How to Distribute an Existing Modular Expert System Shell?

A.-M. Dery W. Lejouad C. Djossou

pinna@sophia.inria.fr lejouad@sophia.inria.fr djossou@sophia.inria.fr

INRIA Sophia-Antipolis France

Abstract

In this paper we point out the reasons that lead us to distribute modular applications and how we distribute them. We illustrate our ideas with an example of expert system shell.

1 Introduction

A modular system has the advantages of giving a clear view of its structure, highlighting its main functionalities and reusing them. The module autonomy is limited when we want to open complex tools, such as an expert system shell, to the external world. Distribution is then well adapted to answer to this need. With this architecture, the resulting parallel processing increases the system efficiency, the protocol definition clearly specifies how a component can be connected to another, and changing the version of one of the components becomes easier. Although many distribution constraints have to be managed. In the rest of this paper, we describe our SMECI [1] experiment and we offer some guidelines for the design of such distributed application.

2 Smeci expert system shell

SMECI is an expert system shell used for design and planning, which is considered as a multiformalism environment for developing knowledge based applications. It groups several components, namely, frames, inference rules, tasks and states. All these components are integrated in a single process residing in a single memory. Our purpose is to separate the main functionalities of SMECI and to distribute them on multiple processes running on possibly distinct memories. With this architecture, the different components can be reused and furthermore some of them can be associated with external tools to provide new functionalities.

3 Guidelines for distributed application design

In this section, we give a methodology for the design of distributed applications. We start by specifying the main system components. These components or *modules* separated on distant machines have to cooperate to solve a problem. For this purpose, the system has to be provided with a type and a protocol of communication.

3.1 From modularity to distribution

An *autonomous* module is provided with appropriate instructions to enable it to perform the maximum operation on its own. But if we look at the whole system running as a single process, where all modules are linked together, the autonomy associated with one module will be lost.

To build a distributed application, a designer has to define the components and to specify for each one which type of services it offers and which type of services it needs. A component implementation is private and only its interface and the description of its service are known by the other components. This step is close to module design or object-oriented application design.

A component can correspond to an independent entity managed by a process. This entity can be made up of several modules or can be localized on a parallel machine and implemented with several processes according to the nature of the application and the environment in which it is developed.

Therefore, distributed processing suffers from difficulties of consistency control and management. Separation of processes (servers and clients) by a network introduces the possibility of performance degradation due to communication delays.

When we spread components over many sites in a distributed network, we wish to satisfy two objectives: maximizing parallelism of process execution and minimizing intersite data traffic. For this purpose, a study must be done on the placement of the different modules.

When we are concerned with distribution architectures, we not only specify the main system components but also we have to deal with some of the following problems: are there shared data? Where must they be placed, and how can one handle the concurrent access of multiple components? How can the client detect a server crash, and conversely? How can we avoid losing data? What happens if a fault occurs when processing is going on?

3.2 Communication mechanism

Several types of communication can be considered. The two main ones are *client-server* model and *broadcast* model. The most famous example of synchronous client-server communication protocol in the INTERNET architecture is Remote Procedure Call (RPC) [3].

The *broadcast* communication is based on the diffusion principle. This mode is a typical example of asynchronous communication. Encapsulation technique represented by tools such as Sophtalk [2] and Tooltalk [4] are examples of broadcast communication.

Once the designer of a distributed application has isolated the basic components and has chosen the communication protocol, he has to define application protocols which specify how components converse. A well-defined protocol (open and extensible) facilitates tools integration and maintenance in a distributed existing application. An application protocol specification must take into account network specificities such as server failure, response time,... but it must remain as independent as possible of the communication platform. Many high level tools can be used to free the designer from the burden of dealing with standard communication protocol such as the generator stubs MAVROS and RPCGEN.

3.3 Smeci experiment

As we have explained above, to distribute SMECI, we have to separate its main components and to answer to architectural constraints such as : do we have to isolate the unifyer or not? In SMECI, we wish to unify as many rules as possible at one time and at one site. The question is whether we have to group the *unifyer* and the *expertise server* in a single component, or whether we can separate them and provide a single transaction to get the list of candidate rules to be processed? The second approach satisfies the distribution and efficiency constraints. We finally proposed four principal modules, namely, an *object server*, an *expertise server*, an *inference engine* (which includes a *unifyer*) and a *reasoning trace* (see Figure 1).

Following the distributed architecture we have proposed, we have developed the application protocol needed to transfer the application objects between components. This protocol is based on *client-server* model and is implemented in RPC Lisp.

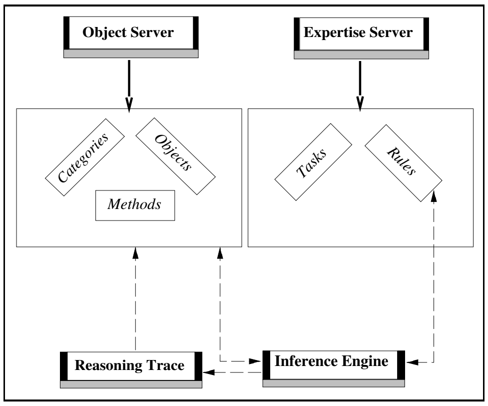

Figure 1: Distributed SMECI architecture

4 Conclusion

As we have shown, distribution is an interesting approach, hence many constraints need to be managed. Moreover, it is important to minimize the impact of distribution on the final user thanks to high level tools such as a configuration tool and a dialogue manager.

References

[1] Inria and Ilog. *SMECI: Reference Manual.* Gentilly, 1990.

[2] I. Jacobs and F. Montagnac. The Sophtalk Reference Manual. Rapport technique 150, INRIA, 1993.

[3] SUN Microsystems. *Network Programming Guide.* USA, 1990.

[4] A Sun Microsystems Subsidiary. *The Tooltalk Service*, June 1991.