INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

From Modular to Distributed Architectures: Centaur and Smeci Experiments

Anne-Marie Dery, Wided Lejouad

N° 2398

Novembre 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

rapport de recherche

1994

ISSN 0249-6399

From Modular to Distributed Architectures: Centaur and Smeci Experiments

Anne-Marie Dery*, Wided Lejouad**

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Secoia

Rapport de recherche n° 2398 — Novembre 1994 — 25 pages

Abstract: In this report we point out the reasons that lead us to distribute modular applications, why we think that modularity is a critical step – but not a final one – in the design of distributed applications, and how we distribute. We illustrate our ideas with two experiments developed in the CROAP and SECOIA projects at INRIA***. They concern respectively CENTAUR and SMECI tools.

Key-words: Programming environments, expert systems, distributed systems, communication protocols, client-server architecture.

(Résumé : tsvp)

*Anne-Marie.Pinna@essi.fr
**Wided.Lejouad@sophia.inria.fr
***Institut National de Recherche en Informatique et Automatique.

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

Le passage d'une architecture modulaire à une architecture distribuée : exemples des systèmes Centaur et Smeci

Résumé : Dans ce rapport, nous présentons les raisons qui nous ont amené à répartir des applications modulaires existantes, pourquoi nous pensons que la modularité est une étape essentielle mais non suffisante dans la conception d'applications distribuées et nous décrivons la démarche que nous avons suivie pour réaliser cette distribution. Nous illustrons ces points par nos deux expériences qui concernent les outils CENTAUR et SMECI développés respectivement dans les projets CROAP et SECOIA.

Mots-clé : Environnements de programmation, systèmes experts, systèmes distribués, protocoles de communication, architecture client-serveur.

Contents

1 Introduction

Distributed systems began to appear in the 1970s, following the availability of computer networks. Since, the key computer technology has been information gathering, processing, and distribution [37] [10]. The idea of distribution is relevant to many areas of computing, indeed all of them. We can find distributed database systems, distributed artificial intelligence systems, distributed software engineering tools, etc. Perhaps the most well-known distributed application in the computer world is the X Window System [6].

In this report, we discuss the common concepts as well as the most striking differences between modular and distributed systems. We describe the advantages of distribution, and some reasons for choosing this approach. We also offer some guidelines for the design of such distributed applications. We illustrate each of our points with concrete examples from our design and implementation experience with the distribution of two applications, a programming environment generated by the CENTAUR system [4] [18] and the SMECI expert system shell [19].

2 Description of used tools

In addition to SMECI and CENTAUR, which will be described in detail in paragraphs 2.1 and 2.2, we work on SUN workstations under the UNIX operating system and an X Window environment. So we are concerned with neither parallel languages nor parallel machines.

2.1 Smeci Experiment

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 concepts, namely, frames, inference rules, tasks and states. It has been chosen as an example for distributing an expert system shell.

The frame language is needed to structure knowledge, the rule language infers new facts in the knowledge base. The task-oriented programming allows the decomposition of one problem into several sub-problems, and consequently the structuring of the rule base in sets of rules associated to a specific sub-problem. While reasoning, SMECI builds a state tree which records the object world evolution. The state tree not only shows the paths leading to solutions, but also explains how to reach them.

All these components are implemented in LE_LISP [16]. They 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. We chose three principal modules: an *object server*, an *expertise server*, and an *inference engine*. The latter is a set of two sub-modules: the filtering module and the deduction module whose role consists in executing rules on the object server (see Figure 1). This architecture provides a cooperation of a set of small processes instead of one big process. Moreover, the amount of information can be partitioned on several machines, this increases the capacity of the whole process to support more quantities of data. For example, the distribution of an expert system shell allows the integration of more than one object server, so the system can be extended easily. The main role of an expert system shell is to make the object servers evoluate. We then consider objects as the basic entities to be manipulated remotely. For this purpose, an object transfer protocol has been developed.

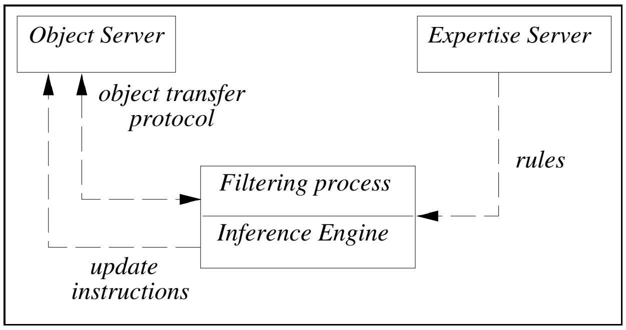

Figure 1: Distributed SMECI architecture

2.2 Centaur Experiment

CENTAUR is a system facilitating the development and use of programming languages. With CENTAUR, the user may define the syntactic and semantic aspects of a particular language and create automatically the appropriate tools to handle programs written in this language. Among the most common tools, are a *parser*, a *structured editor*, a *type-checker*, a *pretty-printer*, and an *interpreter*. Each of these tools is generated within a specific target language suitable to its type of activity (PROLOG, LISP or C). All these tools handle programs as *abstract syntax trees*, which can be created, modified and tailored by the *Virtual Tree Processor* (*VTP*). Figure 2 illustrates an example of a distributed CENTAUR application.

The main part of CENTAUR distribution is the definition of protocols specifically for the transfer of abstract syntax trees between architecture components. Such data protocols facilitate the communication between CENTAUR and external tools, as well as among internal ones. In fact the CENTAUR distribution is only an example and an experiment for distributing programming environments.

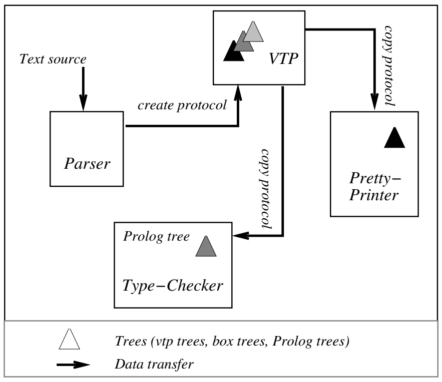

Figure 2: Distributed CENTAUR application

2.3 Common objectives

Our experiments have in common the fact that they offer a methodology for the design of distributed environments from existing tools. The main work consists in defining the basic components, specifying and implementing the application protocol that handles the communication between them. Nevertheless this first step is not sufficient. We also have to study the way to reduce the communication costs for remote access, to manage consistency of the shared data and to facilitate the final use of the distributed tools. In the following section, we point out the reasons that led to distribute environments. We illustrate our remarks with SMECI and CENTAUR examples [1].

3 From modularity to distribution

In this section we briefly analyse the modularity notion and we underline some of its limits which are released with distribution.

3.1 Modularity

Monolithic (non modular) systems are not easily updated and are not reliable. The first issue was to provide a well structured *modular system*, supporting the reuse concept. In modular systems, we deal with sequential processing based on sequential algorithms. The main advantages of modular systems are to provide a better design and to make maintenance easier. Consequently, modularity allows to reuse existing modules, to have several programmers working on the same application, to make some modules available as libraries with abstract interfaces, to provide parallelism through the use of languages dealing with parallel tasks such as ADA, and to facilitate version generation by using incremental links. Hence, a modular system has the advantages of giving a clear view of its structure, highlighting its main functionalities and reusing them.

Modularity has also some `limits`. 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.

[1]To underline those examples in the text, we introduce a specific format with centered items.

> CENTAUR has been designed as a modular application. In SMECI, modularity has been taken into account progressively. Reuse, extensibility and maintenance are preserved, but this is true only in CENTAUR and SMECI environments; problems occur when we integrate heterogeneous external components or we export internal ones.

Modular systems are faced with ever increasing quantities of data, faster data update rates, and more response-time requirements. They need a huge memory size to support the system itself, the programming language, and the user application. A processor may reach a saturation point slowing down the processing time.

> In CENTAUR, programming languages cohabit in a single executable, and in spite of code optimizations it is a big tool. In SMECI all functionalities: task management, rule bases, state tree building, and reasoning, are grouped in a single complex process.

In modular systems, modules can be reused more easily if they are developed in one programming language. Moreover, modules cannot be executed simultaneously and parallelism is not effectively provided, it is reduced to *time sharing*.

> ADA offers parallelism using independent tasks. These tasks are executed on a single physical processor. A scheduling algorithm is implemented to partition time between running tasks.

The natural evolution of complex tools such as SMECI and CENTAUR leads to open them to the external world. This means that some particular internal components should be exported and used in another context, or that the tool should be extended thanks to a communication with an existing tool. So, a complex tool should be an unstable set of components. In case of modular systems, these components are embedded in a single process which can become a big one hardly supported by current processors. Moreover, the import or export of new components from or to the external world is a difficult operation when components are heterogeneous. Thus, the role of modularity becomes limited. Distribution offers not only all the modularity advantages, but also answers to these needs. Let's see in detail those advantages.

3.2 Distribution Goals

Before we start to develop the different ideas and technical details, we specify in the next sections why we are interested in building distributed systems.

3.2.1 Autonomy

Each process has to be provided with appropriate instructions to insure its self reliance, but at some point it may collaborate with others to reach the desired goal. In modular systems, the programmer studies the way to provide autonomous modules and the interactions between them. These interactions are taken into account during the link phase which is performed locally. Unlike modular systems, in the distributed ones, the programmer has to study the way to make modules as autonomous as possible and has to be worried about their interactions, and that not only for code clarity, but also to minimize communications through the network. A component that takes some inputs and gives some outputs without collaborating with another component is considered as completely autonomous.

> The RODEO project which is specialized in network technology has implemented a *Compiler* for the ASN.1 specification language named MAVROS [35]. In another hand, the CROAP project has defined a complete ASN.1 environment with CENTAUR. MAVROS is considered as an autonomous tool waiting for the ASN.1 specifications to produce a C program corresponding to the network data representation. It is the role of the application protocol to make possible the connection between the environment developed with CENTAUR and the MAVROS compiler.

3.2.2 Parallelism

A distributed system is seen as a set of processes, in which each one attempts to deal with its own task. Every process can run on an independent processor. Parallel processing may increase the system efficiency and may supply interactive services with reasonable response time. To offer a great parallelism, several processes at many sites are provided to run autonomously. This simultaneous running increases the communications between components, and produces conflicts due to concurrent accesses to some shared resources. This leads to performance degradation due to communication delays and conflict resolution (see § 3.3).

> In a CENTAUR application, one can ask the *Pretty-Printer server* to pretty-print a program and simultaneously ask the *Type-Checker* to type-check it.
>
> In SMECI, we can make active many inference engines simultaneously at many sites. In this case, we can fire several rules at a time.

3.2.3 Reuse and heterogeneity

A component may have other uses beyond the scope of one particular application, in this case we allow its external reuse. Reuse makes progressive evolution easier. Therefore, we are faced with some obstacles, namely, *the incompatibility* of the components' programming languages and the unavailability of their source codes. These difficulties are reduced in distributed architectures. The protocol definition clearly specifies how a component can be connected to another; all the entities that are faithful to the same communication protocol can converse independently of their own implementation.

> In programming environments generated by CENTAUR, we wish to reuse tools such as an existing CAML [2] *parser*. To reuse CAML without worrying about its programming language, a communication protocol between the parser and CENTAUR is needed to generate new tools, such as an evaluator, a pretty-printer, etc.
>
> In SMECI, we may need to integrate a *type-checker* generated by CENTAUR and intended for some formalism, like the rule language, which is integrated in the *expertise server* of SMECI [24].

[2] Functional language in the ML family.

3.2.4 Data sharing and Collaboration

Designing distributed environments makes programs, data and treatments available, regardless of their physical location, to any user on the network.

> In SMECI and CENTAUR, the *object server* and respectively the *VTP server* can be seen as shared databases. In CENTAUR, the *VTP* presents a common database that the other components access for read or for write. In SMECI, the filtering process sends a query to the object server asking for the set of objects needed to filter rules, and the inference engine creates and updates them.

Distribution is useful when we are a contracting party for designing a tool or for an international project (ESPRIT Project), and when we want a tight collaboration with other partners.

> A graphical interface has been designed in CENTAUR for the HOL theorem prover [38]. In order to insure a good collaboration between the HOL implementors and the CENTAUR designers, both tools communicate through a network to exchange data.

3.2.5 Maintenance

In modular applications, changing the version of one of the components (bug fixing or functionality extensions) entails the generation of a new version for the complete application: the updated module has to be compiled and linked with the others. However, the maintenance becomes easier when the application is distributed because only the concerned component is modified, there is no need to link components. Furthermore, the application protocol insures the global system consistency.

> We improve the maintenance aspect in CENTAUR with the *VTP* component. The *VTP* has three implementations: the CENTAUR version in LISP, a C++ version and a work in progress with the TELOS object-oriented language [17] that will be the future version of the *VTP*. We have tested these versions with the tree transfer

protocol and we can now claim that the transition from the current version to the new version will be greatly facilitated by the distribution technique.

Let's take an example of the X Window system. The X server has a window manager XTWM as a client. Changing the window manager version from XTWM to VTWM in a modular centralized environment entails killing the X server, compiling the new version and linking it. This work is not necessary in a distributed environment.

The main points that interest us in building distributed systems have been illustrated with concrete examples in this section. Before describing the method we use to distribute, it is important to discuss the problems faced with distribution.

3.3 Problems faced with distribution

Distributed processing [29] suffers from difficulties of consistency control in the case of parallel processing and distributed data management. Separation of processes (servers and clients) by a network introduces the possibility of performance degradation due to communication delays. These delays may be due in part to the volume of traffic through the communication network.

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.

In SMECI, we wish to unify as many rules as possible at a time. The question is whether we have to group the *filtering process* and the *expertise server* in a single component, or we 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.

In a parallel processing, we can be faced with some problems when the tasks are not completely independent: if they concern the same data or if they have a consequence relation.

In a CENTAUR application, we have to solve conflicts if we ask simultaneously for the editing and the type-checking of the same program. The read and write locking techniques are implemented to resolve these conflicts.

In SMECI, a new state cannot be built before the binding of the inference rule variables by the *filtering process*. In order to improve system efficiency, the *filtering process* can be applied to several rules concurrently instead of executing them consecutively. The process filtering sends a query to the object server (read operation), when the inference engine updates this server (write operation). Some concurrent accesses could occur when the inference engine of one site updates a set of objects that the filtering process would like to read. The solution consists in re-firing rules until no new substitutions are found. This allows the filtering process to read a correct value in the next pass over the rule base.

Many network constraints must be satisfied when dealing with distributed systems. In this section, we give a non-exhaustive list of these constraints:

- 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?

In CENTAUR we have to transfer abstract syntax trees corresponding to programs. The trees can be large and we decided to use asynchronous communication. The trees are sent node by node. This application protocol is a set of calls of node building primitives. This solution simplifies recovery in case of fault. In SMECI, the data to transfer are objects. We have defined an object transfer protocol where objects are sent in a single bloc.

Having shown the difficulties we could face in dealing with distributed architectures, we will describe in the next section how a designer moves from a modular application to a distributed one.

4 Distributed Application Design

In this section, we give a methodology for the design of distributed applications. We start by specifying the main system components, which usually correspond to the main system functionalities. 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 (see § 4.2.1 and § 4.2.2).

4.1 Modularity step

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 is known by the other components. This step is close to module design or object-oriented application design.

The granularity is perhaps different but slightly related to modularity and autonomy concepts. 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: this is an implementation choice depending on the nature of the application and the environment in which it is developed. Moreover, a component may need to solve an easy or a complex task; its granularity depends on what it can do; we speak about its degree of intelligence as in Distributed Artificial Intelligence (DAI) systems [3].

> In the case of programming environments, applications consist of well defined components, namely, *parsers*, *pretty-printers*, *evaluators*, *compilers*, and *type-checkers* (see Figure 2). Each component provides a specific service, and can make calls to different entities. For example, an editor must parse the source text it has to insert. Some of these components are themselves modular. A *pretty-printer* has two internal modules: a tree compiler and a graphical machine. These two modules may be separated in two processes.
>
> In the SMECI experiment, we wish to distribute an application developed by an expert system shell. For the modularity, we used a

bottom-up approach with *Data Flow* analysis. It consists first in realizing a structural and functional decomposition, then in defining dependency links corresponding to inputs and outputs of each entity. This yielded the main modules explained in section 2.1 and illustrated by Figure 1.

4.2 Communication

The set of components with their input and output descriptions are defined in the same way as modules with import and export clauses. Between modules, the communication is static. The names of the modules from which procedures are imported must be specified at compile-time and present at link-time. In our case – distributed applications – the communication between components takes place across a network and it can take different forms.

4.2.1 Communication Types

Several types of communication can be considered. The two main ones are *client-server* model (see Figure 3) and *broadcast* model (see Figure 4). The former one is the most widely-used interaction mode between applications: all standard high-level services proposed in the INTERNET and OSI architectures function according to it. The most famous example in the INTERNET architecture is Remote Procedure Call (RPC) [2].

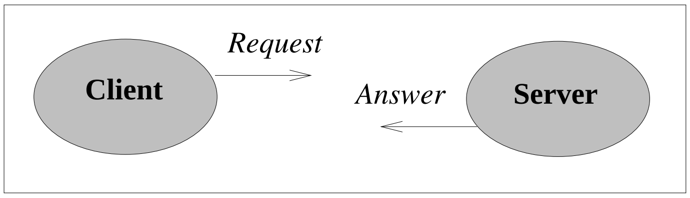

Figure 3: *Client-server* model

In the case of RPC, when the client asks the server to execute a procedure, it is locked waiting for the result (i.e. synchronous). An extension of this request-response model has been implemented to make the client continue the running process without worrying about information arrival (i.e asynchronous). Asynchronous communication raises the system efficiency and helps the parallelism implementation. When an application is complex (many components and great data exchange rate) it is not always reasonable to lock the process waiting for data arrival; in this case asynchronous communication would be well suited.

The *broadcast* communication is also widely-used; it is based on the diffusion principle. Requests are sent on the network and all components able to answer, deal with these requests. This mode is a typical example of asynchronous communication. The distributed application designer has to choose between these types.

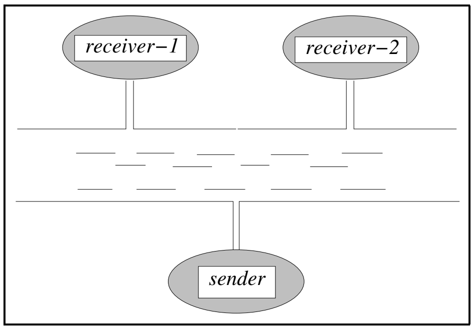

Figure 4: *Broadcast* model

In the CENTAUR architecture, we have a mixed solution: some components communicate directly following the *client-server* model and others address their requests to a dialogue manager. We attempt to preserve as often as possible an asynchronous communication: the protocol of tree transfer is built with this principle.

In the case of SMECI, the *client-server* model is considered as a natural organization. The *expertise server* is considered as a server because it provides rules, and as client because it asks for their firing. The *inference engine* is also considered as a client because it asks for the set of objects needed to filter rules and asks for their execution. Finally, the *object server* is only a server, it provides objects and executes rules to enrich the database. The communication between components is performed in deferred mode. With this mode, the client continues its processing when sending a query (asynchronous mode), and is locked when the result is needed.

To integrate communication primitives in software components, we have to specify which "interprocess communication" (IPC) mechanism to use and on which architecture it is based. We dispose of a wide range of communication protocols evolving from the lowest to the highest level. We describe them in the next section.

4.2.2 Communication protocols

The choice of the appropriate communication protocol depends on the designer's environment and needs. It also depends on the programming language in which the components are developed. The IPC mechanisms we studied are the following:

Low level tools

- *Transmission Control Protocol:* TCP is a reliable stream transport service provided by the TCP/IP Internet protocol. It specifies the format of the data and acknowledgements that two computers exchange to achieve a reliable transfer, as well as the procedures the computers use to ensure that the data arrive correctly [10].

- *Unix Socket:* UNIX *socket* is a generalization of the UNIX file access mechanism that provides an endpoint for communication. *Sockets* can be used with traditional operations like *read* (receive) and *write* (send) [31].

High level tools

- *Remote Procedure Call:* In this protocol, in order to call a remote procedure, the client calls a local *stub*. This stub *marshals* (encodes) the parameters and transmits a request message to the server. When the message arrives, the parameters are *unmarshaled* (decoded) and the server procedure is called. The eventual result follows the same path in the reverse direction [37]. We recall that C is the programming language used with RPC. A functional interface LE_LISP/C, called RPC-LINK, has been developed to allow the communication between processes developed in C and LE_LISP [17]. A distributed environment manager, called CHOOE [23], has been recently developed to control process synchronization and communication.

- *Stub generators:* Many high level tools are developed to free the designer from the burden of dealing with low level routines. Network data representations (e.g. XDR[SUN] and ASN.1[OSI]) are abstract specification languages used to describe data to be propagated on the network. Associated to *socket* handling (e.g. RPC[SUN] and ROSE[OSI]) and stub generators such as (e.g. RPCGEN[SUN] and MAVROS[OSI]), they facilitate the development of client/server programs. They handle the connection between components, the marshaling and unmarshaling of data and the RPC protocol. All these tools allow the heterogeneity of systems and programming languages.

- *Encapsulation Technique:* Encapsulation technique is represented by tools such as SoftBench, Field, Sophtalk and Tooltalk [36, 5, 30, 7, 8]. An object is encapsulated within *a node* provided with *input* and *output* ports. The inputs correspond to primitives supplied by the object, the outputs correspond to messages called *events* that an object could emit. Sophtalk is used in the SECOIA project to allow knowledge-base systems to communicate and collaborate to solve a problem [13].

After this overview on the communication systems, we underline, following our experiments, their characteristics. First, sockets are considered as a low level inter-process communication useful in the case of applications when users need to manage the underlying communication mechanisms. In the other hand, RPC-LINK, which is based on sockets, makes easy exchanges between LE_LISP and C processes and handles the serialization of data through the network. In the same kind of tools, ASN.1/MAVROS is a useful heterogeneous platform still in progess. Finally, SOPHTALK can be considered as a high level tool facilitating complex architecture design, hence it remains poor for heterogeneous low level communication.

In SMECI, components are based on the client-server model and developed in LE_LISP. We thought that RPC-LINK is well-suited for such an application. The process network created in an application generated by SMECI has been managed by CHOOE, which is a distributed environment manager based on RPC-LINK.

An application generated by CENTAUR is more complex. It is based on the client-server and the broadcast model in the same time. The encapsulation technique can provide a mixed model.

Application protocols

Once the designer of a distributed application has isolated the basic components and has chosen the communication protocol, he has to define application protocols. Application protocols specify how components converse. In order to ask a service, the client must know the conversation modality. A well-defined protocol (open and extensible) facilitates tools integration and maintenance in a distributed existing application. The protocol could also be reused in another context. 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.

In CENTAUR, we specified an application protocol for data transfer between components. This protocol is specific to tree propagation, it assumes that trees can be large and that tools should not be blocked waiting for a data reception. It is designed in such a way that trees could be re-sent in case of communication failure.

In SMECI, the basic element of communication between components is the object. We developed an object transfer protocol in which objects and the semantic links between them are preserved when they migrate from one process to another. A message protocol is also developed to enable the inference engine to send the update instructions needed following the rule filtering.

4.3 Process synchronization

Protocol definition and implementation present the connection between distributed application components. Although it is not sufficient, distribution difficulties and constraints need to be managed: the consistency of shared data, the synchronization of some services and the configuration of an application. It is important to minimize the impact of distribution on the final use. Users should not be network specialists, and moreover the use and the design of their distributed applications must be facilitated. To solve this type of problem, we are interesting in high level tools such as a configuration tool and a dialogue manager [21, 26, 14, 9].

A study for CENTAUR leads us to define a configuration tool based on a library of components provided to the users from which they can build their own application. This tool should mask the location of each component (machine or network address) and the connection between them: it concerns the *transparency* concept. The user converses only with one component. The latter manages the service calls by addressing them to the appropriate component(s) and manages some of the synchronization aspects such as sequentiality or exclusion. For example, the user that asks for *editing*, *type-checking* and *pretty-printing* a program, addresses its request to the dialogue manager. An edition can modify the program (abstract syntax tree), so it must be completed before requesting *type-checking execution*. Requests to the *type-checker* and *the pretty-printer* can be sent at the same time and their actions parallelized because they are both readers.

> In SMECI, the process synchronization control is performed by CHOOE. Every client owns a mailbox which holds the answers to its queries. This mailbox makes the client able to perform other tasks, and checks the content of the mailbox only when results are needed to carry on the eventual processing. For example, the expertise server sends in parallel, to the inference engines, several rules to fire. The expertise server has to stop when all inference engines are busy and it could pursue its work only when it receives in its mailbox the identity of the free processes.

5 Conclusion

We have reviewed some of the developments which are being pursued by two research projects in distributed system area. We were concerned with the distribution of software engineering and artificial intelligence tools: CENTAUR and

SMECI. We started by giving the reasons to design distributed systems, what they provide in comparison to modular ones, and describe how we proceed and which methodology we follow for the design process.

Both applications have been recently developed. Some of our objectives have already been achieved. In SMECI, following the distributed architecture we have proposed, we are integrating the remote functions calls needed to transfer the application objects between components. Several object servers and inference engines can be included in the environment. Components co-operate in a client-server model and communicate in a deferred mode [25]. In CENTAUR, semantic tools and a parser are already running as remote processes which communicate through data protocols. High level tools such as a configuration tool and a dialogue manager have been specified but are not already integrated. This experiment continues in the context of a `MESR`[3] project and is still under specification and development. We also collaborate with the RODEO project at `INRIA` on the OSI platform called MAVROS. Such a collaboration is useful for both domains: *software engineering* and *networks*. For the latter, our needs have implied research work around heterogeneous communication [20] and distribution of existing tools.

[3]French Research Ministry

References

[1] AAAI. *Eighth National Conference on Artificial Intelligence: Volume One*, Massachusetts Institute of Technology, Cambridge, 1990. The MIT Press.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39 – 59, February 1984.

[3] A. H. Bond and L. Gasser. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, Palo Alto, CA, 1988.

[4] P. Borras and al. Centaur : the system. Rapport de recherche RR-777, INRIA, décembre 1987.

[5] M. Cagan. Hp softbench : an architecture for a new generation of software tools. Technical Report Revision 1.4, SoftBench Technical Note Series, November 1989.

[6] J. Campbell and D. Comer. The X Window System. *Software Practice & Experience*, 20(S2), October 1990.

[7] D. Clément. A distributed architecture for programming environments. Rapport de recherche 1266, INRIA, juillet 1990. Also in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, R.N. Taylor, Irvine California, December 1990.

[8] D. Clément. Integrated software components : A paradigm for control integration. In *European Symposium on Software Development Environments and case technology*, Konigswinter, Germany, 1991.

[9] R. Cole. Application configuration in a client-server distributed system. In *Proceedings of the International Workshop*, pages 309–317, 1992.

[10] D.E. Comer. *Internetworking With TCP/IP VolI: Principles, Protocols, and Architecture*. Prentice-Hall, Purdue University West Lafayette, second edition, 1991.

[11] G.F. Coulouris and J. Dollimore. *Distributed Systems: Concepts and Design.* International Computer Science Series. Addison-Wesley, Great Britain, 1988.

[12] Ecole d'été Inria. *Construction des systèmes d'exploitation répartis.* INRIA-IRISA-RENNES-BULL-C3-OSF Research Institute, Saint-Malo France, août-septembre 1990.

[13] C. Djossou. A client-server based architecture for communication between expert systems. In W.W. Koczkodaj, P.E. Lauer, and A.A. Toptsis, editors, *4th International Conference on Computing and Information*, pages 231–235, Canada, May 1992.

[14] H. Gomaa. Structuring and configuring distributed applications. In *Proceedings of the International Workshop*, pages 18–32, 1992.

[15] M. Gondran. *Introduction aux systèmes experts.* Eyrolles, Paris, 1984.

[16] Ilog. *Le-Lisp : Le manuel de référence.* Gentilly, 1989.

[17] Ilog. *Le-Lisp Communication Inter-Processus (RPC).* Gentilly, France, décembre 1991.

[18] Inria. *The CENTAUR User's Manual.* Sophia-Antipolis, 1989.

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

[20] A. Jaber and T. El-khatib. Communication lisp-c en environnement distribué hétérogène. Rapport de stage, ESSI, Septembre 1992.

[21] G.R.R. Justo and P.R.F. Cunha. Programming distributed systems with configuration languages. In *Proceedings of the International Workshop*, pages 118–127, 1992.

[22] S.G. Kochan and P.H. Wood. *Unix Networking.* Hayden Books. Pipeline Associates Inc, USA, 1989.

[23] F. Lebastard. CHOOE : un gestionnaire d'environnement distribué. Rapport technique 93–22, Inria-Cermics, 1993.

[24] W. Lejouad. Programming environment for expert system shell. In *Proceedings of PANEL'91: XVII Conferencia Latinoamericana de Informatica*, pages 249–265, Universidad Simon Bolivar Caracas Venezuela, July 1991.

[25] W. Lejouad. *Etude et application des techniques de distribution pour un générateur de systèmes à base de connaissances.* Thèse de doctorat, Université de Nice, 1994.

[26] J. Magee, J. Kramer, and M. Sloman. Constructing distributed system in conic. In *Proceedings of the International Workshop*, pages 118–127, 1992.

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

[28] R.M. Needham and A.J. Herbert. *The Cambridge distributed computing system.* International Computer Science Series. Addison-Wesley Publishing Company, University of Cambridge, 1982.

[29] Y. Ohno. *Distributed Environments: Software Paradigms and Workstations.* Springer-Verlag, Japan, 1991.

[30] S. P. Reiss. Connecting tools using message passing in the FIELD environment. In *IEEE Software*, pages 57–67, July 1987.

[31] J.M. Rifflet. *La Communication sous Unix.* McGRAW-HILL, Paris, 1990.

[32] J. Shirley. *Guide to Writing DCE Applications.* OSF Distributed Computing Environment. O'Reilly & Associates, Inc., USA, 1992.

[33] H. A. Simon. Artificial intelligence: Where has it been, and where is it going. *IEEE Transactions on Knowledge and Data Engineering*, 3(2):128 – 136, June 1991.

[34] R. Snodgrass and K. Shannon. Fined grained data management to achive evolution resilience in a software development environment. In *Proceedings of the Fourth ACM Sigsoft Symposium on Software Development Environment*, pages 144–156, 1990.

[35] D. Steedman. *Abstract Syntax Notation One: ASN.1. The Tutorial and Reference.* British Library Catalogue in Publication Data, 1990.

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

[37] A.S. Tanenbaum. *Computer Networks.* Prentice-Hall, Vrije Universiteit Netherlands, second edition, 1988.

[38] L. Thery, Y. Bertot, and G. Kahn. Real theorem provers deserve real user interfaces. Rapport de recherche 1684, INRIA, November 1992.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399