Jocaml: mobile agents for Objective-Caml

Sylvain Conchon and Fabrice Le Fessant
(Email: {Sylvain.Conchon,Fabrice.Le_Fessant}@inria.fr)
INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

May 3, 1999

Abstract

Jocaml is a system for mobile agents built inside the Objective-Caml language. Jocaml eases the development of concurrent, distributed and mobile agent based applications, by expressing useful distribution abstractions using a small set of simple but powerful primitives taken from the Join-Calculus[FG96].

The system provides total transparency for migration, application states (after migration, all threads resume their execution in the state before migration), communications (communication channels with other agents are kept during migration) and composition (sub-agents migrate with their parent agent). Other features of the Jocaml system are mobile objects with transparent remote method invocation, distributed garbage collection, failure detection and execution efficiency. Jocaml has already been used in several applications, such as a mobile editor, some distributed games and a distributed implementation of Ambients[CG98].

This paper describes the Jocaml programming model and language, its current implementation and some interesting applications.

Keywords: distributed language, mobile agents, join-calculus.
Contact Author: Fabrice.Le_Fessant@inria.fr

1 Introduction

Developers expect three main qualities from mobile agent systems: *simplicity*, agents should be easy to create and manipulate; *expressiveness*, agents programs can make complex distributed computations; and *consistency*, abstractions and primitives should have a strong semantics, so that programs behaviors should be predictable.

In the last ten years, many agents systems have been designed, with different achievements in these three directions.

Agent-Tcl[Gra95], TACOMA[JRS95] and Telescript[Whi96] are examples of agents systems based on scripting languages. They achieve simplicity and consistency, thanks to a few powerful primitives to manage agents. However, they can hardly be used to write complex software projects, since their scripting languages do not provide enough structure, such as modules, interfaces, objects and classes, or user-defined types. Moreover, they are often efficient only for some specific tasks, whereas offering very poor performances for other operations.

With the emergence of Java, *"a language designed for Internet"*, researchers have seen a new hope for mobile agents systems. However, a problem immediatly appeared: since threads states are not directly available from the API, migration of executing threads is not possible without significant modifications of the virtual machine. Thus, systems based on Java either do not allow strong migration of agents, losing some simplicity and expressiveness, or use modified virtual machines [ARS98], gaining transparency but losing the main interest of Java, the standard Java interpreter installed on *"every computer in the world"*.

Moreover, Java-based agents systems often lack a good semantics. For example, objects behaviors during migrations are hardly predictable. Even when some semantical basis are introduced, such as object groups [ARS98], objects behavior still depends on implementation details, such as reachability from thread stacks pointers. In most systems with strong migration, threads and objects cannot migrate in a single step, and different methods must be used to move them between sites. This is too restrictive, since in most cases, all the threads working on some data are to migrate with that data.

With the Jocaml system, we tried to design a system where creating and manipulating agents would be simple, thanks to a small set of powerful primitives taken from the Join-Calculus [FG96] providing network transparency. To improve expressiveness, it is well integrated in a general-purpose high-level language – Objective-Caml (see `http://ocaml.inria.fr`). The combinaison of a powerful type system and a well-defined process calculus semantics avoids many runtime errors, often hard to debug in distributed environments.

In this paper, we overview the main aspects of Jocaml, seen as a mobile agents system. We first introduce in section 2 the main programming paradigms to create communicating agents, i.e. *channels* and *locations*. Then, in section 3, we describe the language extension to the Objective-Caml language, with some programming examples. Then, in section 4, we discuss the expressiveness of Jocaml through a more complex example of multiple collaborating sites, and in section 5, some implemented applications. Finally, in section 6, we briefly present implementation issues, and the section 7 concludes.

2 Programming model

The Jocaml system attempts to meet the need for a programming language that eases the development of distributed applications. By providing simple but powerful primitives to create communicating agents and migrate them, it allows swift and effective programming of agent based applications.

Our model, taken from the Join-Calculus [FG96], is based on *locations*, which gather both agents and sites in a single abstraction, and *channels*, which are communication links kept during location migrations.

Locations are organized in a hierarchical tree: sites are *toplevel* locations, i.e. locations which are created at the root of the tree, whereas agents are *nested* locations, i.e locations which are created inside other locations. From an implementation point of view, locations may contain active computations – threads –, receivers for channel messages and other nested sub-locations. As a consequence, the migration of a single location moves all of them in a single step (as illustrated in figure 1). Thus, contrary to many concurrent languages, the programmer does not need to manage threads – which are not part of the language semantics – but only locations. However, we will continue to use the thread unit, when only this part of a location is of interest in the discussion.

A dedicated construct, the *definition*, is used to create channels, and associate messages to processes. When a message is received on a channel, a new thread is created to execute the associated process. Thus, channels are asymetric links, on which threads can only send messages which are all received by the channel definition.

Communication links are kept during all migrations, i.e messages sent on channels are transparently forwarded from any agent to the channel definition. Moreover, channels can be transmitted between agents in messages as other standard values. Jocaml allows two kind of channels: basic *asynchronous* channels, and *synchronous* (a la RPC) channels, which can be used like functions.

These abstractions considerably simplify the programmer's task, by hiding distribution details related to agents mobility, such as precise agent positions, communication links, protocols, and data marshaling/unmarshaling. From the developer's point of view, there is no difference between local and remote agents, since local and distant channels are manipulated in the same manner .

The transparency of channels and locations ensures the safety of our migration mecanism: the migration is triggered by a single thread, which calls the `go` primitive, but is transparent for other threads and nested locations included in the migration.

Locations are also used as the basic unit for failure detection. The `fail` guard only returns when the location given as parameter is halted or crashed, following a fail-stop model. Due to the impossibility of accurate failure detection in asynchronous systems, Jocaml currently implements a weakened version of the `fail` guard, suffisant for most applications, and this part is still under active research.

3 Language design

Till now, we have only described the Jocaml programming model, ie the abstractions we want to manipulate in our language. In this section, we describe how these abstractions have been integrated in the Objective-Caml language.

3.1 Objective-Caml

Contrarily to lots of mobile agents, based on scripting languages, the Jocaml system is based on Objective-Caml (Ocaml), a compiled, general purpose, high-

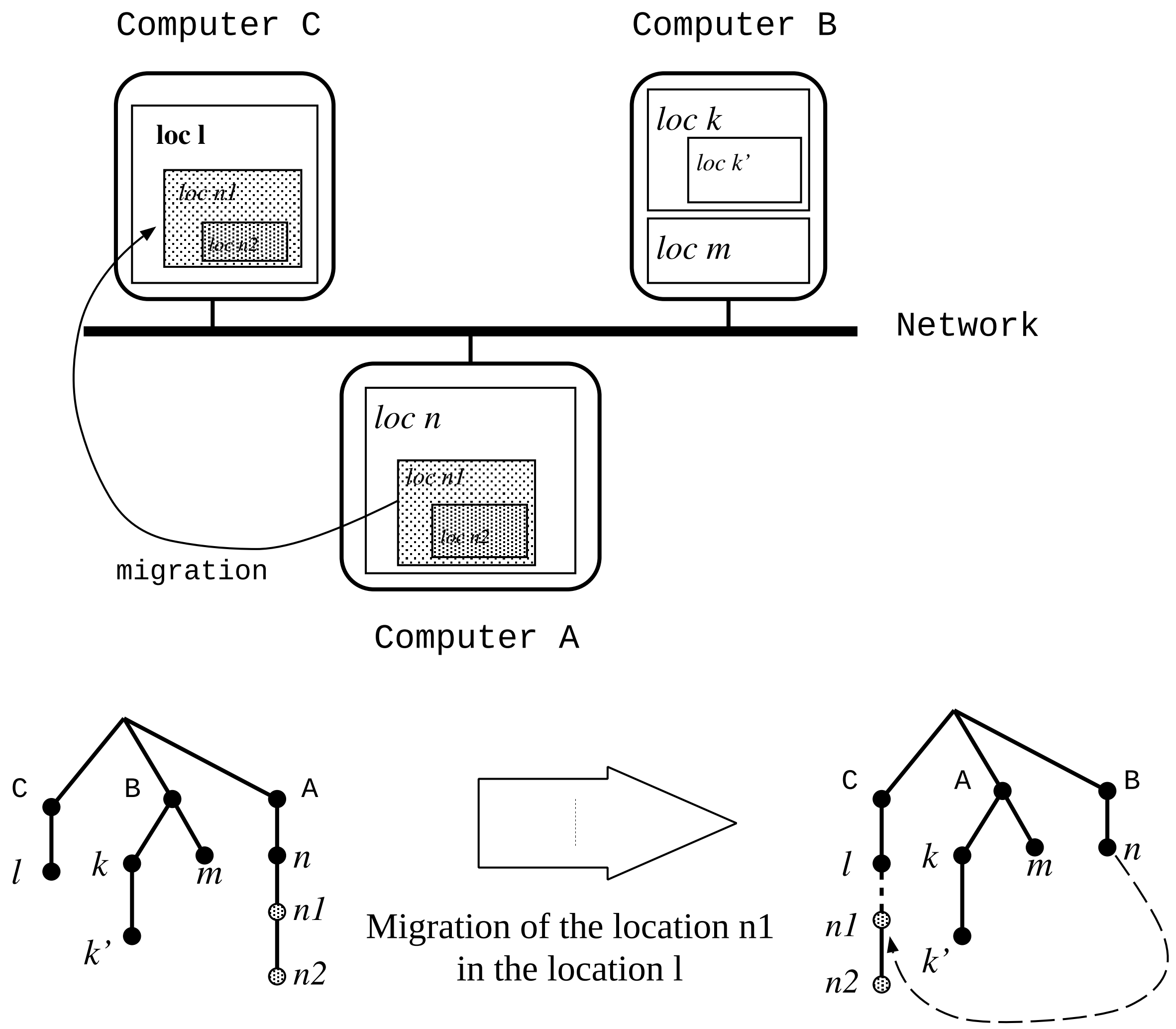

Figure 1: Location tree and migration

level programming language, which combines functional, imperative and object-oriented programming styles. Objective-Caml has also several interesting features for distribution.

First, programs are *statically typed*, and the type system ensures the correctness of the evaluation. This is really important in distributed systems where debugging runtime errors on remote sites is really hard for developers. Moreover, the type system is quite powerful, with parametric polymorphism (extended to the Join-Calculus constructs [FLMR97]), type inference, subtyping for objects and user definable data-types, such as records, exceptions and variants.

The Objective-Caml runtime also provides interesting features, such as the ability to marshal and unmarshal all data types between heterogeneous platforms, an efficient garbage collector, which we have extended for distributed collection, and low-level access to the system, with many Unix system calls, such as sockets management, and so on.

For these capabilities, Objective-Caml has already been used in several complex programs for distributed systems, such as Web browsers (MMM), group communication platforms (Ensemble[Hay97]), or Active Networks (SwitchWare[AHK+98]).

3.2 Programming tutorial

This section gives an overview of the Jocaml language. Throughout this tutorial, we will illustrate the primitives of Jocaml through small examples. We first show the use of channels and processes, and then how to distribute the computation with mobile agents.

3.2.1 Starting communications

In the following examples, we will make different Jocaml programs work together. There are different ways to initiate the communication between these programs. In this paper, we use a simple program, the name-server, bound to a well-known port of a well-known machine, which can either be specified by the environment, by an URL, or found by multicasting the local network.

Jocaml provides two simple functions `Ns.register` and `Ns.lookup`, to respectively register named values in the name-server, and request these values from their names.

Contrary to Java, Objective-Caml does not provide runtime type information, and never performs runtime checks on types, since the static type system already ensures execution correctness. However, in open systems, some dynamic types must be used at application interfaces: in Jocaml, dynamic type-checking is realized by the name-server, to ensure that requested values have the same types as used during the static type-checking at compile time. Consequently, a special keyword, `vartype`, has been added to provide the type information necessary for the `Ns.register` and `Ns.lookup` calls.

Here is a small example, of two programs exchanging a simple string through the name-server facility:

```
(* SERVER *)

let host = Unix.gethostname ();;
Ns.register "server host" host vartype;;


(* CLIENT *)

let host = Ns.lookup "server host" vartype;;
printf "The server is on %s" host;;
```

3.2.2 Processes

Jocaml extends the Objective-Caml language with the notion of *process*. Indeed, Objective-Caml only uses *expressions*, i.e. terms which are computed to a value. On the contrary, the execution of a Jocaml process does not yield any value. Instead, it is used for its side-effects, mainly for communication and synchronization tasks.

The main reason for separating expressions and processes is to force the programmer to pay attention to concurrency problems. Indeed, the basic process, the asynchronous send of message, can hardly be expressed by a sequential expression without hiding important synchronization aspects and misleading the programmer: if expressions were used for asynchronous sends, programmers could believe that a message has been sent, or even received, when the expression following an asynchronous send is executed. However, asynchronous messages are often queued before sending, and might be delivered very late after being forwarded, thus breaking the programmer's reasoning.

Thus, Jocaml processes are syntactically different from Objective-Caml expressions. They are introduced by curly braces `{ ... }` containing both expressions and sub-processes, while `|` denotes parallel composition of processes. The `spawn { P }` construct is used inside Ocaml expressions to create a new thread to execute the process P.

3.2.3 Channels

Jocaml programs communicate and synchronize through messages sent on channels, called *names* in the Join-Calculus terminology. In the two following sections, we focus first on the communication aspect of channels, and then on the synchronization aspect.

Mainly, channels are asymmetric links between one receiver and multiple senders: when messages are sent on a channel by different agents, they are all received by the same agent. This mecanism is not modified if the channel name is exchanged between agents on different machines, nor if the receiver-agent migrates: messages are simply forwarded by the network to the correct agent.

A simple *synchronous* channel definition has the syntax:

$$\boxed{\texttt{let def } \mathit{name}(\text{args}) = P(\text{args})}$$

which creates both the channel name *name*, which can be used to send messages, and a receiver, which will create a new thread in the agent to execute P for each message received, with *args* being the message content. Since Objective-Caml is able to marshal and unmarshal any data type, a message can contain any value, provided the program is correctly typechecked. The process must contain a `reply` statement to specify the returned value, since the message is synchronous (a la RPC). Asynchronous channels are defined by simply suffixing the channel name with a `!` symbol.

In the following example, we define a simple `print` channel, on which other agents can send messages to be printed.

```
(* SERVER *)

let def print s = { print_string s; reply};;
Ns.register "Printer" print vartype;;


(* CLIENT *)

let printer = Ns.lookup "Printer" vartype;;
printer "Hello ";;
printer "I'm the client";;
```

3.2.4 Join-patterns

Channel definitions also have some synchronization power, thanks to the use of *join-patterns*.

Join-patterns are used to express rendez-vous between message reception on several channels . For this, several channels names, separated by `|` symbols, can be specified to trigger a process in a `let def` construct. Moreover, several processes can be specified for different configuration of messages on the channels.

For example, in the following definition, `P` may be triggered when messages are present on both `u` and `v`, and `Q` may be triggered when messages are present on both `u` and `w`. Since `u` appears in both join-patterns, there is a possible synchronization between `P` and `Q`: if there is only one `u` message available, `P` and `Q` threads will not be executed in parallel when messages are available on both `v` and `w`.

$$\texttt{let def}\ \boxed{u(a)\ |\ v(b)} = P(a,b)$$
$$\boxed{\texttt{or}\ u(a)\ |\ w(b)} = Q(a,b)$$

Other examples of synchronizations will follow. Here, we simply refine the previous example of a `print` channel, by requiring that only one `print` process is executed at any time. This is done by adding an asynchronous `lock` channel, whose unique message is consumed by the process, and released when the process is finished. In the client, a rendez-vous is used on three asynchronous messages to decide when the client can be terminated. Indeed, just putting an `exit` statement at the end of the client would be incorrect, since it could be executed before the delivery of the three messages to the server.

```
(* SERVER *)

let def print! (str, cont) | lock! () =
    { print_string str; { cont () | lock () }};;
spawn { lock () };;
Ns.register "Printer" print vartype;;
```

```
(* CLIENT *)

let printer = Ns.lookup "Printer" vartype;;
let def cont1! () | cont2! () | cont3! () = { exit 0; }
spawn { printer ("Hello", cont1) |
        printer ("I'm a client", cont2) |
        printer ("I must do my job", cont3) }
```

3.2.5 Defining agents

In Jocaml, sites, agents and groups of agents are represented by a single abstraction, the *location*. The syntax of a location definition is:

`let loc` *agent* `do` *process*

which creates a new agent, named *agent*, executing the thread *process*.

In the following example, the server defines a synchronous `print` channel, queries the name server for a location registered by the client, and then creates an agent which will migrate to that location. The agent calls twice the `Unix.gethostname` function, once before the migration, to know its current hostname, and once after the migration, to learn its new hostname. This shows the difference between Jocaml synchronous channels and Ocaml functions, since agents can use local resources simply by calling local functions, whereas channels are always bound to the same receiver.

On the other hand, the agent uses twice the `print` channel, once before the migration, and once after the migration. In both cases, the message will be sent to the server, which will print the message on its terminal. This illustrates that agents can migrate without breaking their communication channels, thus keeping complete network transparency.

```
(* SERVER *)

let def print s = print_string s; reply;;
let target = Ns.lookup "client" vartype;;

let loc agent do {
  print (sprintf "I'm on computer %s" (Unix.gethostname ()));
  go target;
  print (sprintf "I'm on computer %s" (Unix.gethostname ()));
  print_string "Hello";
  };;
```

```
(* CLIENT *)

let loc target do {};;
Ns.register "client" target vartype;;
```

Furthermore, agents have to communicate. However, since channels are unidirectional agents can only send messages to others. To receive messages, the syntax of location definition is extended with channels declarations:

`let loc` *agent*
  $\boxed{\texttt{def}\ \textit{channels}}$
`do` { *process* }

which create a new agent composed of threads and channel definitions. All these definitions are lexical scoped, as in Obliq [Car95]. This means that each channel declared in a location has a binding that never change, wherever this agent or the others using this name migrate. This is the transparency mechanism used in Jocaml that eases the programmer task to distribute applications.

In the next example we point out the simplicity and easiness of handling agents in Jocaml. This example makes use of agents in a simple "talk" program. The server waits for two connections, and then, creates and sends a speaker agent to each client. After that, the server can be shutdown without disturbing the discussion of the clients, or can stay available for other pairs of clients.

```
(* SERVER *)

let def create_speaker(l,name) =
(* This channel is used to create a new speaker agent with two channels:
   print: to print messages on agents' display
   speak: allows the client to enter a new sentence *)

  let loc agent def     (* A new speaker agent *)
      print! (s,send) = { printf "%s\n" s; speak(send) }
  and speak! send     = { printf "%s>" name;
                          let s = read_line () in
                          send(s,send) }
   do { go l; } (* Agent migrates to the location l *)

  in (* The "interface" to the agent is return to the server *)

   { reply print,speak };;

(* The connexion part of the server *)
let def talk!(l1,n1) | no_client!() = one_client(l1,n1)
     or talk!(l2,n2) | one_client!(l1,n1) =
          { let  print1,speak1 = create_speaker(l1,n1) in (* creation of *)
            let  print2,speak2 = create_speaker(l2,n2) in (* two agents *)
            speak1(print2) | no_client ()};;
spawn{no_client()};;
Ns.register "talk" talk vartype;;
```

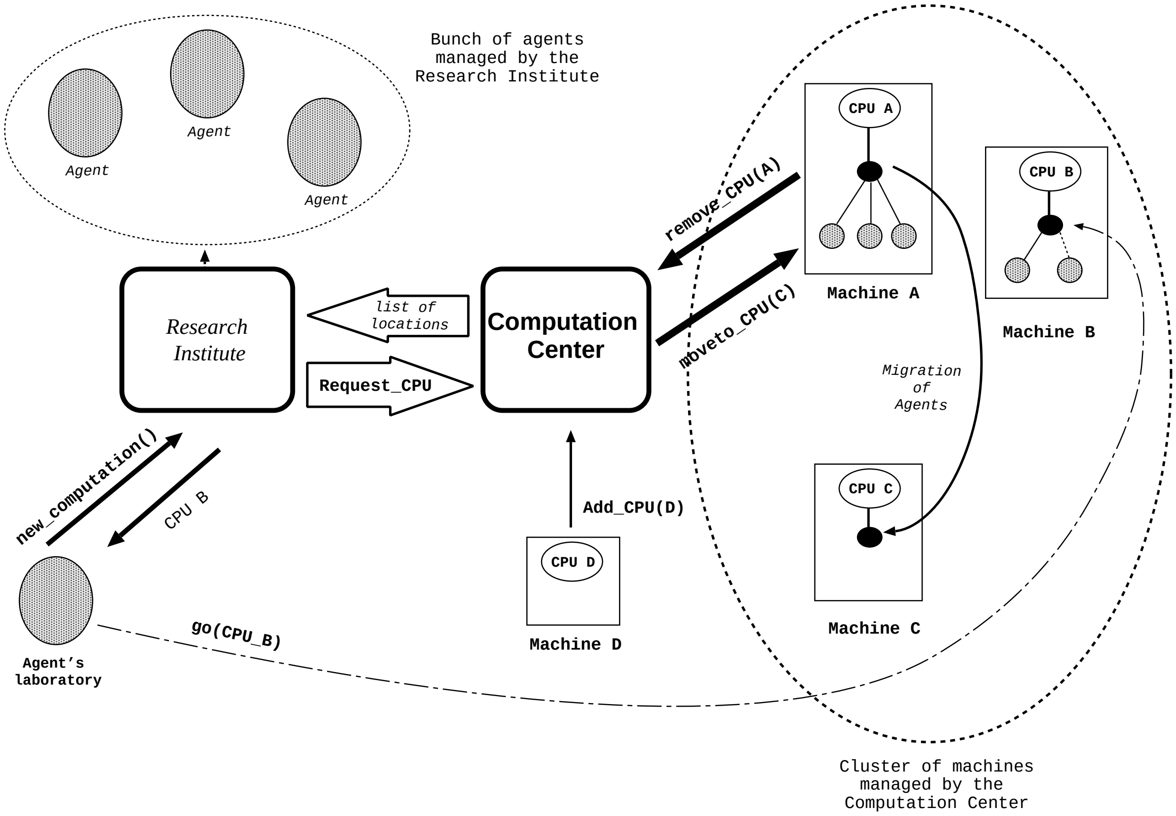

Figure 2: The computation center

```
(* CLIENT *)

let username = getenv "USER";;
let talk = Ns.lookup "talk" vartype ;;
let loc client do { talk(client,username) } ;;
```

4 Case of study

To illustrate the simplicity of programming in Jocaml and its expressiveness, we now present a more complex example. In this example, multiple sites are collaborating: a *computation center* manages a set of super-computers, called *cpus*, for the need of its clients, the *research centers*, which must provide computation power for their *laboratories*.

We can now describe each site requirements: each super-computer can be in two states, *available*, or *out-of-order* for maintenance. Thus, a super-computer will send two messages to the computation center: `add_cpu` when the computer becomes available, and `remove_cpu` when the cpu needs maintenance. `Remove_cpu` returns when the computer can be stopped. Each laboratory has different computations to make. For each computation, it sends a `cpu_for_computation` to its research center, which must return some site where the laboratory will send an agent for the computation. Finally, in this example, each research center is allocated four super-computers at a given time, each one requested with the `request_cpu` message.

This seems quite simple. However, we must add two constraints: First, maintenance of super-computers should be completely transparent for laboratories and research centers. Second, at the beginning, laboratories only know the research center channel name "cpu_for_computation", research centers only know the computation center channel name "request_CPU", and super-computers only know the computation center channel names "add_CPU" and "remove_CPU".

Here is our solution. When a research center requests a computer, the computation center creates an agent – a location –, moves it in one of the available super-computers and return this location name to the research center. When a laboratory requests a computer, its research center returns one of the location given by the computation center. Then, the laboratory sends an agent into that location to make the computation. Now, when a super-computer needs maintenance, the computation center moves its agent in another available – and free – super-computer. As a consequence, all sub-agents sent by laboratories move in a single step to the new super-computer. Once the operation is finished, the old computer can be stopped without affecting the laboratories' computations.

In the commented sources (figure 3), we only give the skeletons of the computation center and the research centers, since laboratories and super-computers are straightforward.

5 Applications

As a proof of concept, Jocaml has already been used in several distributed applications, from simple distributed games and distributed computations of the Mandelbrot fractal to more complex programs, such as a mobile text editor and the first distributed implementation of the Ambient system[CG98].

These examples have shown that the current Jocaml prototype is fairly robust and efficient, and the language quite powerful. For example, our mobile editor, Jefuns, is an almost complete Emacs clone, which is able to migrate between computers (from an Alpha workstation to a simple laptop for example) with its entire state (opened buffers, modifications lists for undo, histories, frames states, etc ...). It has been entirely written in 11000 lines of Objective-Caml (plus 14000 lines for the Xwindow library), to which 200 lines have been added to allow the migration between runtimes. During the migration of the editor agent, about 65 modules of Objective-Caml code are moving, plus all buffers, all live threads and all meta-information data.

Other examples are also interesting agent-based applications, since most of them use *generic clients*, ie clients which do not contain any specialized code at startup, while servers contain both the server and client code. For example, in

```
(* COMPUTATION CENTER *)
(* The GET_CPU channel returns an available cpu. Each ”new_cpu” message contains the location and number of
one available super-computer. *)

let def get_cpu () | new_cpu! (cpu_loc, numero) =
   match cpus.(numero) with      (* "cpus" is the array of CPU's states *)
     Free -> { reply (cpu_loc,numero) } (* The cpu is free.            *)
   | _    -> { reply (get_cpu ()) }     (* Retry if the cpu is already  *)
;;                                      (*   allocated or out-of-order. *)

(* ADD_CPU: sets the cpu state to free, and sends a ”new_cpu” message *)

let def add_cpu (cpu_loc, numero) =
  cpus.(numero) <- Free; new_cpu (cpu_loc, numero) | reply;;

(* REQUEST_CPU: gets a free cpu, creates an agent with a communication channel ”moveto_cpu”, which triggers the
agent migration, moves the agent to the cpu and returns the agent name to the caller. *)

let def request_cpu () | lock! () =
  let (first_cpu, numero) = get_cpu () in
  let loc location def moveto_cpu new_cpu = go new_cpu; reply
  do { go first_cpu }
  in
  cpus.(numero) <- Used moveto_cpu; (* Change cpu state to allocated, *)
                 (* and record the "moveto_cpu" communication channel *)
  { lock () | reply location }

(* REMOVE_CPU: Tests if the cpu is already used, in which case it triggers the migration of the computation center
agent to another available computer, and sets the cpu state to out-of-order. *)

or remove_cpu (numero) | lock! () =
  (match cpus.(numero) with
    Used moveto_cpu ->
       let (other_cpu, other_numero) = get_cpu () in
       moveto_cpu other_cpu;
       cpus.(other_numero) <- Used moveto_cpu
  | _ -> ());
  cpus.(numero) <- OutOfOrder;
  { lock () | reply };;

(* The ”lock” message is used to execute ”request_cpu” and ”remove_cpu” requests in mutual exclusion, by consuming
a ”lock” message before executing their associated process, which releases a new ”lock” message at its end.*)

spawn { lock () };;

(* Register all channel names in the name-server. *)

Ns.register "add_CPU" add_cpu vartype;;
Ns.register "remove_CPU" add_cpu vartype;;
Ns.register "request_CPU" add_cpu vartype;;
```

```
(* RESEARCH INSTITUTE *)

let request_cpu = Ns.lookup "request_CPU" vartype;;

let def cpu_for_computation () | state! (next, cpus) =
  reply cpus.(next) | state (next+1 mod allocated_cpus, cpus)
;;
for i = 1 to allocated_cpus do
  cpus.(i-1) <- request_cpu ();
done;;
spawn { state(0, cpus) };;
Ns.register "cpu_for_computation" cpu_for_computation vartype;;
```

Figure 3: The skeletons of the computation center and the research center.

our four games, the server waits for players connections, then creates an agent which is moved to the player computer with both the client code and already opened communication links(channels) with other players' agents. Then, even if the server crashes, clients can continue to play together.

6 Implementation

The Jocaml system has been implemented by modifying the Objective-Caml distribution version 1.07.

The Objective-Caml compiler has been modified to translate the Join-Calculus syntax into Objective-Caml code. Although the translation is mainly source to source, it is implemented during the typing phase of the compilation, since the pre-translation source is easier to type-check[FLMR97] than the post-translation source. Some optimizations are realized on threads and channels at this level. In particular, definitions are compiled into efficient automatas, which transitions occur when messages are received[LM98].

The bytecode threads library has been used, and modified to extract running threads from the scheduler during agent migrations. Since Jocaml makes heavy use of small threads – although most threads creations are eliminated by the compiler –, some standard optimizations have been realized, on stacks management for example.

Although Objective-Caml is able to marshal and unmarshal almost all values, some types require some work: marshalers for runtime exceptions, file descriptors and objects were added. For the latter ones, the compiler was modified to add runtime information about methods, for the runtime to be able to transparently create proxies for remote objects. A particular care was also taken for self-inflicted methods during migration, due to the unusual way objects and classes are implemented in Objective-Caml.

Finally, a lot of work has been done on remote pointers. The Jocaml implementation uses a variant of the Stub-Scion Pair Chains system[SDP92], a sophisticated, efficient and fault-tolerant system of remote references supporting asynchronous distributed garbage collection. In particular, the distributed garbage collector has been improved to collect distributed cycles. Lots of optimizations have been used, to limit the cost of remote pointers. In particular, the remote pointers on threads, used for replies to synchronous channels (RPC), are saved, and re-used by other threads. Marshaling and unmarshaling of remote pointers (for threads, locations, channels and objects) are avoided thanks to pair-to-pair caches. Both mecanisms are very efficient in Jocaml, since most applications create only a small number of permanent remote references, which are used for synchronous calls by many small threads.

7 Conclusion and future work

We have presented a new mobile agents system, based on a general-purpose high-level language, Objective-Caml. Contrarily to other systems based on existing languages, our system achieves complete network transparency, while extending the language with new simple but powerful abstractions, ie locations and definitions. Moreover, these abstractions, taken from the Join-Calculus – a distributed calculus of processes – have a strong semantical basis. Due to size limitations, we have omitted several interesting features in this paper, such as transparent invocation of remote methods and strong objects migration.

Our system has been successfully implemented, and supports strong migration of threads, transparent forwarding of messages, remote method invocations, and distributed garbage collection. Different applications have already been written for our system, such as a mobile editor and some distributed games.

The current system (freely available at `http://pauillac.inria.fr/jocaml` already supports site failures, but we plan to improve failure detection by providing different levels of view consistency. We are also working on security issues, which have been neglected in the current implementation.

A Bibliography

References

[AHK+98] D. Scott Alexander, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Marianne Shaw, Jonathan T. Moore, Carl A. Gunter, Trevor Jim, Scott M. Nettles, and Jonathan M. Smith. The switchware active network implementation. In *The ML Workshop*, International Conference on Functional Programming (ICFP), September 1998. ACM SIGPLAN.

[ARS98] Anurag Acharya, M. Ranaganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems*. J. Vitek and C. Tschudin, Springer Verlag (LNCS), 1998.

[Car95] Luca Cardelli. A language with distributed scope. In *22nd annual ACM Symposium on Principles of Programming Languages*, January 1995.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures*, volume 1378, pages 140–155. Springer (LNCS), 1998.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 21-24 1996. ACM.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ml for the join-calculus. In *Proc. of the 1997 8th International Conference on Concurrency Theory*. Springer-Verlag, 1997.

[Gra95] Robert S. Gray. Agent tcl: A transportable agent system. In *CIKM Workshop on Intelligent Information Agents*, Baltimore, Maryland, dec 1995. Fourth International Conference on Information and Knowledge Management (CIKM 95).

[Hay97] Mark Hayden. Distributed communication in ML. Technical Report TR97-1652, Cornell University, Computer Science, November 11, 1997.

[JRS95] Dag Johansen, Robbert Van Renesse, and Fred B. Schneider. An introduction to the tacoma distributed system version 1.0. Technical Report 95-23, University of TromsO, Norway, June 1995.

[LM98] Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. In *CONCUR'98 Workshop on High-Level Concurrent Languages (HLCL'98)*, Nice (France), 1998.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust distributed references and acyclic garbage collection. In *Proc. 11th Symp. PODC*, pages 135–146. ACM Press, August 1992.

[Whi96] J.E. White. Telescript technology: Mobile agents. In *Software Agents*. J. Bradshaw, editor, AAAI Press/MIT Press, 1996.