Transport Primitives for Functional Agents *

Víctor M. Gulías, Juan J. Quintela, José L. Freire
{gulias,quintela,freire}@dc.fi.udc.es
LFCIA, Department of Computer Science
University of Coruña, SPAIN

Abstract

In this paper, a basic set of transport primitives is shown. Such primitives constitute a basis on top of which a distributed functional framework can be developed. The distributed system uses asynchronous message passing to communicate a cluster of virtual processors, typically a computer network. The primitives are described in a module signature using the advanced module facilities provided by the functional language OBJECTIVE CAML. Two implementations of such signature are shown, using TCP/IP sockets and a library that implements the MPI standard, in order to show how modules can be used to isolate communication-specific details from the rest of the system.

Keywords: Functional Programming, Distributed Computing, Communication Networks, Module System, Runtime System.

1 Introduction

Much of the elegance of functional languages stems from their abstract semantics, devoid of operational or machine-dependent detail. It is this abstract nature that makes functional languages so attractive to software developers, who generally wish to stay far from machine-specific details. However, functional languages demand much more resources (time, space) than imperative ones. In order to improve the execution time, a great deal of effort has to be made to improve efficiency in such languages. In the last years, our two cents on this matter has been the research of distributed architectures in which functional programs can be executed. In order to develop a complete concurrent functional system running on top of a distributed system, such as a cluster of workstations, a small and consistent set of primitives to transport data among nodes (*virtual processors*, or processors for short) should be provided.

In this paper, a module signature describing such primitives is shown. This interface constitutes the basis on top of which a distributed functional framework, an evolution of [5, 4], can be developed. The distributed system uses asynchronous message passing to communicate a cluster of processors, such as a computer network or a distributed-memory multiprocessor. The primitives are described in a module signature using the advanced module facilities provided by the functional language OBJECTIVE CAML [8, 7]. Two implementations of such signature are shown, using TCP/IP sockets [12] and a library that implements the MPI standard, CHIMP/MPI [9], in order to show how modules can be used to isolate communication-specific details from the rest of the system.

*Supported by the *Xunta de Galicia* XUGA10504B96 and XUGA10505B96

2 Previous Work

This work is an evolution of [3] in which a functional distributed system is designed to speed up program execution by spawning different tasks on a network of computers using computation servers. The distribution is carried out explicitly by indicating which remote services (functions) will be available in the remote servers. The function signature of each service is used to determine the communication protocol (encoding of arguments and results) and to generate automatically the stub code necessary to build the servers. A client process spawns remotely the computation of an expression using one of those services, breaking down the synchronism of the evaluation sequence of a strict language (evaluation of arguments, and then evaluation of the function body with those results). The main inconvenience was that functions were not first-class citizens, so many powerful facilities of functional languages (partial application, abstraction, higher-order functions) should be abolished from programmer's mind. In [5], a closure shipping algorithm is shown in order to allow higher-order interaction among agents. In addition, this closure shipping algorithm preserves the sharing of data structures, reducing communicating costs and allowing cyclic data structure transmission. The monolithic approach taken in the aforementioned works was replaced by a naive decomposition of the prototype [4] using the advanced module system introduced in [7].

3 Transport Primitives

Several models of concurrent computation use communication independent computational agents. Communication provides a mechanism by which each agent retains the integrity of information within it. There are two possible assumptions about the nature of communication between independent computational elements; communication can be considered to be either (a) *synchronous*, where both the *sender* and the *receiver* of a communication must be ready to communicate before a communication can be sent; or, (b) *asynchronous*, where the *receiver* does not have to be ready to accept a communication when the *sender* sends it.

Hoare's CSP [6] and Milner's CCS [10] assume synchronous communication while the *actor model* [1] and *dataflow* [2] do not. Even though we are implementing a system with concurrent functional agents, this system is built on top of a distributed architecture. Hence, this implies that before a sender can know that the receiver is ready to accept a communication, a communication must be sent to the receiver, and viceversa. Thus one may conclude that any model of synchronous communication is built on asynchronous communication, and that is why our transport primitives are asynchronous in nature.

3.1 Value Shipping

Implementing a distributed layer implies that encoding/decoding of data as well as its transmision along the communication network are needed. The encoding of data is a refinement of the one exposed in [3] in which a traversal of the internal representation of any value is carried out to build a flatten homomorphic representation using a stream of bytes stored as a OBJECTIVE CAML string. In [4], the type system is used to assure the correctness of data transmission among agents. However, at this level a protocol should provide safe data conversions from $\alpha \rightarrow package$, and back to $package \rightarrow \alpha$ to

the functional agents. A module that *packs* data before being shipped to other location should match the following signature:

```
type package = string
module type PACK =
sig
  val pack   : 'a -> package
  val unpack : package -> 'a
end
```

The pack algorithm should provide higher-order packing, which means that closures can be sent freely from one agent to others like any other value. In our implementation, `Pack : PACK`, this property holds, although we are restricted, at the moment, to the use of the same object file in each side, if higher-order interaction is required. This module also preserves sharing, which reduces communication cost as well as allows the transmission of cyclic data structures such as the internal representation used in OBJECTIVE CAML for recursive functions.

3.2 The Distributed Virtual Machine

The *distributed virtual machine* (DVM) is composed of a set of processors executing functional agents. The configuration and startup of the machine are performed by external means dependent of the sublying communication implementation. For example, in an implementation of MPI running on a network of computers, a script can be used to declare in which host a functional agent runs.

Processors are organized in *groups*, and a special processor is chosen as the *processor leader* for that group. At boot time, every processor only knows the "address" of itself as well as the leader's one. More knowledge about the topology of the world should be obtained by a protocol among all the agents. The simplistic approach consists of having a unique group and, therefore, one unique leader. However, more complex topologies may be instantiated like a cluster topology.

3.3 Module Signature

The communication module should provide the following services:

- A *processor* abstract data type. The processor must be kept abstract to avoid any knowledge about the identification used internally by the communication module. Given that we cannot access to the internal representation, a predicate `equal` should be added to compare the equality of two processors.
- Two methods are needed to get some information from the DVM: `self : unit -> processor` to identify our processor handler, and `leader : unit -> processor` to identify the processor which leads the group in which the current processor is located. `unit` is a type with one only construct `()`. It is used for functions what are important for their side-effects.
- Initialization and finalization methods to deal with module specific implementation, such as resource allocation (`init : unit -> unit`, and `exit : unit -> unit`).

- Two methods to deal with asynchronous reliable communication between processors (`send : processor -> package -> unit` and `recv : unit -> processor * package`). The communication is asynchronous in the sense that there is no rendezvous point between sender and receiver. The communication is reliable in the sense that no transmision error may occur.

To assure that an implementation of a module matches the proposed `COMMUNICATION` module, the following signature is given:

```
module type COMMUNICATION =
  sig

    type processor

    val self       : unit -> processor
    val leader     : unit -> processor
    val equal      : processor -> processor -> bool

    val init       : unit -> unit
    val exit       : unit -> unit
    val recv       : unit -> processor * package
    val send       : processor -> package -> unit

  end
```

It is conceivable to define the transmision primitives using the polymorphic data type `'a` instead of the encoded `package` abstract datatype, delegating the packing of data structures behind this layer. Nevertheless, the proposed solution using the `package` ADT is more flexible because it allows us to design more tuned algorithms such as efficient multicast communication without packing data more than once. It is quite simple to extend our module to support the polymorphic ones. The following example shows how to implement this alternative implementation of `send` and `recv`, giving concrete package and communication modules:

```
module MyPack: PACK = Pack
module MyComm: COMMUNICATION = Mpi
...
  (* alt_send : processor -> 'a -> unit *)

  let alt_send proc msg = MyComm.send proc (MyPack.pack msg)

  (* alt_recv : unit -> processor * 'a *)

  let alt_recv ()       = let (proc,pkg) = MyComm.recv ()
                          in (proc,MyPack.unpack pkg)
...
```

Parametrized modules (*functors*) can be used to generalize the above construction. In the following example, a new module (`Alt_Communication`) is parametrized by two modules with signature `PACK` and `COMMUNICATION`, respectively:

```
module Alt_Communication =
    functor (MyPack: PACK) ->
       functor (MyComm: COMMUNICATION) ->
struct
...
 let send proc msg = MyComm.send proc (MyPack.pack msg)

 let recv ()       = let (proc,pkg) = MyComm.recv ()
                     in (proc,MyPack.unpack pkg)
...
end
```

3.4 Additional Advantages of Functors

The use of functors allows us to compose programs easily. As an example of that argument, take a look at the following. A profiler of the events generated by the `COMMUNICATION` module is going to be designed. Our only interest is to keep track of the moment when communication takes place.

```
module type PROFILER =
sig
  type event = Send of package
             | Recv of package
  val event : event -> unit
end
```

A profiled communication module is a functor that takes a profiler, the concrete action performed when an event occurs, and the communication module to be profiled.

```
module ProfCommunication =
    functor (MyProf : PROFILER) ->
        functor (MyComm : COMMUNICATION) ->
struct
  ...
  let send proc msg = MyProf.event (MyProf.Send msg);
                      MyComm.send proc msg

  let recv () = let (proc,msg) = MyComm.recv ()
                in MyProf.event (MyProf.Recv msg);
                   (proc,msg)
  ...
end
```

Many profilers may be designed, for example one to measure the mean time between transmissions, another to measure the number of packages received, the total amount of bytes received or sent, and so on. Many communication modules can also be implemented. Combining profilers and modules using the `ProfCommunication` functor allows us to cover a great deal of combinations to tune a system.

3.5 An Example: Master/Slave Architecture

As an example of how more complex protocols can be defined on top of the given primitives, a very simple master/slave architecture is going to be shown. In order to simplify even more the example, the `Alt_communication` module is used (section 3.3).

```
module MyComm = Alt_Communication (Pack) (Socket)
open MyComm
```

Master and slave interaction is achieved by exchanging messages: (a) a slave may communicate with the server to inform that it is ready to collaborate, (`SlaveReady`); (b) a master may inform a slave that it is no longer useful (`Kill`); (c) a master may send a request, a pair (*function, argument*), to the slave (`Request`); (d) a slave may respond to a previous request from the master (`Response`). The `message` datatype represents such possible messages.

```
type ('a,'b) message = SlaveReady
                     | Kill
                     | Request of ('a -> 'b, 'a)
                     | Response of 'b
```

The slave agent is quite simple. First of all, it sends a `SlaveReady` message to the leader processor (*its master agent*). Then, it loops until a `Kill` message is communicated. If a `Request` is received, the slave calculates the result and sends the `Response` back to the master.

```
let slave () =
  let rec loop () =
    match recv() with
       (_, Request (f,x)) -> send (leader()) (Response (f x));
                             loop()
     | (_, Kill)          -> ()
  in send (leader()) SlaveReady
```

The master receives as argument a list of pending tasks (pairs of functions and its arguments). As soon as a slave becomes ready (because it joins the party by using a `SlaveReady` message or when it finishes a pending task), the master sends it the first request not yet processed. If there is no such a task, a `Kill` message is delivered. All the results collected from the slaves are returned in a list.

```
let master reqs =
  let rec loop = function
      (0,[])    -> []
    | (n,[])    -> match recv() with
                       (slave, Response x) -> send slave Kill;
                                              x :: loop (n-1,[])
                     | (slave, SlaveReady) -> send slave Kill;
                                              loop (n,[])
    | (n,r::rs) -> match recv () with
                       (slave, Response x) -> send slave (Request r);
                                              x :: loop (n,rs)
                     | (slave, SlaveReady) -> send slave (Request r);
                                              loop (n+1,rs)
  in loop (0,reqs)
```

Finally, the main block of code initiates the DVM, chooses the apropriate role for the processor (the master is the leader processor, the slaves are the rest of the processors). The list of results is processed by the agent used the argument `present_results`, irrelevant for the example.

```
let main present_results requests =
  init ();
  if equal (self()) (leader())
    then present_results (master requests)
    else slave ();
  exit()
```

4 Implementation Issues

Ij the following subsections, we are going to describe two different implementations of the communication module (`Mpi` and `Socket`). Both implementations match the signature commented in section 3.3, so they can be changed freely in a given program.

4.1 The `Mpi`: `COMMUNICATION` Module

The implementation of `COMMUNICATION` using the standard message passing library has been almost straightforward. CHIMP/MPI [9] has been the concrete library used in our implementation, although it should be easily ported to other MPI platforms.

The `init`, `exit`, `send`, and `recv` primitives maps directly to MPI library functions, with the corresponding wrappers to perform type conversion between C and OBJECTIVE CAML worlds. The positive integer used by MPI to identify a host is hidden in the `processor` ADT. In the most naive approach, there is only one group and MPI host 0 becomes the leader processor. However, complex architectures involving several clusters of computers should be defined externally at (virtual) boot time.

4.2 The `Socket`: `COMMUNICATION` Module

The implementation of `COMMUNICATION` using TCP/IP sockets, covered in [11], requires the design of a *reliable datagram service*. This service, whose algorithm is similar to the one exposed in [12], uses UDP datagrams for the transmission of messages. *Stream sockets* were not selected because of the cost of establishing a conexion for a single message or maintaining a point-to-point communication with every possible destination. The concurrent features of OBJECTIVE CAML has been used to interweave the execution of the functional program with the management of the communicating ports.

4.3 Benchmarks

The execution of the master/slave example presented in section 3.5 is tested using both implementations of the `COMMUNICATION` module. Note that the only modification in the code is the change of the module used for communication: `module MyComm : COMMUNICATION = Mpi` or `module MyComm : COMMUNICATION = Socket`. The master calculates 16 times $fib(36)$, being fib the well-known fibonacci function. As a reference, a sequential OBJECTIVE CAML program (bytecode compiler, as used in the distributed

benchmarks) takes 18m 21s. The tests are performed on a network of SPARC 4 with 16Mb of memory, running with Solaris 2.5.1. Every agent runs in a different physical host, including the master process. In table 4.3, we show the execution time varying the number of slaves.

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves
CHIMP/MPI	33:03	16:39	8:10	4:55	3:09
Sockets	18:34	9:19	4:41	2:23	1:28

Table 1: Results for 16 $fib(36)$

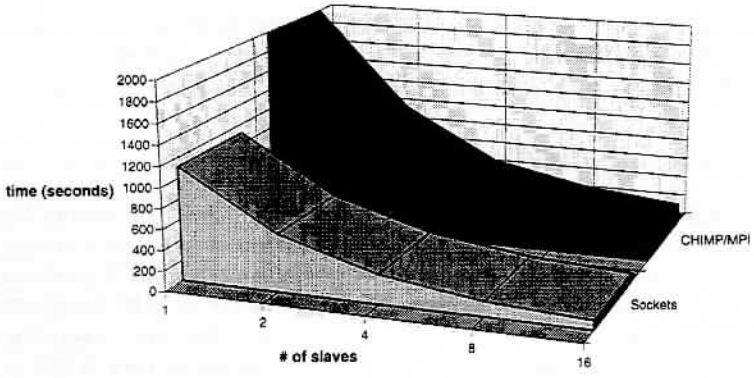

Figure 1: MPI vs. sockets

The CHIMP/MPI is quite inefficient, taking 30m 33s to execute the test with one master and one slave in the same machine. Perhaps, we should move to a more efficient implementation of MPI.

5 Conclusions and Further Work

A set of primitives to perform transport of data between processors, described by an OBJECTIVE CAML signature, has been presented. Such primitives are the basis for distributed systems on which functional programs can be carried out. We also have pointed out the importance of using modules and functors as powerful tools for composing pieces of the distributed system. As an example, a small protocol, a master/slave architecture, has been developed and tested with two different implementations of the communication module. It is well know that results depend on the amount of work done by each task. Thus, layers on top of the communication module should implement scheduling strategies to manage better a great deal of system information and include more information to guide the distribution process such as annotations or previous executions profiles.

Currently, we are extending OBJECTIVE CAML concurrency features to spawn threads on a network of computers. Transparent migration of threads as well as remote communi-

cation among agents seems to be quite interesting future goals. However, some problems should be solved. Despite the fact that our pack algorithm allows higher-order communication among nodes, currently different functional programs cannot be merged in a distributed system. That means that the behaviour of all agents must be defined in the same object file. This difficulty must be overcome in order to create a distributed system, whose nodes may have specific features needed by other agents.

References

[1] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts 02142, 1990.

[2] Arvind and D. E. Culler. Dataflow architectures. In *Annual Reviews in Computer Science 1986*, pages 225–253. Annual Reviews,, Palo Alto, CA, 1986.

[3] J.L. Freire, V.M. Gulías, and B.B. Fraguela. Extending Caml Light to Perform Distributed Computations. In *Joint Conference on Declarative Programming (GULP-PRODE'95)*, Salerno, Italy, September 1995.

[4] V.M. Gulías, J.J. Quintela, and J.L. Freire. Distributed Computing using Objective Caml. In *8th International Workshop on Implementation of Functional Languages IFL'96*, Bonn, Germany, September 1996.

[5] V.M. Gulías, J.J. Quintela, and J.L. Freire. Towards Higher-Order Distribution in Functional Languages. In *APPIA-GULP-PRODE'96*, pages 219–232, San Sebastian, SPAIN, June 1996.

[6] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[7] X. Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. Rapport de recherche 2721, INRIA, November 1995.

[8] X. Leroy. *The Objective Caml System, release 1.05.* INRIA, 1997.

[9] J. Mills, L. Clarke, and A. Trew. CHIMP Concepts and Development. Technical Report EPCC-TR94-14, The University of Edinburgh, March 1994.

[10] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, Vol. 92. Springer-Verlag, Berlin/New York, 1980.

[11] J.J. Quintela. Concurrencia Distribuida. Master's thesis, Faculty of Informatics, University of Coruña, June 1.997. (to appear).

[12] R. Stevens. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1993.