int arbre_to_xdr (FILE *f);   // translation to XDR, diretly
                                  // to the chanel
    int xdr_to_arbre ();          // translation from XDR
      ...
};
```

Figure 5: Interface for `Variable` class.

- contains both the variable tree representation and the standard XDR representation of the variable (where one is computed from the other only when necessary),
- offers methods to translate automatically from one to the other representation. These translations are possible because the variable is always aware of its own type definition.

The tree representation included in the `Variable` class is a pointer to the root node of the tree. Figure 6 shows the interface for the abstract class for a node of the variable tree. This class contains a `Type` which is the type of the node and a number of references to control how many copies of this node are in use. The most important methods in this class are: `accedir` which access a component of the variable from the corresponding index (only useful for records and arrays), and `fromto_xdr`, a pure virtual method that translates from XDR

representation to the tree representation and vice-versa (compulsory for any derived class).

```
class node {
  protected:
    Type tip;
    int referencies;
  public:
    node () {}
    node (Type tipus) : tip(tipus) { referencies = 1; }
    virtual node *accedir (int index) {}
    virtual node & operator = (const node & n) = 0;
    virtual int fromto_xdr (XDR *xdrs) = 0;      // translation from/to XDR
      ...
};
```

Figure 6: Interface for the abstract class `node`.

The `node` derived classes can be divided in two sets: one for the intermediate nodes in the tree and the other for the leaf nodes. The intermediate nodes are represented by the `nodestruct` and `nodevector` classes deriving from `node` and containing the number of elements they involve (fields in records or elements in arrays) and an array of node pointers containing their elements. Both intermediate nodes redefine the method `accedir` with their own code, and define the method `fromto_xdr` as a recursive call to the same method in each one of their elements. The leaf nodes are represented by the classes `nodeenter`, `nodereal`, `nodeboolea` and `nodestring`, also deriving from `node` and containing respectively an integer, a real, a boolean and a string value. These leaf nodes define the method `fromto_xdr` by doing the corresponding translation for their basic type (using the standard XDR routines for basic types).

The use of the compact type definition facilitates the construction of the variable tree (figure 7 shows the node tree representation for a `pyramid` type variable), because it can be done recursively having only basic types in the leaves and building the intermediate nodes as records or arrays of other nodes or leaves.

Although the set of types usable in ATL language is restricted to records and arrays, the presented design for the Atlas variable representations makes the possibility of extensions easy. It would be quite easy to extend the accepted types in ATL to lists or hash tables, for example, by including the description of these types in the language and extending the `node` types (classes deriving from `node`) with `nodelist` and `nodehash`. Because of the modularity in the `Variable` mechanism design the only effort needed to do would be inside these two new `node` derived classes in order to achieve the methods implementation for the translation to XDR and the access to their components.

Besides the possibility of types extensions, Atlas also offers more flexibility to the mechanism by accepting the passing of `Variables` of an unknown type. These variables are not accessible for the command subsystem since it doesn't know what their types are and therefore it cannot access their values; but it allows these variables to pass through its execution only to go from an extern

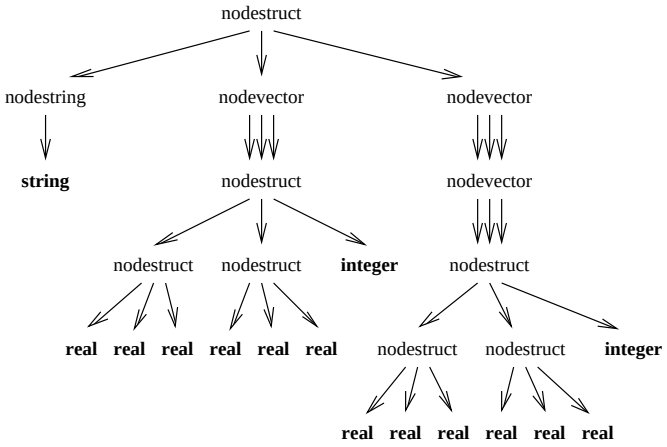

Figure 7: Node tree representation for a `pyramid` type variable.

function to another one, i.e. a `Variable` of an unknown type can be used in ATL to recover the result of an extern function and pass it to another one as a parameter, for example.

These unknown type `Variables` make the Atlas variables mechanism totally flexible because the developer can use any type for his data, but on the other hand the developer must implement the XDR translation for these types by himself since Atlas is not aware of the composition of these types.

4 Achieving transparency for the developer

Not only does Atlas offer the automatic translation between XDR representation and Atlas `Variable` representation, but it also isolates the developer from these Atlas `Variables` representation.

In order to achieve the desired transparency for the developer, Atlas provides code stubs to automatically transfer the user's data into Atlas `Variables`, and backwards, through *bridge types* used to isolate the user from the details of the Atlas `Variables` (which an advanced user can use directly if he wishes to).

These code stubs are automatically constructed by Atlas from the interface declaration of the process (like the example in figure 8), which contains the type definitions used for variables to be exported and the prototype definitions of the process extern routines, which describe the parameters and result types for them. All these definitions give Atlas enough information to generate code stubs to prepare the arguments for user functions or collect results and encode them for being transported over the network, and dispatch calls to user functions.

```
USE se;
        ...
EXPORT #deftype simplex pyramid
EXPORT #deftype scene VECTOR [100] OF simplex
EXPORT #deftype property integer

EXPORT scene totalsc;
        ...
PROT
  EXTERN FUNCTION segmentation (scene sc, property p) RETURNS scene;
  EXTERN PROCEDURE display_scene (scene sc);
      ...
ENDPROT
        ...
EXPORT PROCEDURE SegmentSimplex () IS
  display_scene (segmentation(totalsc,GETDATA("Input the property value")));
  se::Output ("Segmentation completed","m");
ENDPROCEDURE
```

Figure 8: Portion of the interface definition in ATLAS for the volume modeling process ("volum").

The *bridge types* are used to build intermediate objects with the data structure of the user's objects (as per their ATLAS declarations) but without the methods of the user's objects (which remain unknown to ATLAS). Each *bridge type* (also automatically generated by ATLAS) has also methods to translate ATLAS `Variables` into it and an operator to build a `Variable` from it, making then both translations transparent to the developer. The only burden on the developer is then to provide his classes with conversion methods to and from these *bridge type* objects, which is normally trivial (unless the developer choses to have a very different structure for the ATLAS data that the one used internally by his program).

As an example, we can see in figures 9 through 11 portions of the automatically generated code stubs to link with the user code for the example in figure 8. Notice the use of *atl_scene* in figure 11 as a *bridge type* to isolate the user's class (*scene*) from its ATLAS external representation.

5 Conclusions

In this paper the ATLAS `Variables` mechanism and its representation has been reported. A small overview of ATLAS platform has also been presented.

The most important issues in ATLAS for exchanging data between processes are robustness and transparency.

By using the standard XDR representation to pass data through the network, ATLAS ensure these data mean the same for each process that uses them; therefore this exchanging of data between processes becomes robust and reliable because it doesn't depend on the workstation architecture where the process runs.

```
      ...
namespace volum {
typedef atl_pyramid atl_simplex;
}

namespace volum {
struct atl_scene{
   atl_simplex cont[100];
operator Variable() {
   Type t("volum::scene","V[100]S(name string,base V[3]S(p1  S(x real,y real,z real),
                                                          p2 S(x real,y real,z real),
                                                          ident integer),
                                    sides V[3]V[3]S(p1 S(x real,y real,z real),
                                                    p2 S(x real,y real,z real),
                                                    ident integer))");
   Variable v(t,""); v.crea_arbre();
   for (int i1=0;i1<100;i1++)
     { *((*(v.Arbre())).accedir(i1)) = *(((Variable)cont[i1]).Arbre()); }
   return (v);
   }
atl_scene() {}
atl_scene(Variable &v) {
   if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
   for (int i1=0;i1<100;i1++) {
      Variable v2("S(name string,base V[3]S(p1 S(x real,y real,z real),
                                            p2 S(x real,y real,z real), ident integer),
                     sides V[3]V[3]S(p1 S(x real,y real,z real),
                                     p2 S(x real,y real,z real), ident integer))","");
      v2.crea_arbre();
      *(v2.Arbre())=*((*(v.Arbre())).accedir(i1));
      atl_simplex tpaux(v2); cont[i1]=tpaux;
      }
   }
   };
}
       ...
```

Figure 9: Portion of the automatically generated atl_volum.H file.

Transparency is the biggest aim for ATLAS in every mechanism it offers. In the **Variables** design a total transparency is achieved for a wide set of data types and the developer doesn't need to take care of the internal representation used by ATLAS for his data going through the network.

Not only can the **Variables** mechanism be used with the wide set of data types ATL understands, but it is also flexible to be used with *unknown* types (types out of this set), yet in this case flexibility is against transparency.

Finally, ATLAS itself also achieves some transparency with this representation since the ATLAS kernel (process **distr**), which is the processes communication center, doesn't need to know about data passing through it. It doesn't know their value or their type, it only receives and sends **Variables**.

6 Acknowledgments

ATLAS development has been greatly facilitated by the use of the ACE-Wrappers (see [3]) library to build the communications drivers, and also by the PCCTS

```
#ifndef __ATL_volumhh__
#define __ATL_volumhh__

#ifndef NOHEADER
#include "volum.h"
#endif
#include "atl_volum.H"
scene segmentation(scene,property);
void display_scene(scene);

#endif
```

Figure 10: The automatically generated atl_volum.hh file.

```
      ...
void aux_segmentation(String codi,DLList<Variable *> &parametres) {
   Pix p=parametres.first();
   atl_scene ptp0(*(parametres(p)));
   scene par0(ptp0);
   parametres.next(p);
   property ptp1(((nodeenter *)parametres(p)->Arbre())->Getvalor());
   parametres.next(p);
-->  scene res=segmentation(par0,ptp1);
   atl_scene restp;
   restp=res.conversio_a_tipus_pont();
   Variable *vr=new Variable(restp);
   ReturnValue *rv=new ReturnValue(codi,vr);
   distrib.envia(rv);
   }

      ...
void main(int argc,char **argv) {
      ...
   ini_process();
   driv.Dispatch();
   close(CANAL_COMUNIC_DISTR);
   exit(0);
   }
```

Figure 11: Portion of the automatically generated atl_volum.C file. The arrow has been added pointing to the point where user code is actually invoked.

(see [7]) compiler construction tool.

This research has been supported partially by grants TIC-92-0605 and TIC-95-0630-C05-01 of the CICYT

References

[1] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1), March 1991.

[2] R. Srinivasan. Rfc 1832: Xdr: External data representation standard, August 1995.

[3] Douglas C. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *12th Sun Users Group Conference*, 1994.

[4] Marta Fairén and Àlvar Vinacua. ATLAS. Sistema de Comandes: Manual tècnic (in Catalan). *Report LSI-95-11-T*, 1995.

[5] Jon Siegel. *CORBA Fundamentals and Programming.* Jon Siegel. OMG, 1996.

[6] Marta Fairén and Àlvar Vinacua. Interacción Gráfica en ATLAS (in Spanish). 1997. To appear in Proceedings of CEIG'97.

[7] Terrence J. Parr. Language Translation Using PCCTS and C++ (A Reference Guide), June 1995. Address: http://www.parr-research.com/ parrt.