Distribution and Mobility with Lexical Scoping in Process Calculi

Vasco T. Vasconcelos [a,1] Luís Lopes [b,2] Fernando Silva [b,2]

[a] *Department of Computer Science, University of Lisbon, Portugal*

[b] *Department of Computer Science, University of Porto, Portugal*

Abstract

We propose a simple model of distribution for mobile processes, independent of the underlying calculus. Conventional processes compute within sites; inter-site computation is achieved by message sending and object migration, both obeying a lexical scope. We focus on the semantics of networks, on programming practice, and on physical realization with current technology.

1 Introduction

Milner, Parrow, and Walker's π-calculus [12] has provided a formal framework for most of the research on concurrent, communication based systems. Several forms and extensions of the asynchronous π-calculus [9] have since been proposed to provide for more direct programming styles, and to improve efficiency and expressiveness [3,6,18]. The π-calculus has also been used as a basis to reason about distributed computations. Introducing distribution, code mobility, and failure detection and recovery into π-computations is a fast growing research field, with immediate applications in mobile computing, *web* languages, cryptography, to name a few.

We propose a simple model of distribution for mobile processes. The following major constraints guided its design:

(i) the model must be a simple extension of the calculi we have today;

(ii) must be independent of the base calculus chosen;

(iii) must meet realistic expectations of current distributed systems;

(iv) must be efficiently implementable in current hardware.

[1] Partially supported by Project Escola PRAXIS/2/2.1/MAT/46/94.

[2] Partially supported by Project Dolphin PRAXIS 2/2.1/TIT/1577/95.

No distributed system can be conceived without the notion of *site* (or location) where conventional (name-passing, in this case) computations take place. So we have sites, and we have site identifiers, distinct from the usual names. Our processes are *network aware*: names can be local or remote; the distinction is explicit in the syntax. Local names are those of the base calculus; remote names are pairs site-name, called *located names*.

Sites abstract nodes in a network. They are composed of *located processes* — processes paired with site identifiers — denoting the execution of the process in the site, which is similar to most proposals to date [1,7,8,14,16]. Located processes can be put to run in parallel. Furthermore, since name-passing calculi are capable of extruding the scope of a (local) name, our networks are equipped with a located name restriction operation. In summary, networks are located processes equipped with a composition and a restriction operator, yielding a *flat organization of sites* quite close to Distributed-π [14], and in contrast with the tree structure of Mobile Join [7], and the nested structure of Ambients [5].

The model encompasses two levels: processes and networks (cf. [1,8,14]). Local computations happen at located processes, as prescribed by the semantics of the base calculus. What do we want for remote computations? For the moment we only allow the communication of prefixed processes between different sites. These include remote message invocation (messages in the asynchronous π-calculus [9], Join [6], or TyCO [18]), the migration of procedures (input-prefix processes in π-calculus and resources in the Blue calculus [3], replicated or not), the migration of objects (in TyCO), and the migration of messages with continuations (output-prefix processes in the π-calculus). The transport of prefixed processes is deterministic, point to point, and asynchronous; synchronization only happens locally, at reduction time.

We adhere to the *lexical scoping in a distributed context* of Obliq [4]. The free names of any "piece of code" transmitted over the network are bound to the original location. Network transmission implies the translation of the free names in the code in order to reflect the new site where the code is to be executed.

An important design decision related to points (iii) and (iv) above is the incapacity of the model to create remote names and the inability to spawn processes at remote sites, thus providing for site protection against arbitrary uploads. Section 3.4 shows how this can be circumvented with the collaboration of the remote site.

As a first proposal, our site identifiers are not first class objects: they cannot be sent in messages; we deliberately eschew the possibility of checking whether a site is alive and of killing a site [1,7,14], of checking whether two remote names reside at the same site [16], of comparing site identifiers, of dynamically constructing a located name given a name and a site identifier.

The outline of the paper is as follows. The next section introduces the network model, its syntax and semantics; section 3 presents several program-

ming examples that attest the flexibility of our proposal; section 4 discusses implementation; and section 5 includes a comparison with related work. The last section presents ideas for future development.

2 The Model

The ideas presented in the previous section can be embodied in any name-passing calculus. The model is two level: on the first level we have the *processes* in the base calculus; on the second level we build *networks*.

We have said that our model is independent of the base calculus. There are however a few conditions that it must fulfill:

(i) the base calculus may incorporate values in general, and should provide for names in particular. For the purpose of this exposition, we let a range over names, and v over values.

(ii) it should allow to create a new name visible only in a given process, obeying the lexical scoping convention. We write $\nu x P$, as usual.

(iii) it should have processes prefixed on some name. Examples are output and input prefixes ($\overline{a}vP$, $a(x)P$, $!a(x).P$) in the π-calculus [12], messages and objects ($a \lhd m$, $a \rhd M$) in TyCO [18], requests for session initiation (**accept** $a(k)$ **in** P, **request** $a(k)$ **in** P) in Structured Communication-Based Programming [10], and names and resources (a, $a \Leftarrow P$, $a = P$) in the Blue calculus [3]. All the above examples are prefixed at name a; for the purpose of this exposition we write them aC.

(iv) it should have a parallel composition operator and the corresponding neutral element. We write them $|$ and $\mathbf{0}$, respectively.

(v) it should incorporate a notion of substitution of names by values in a process, avoiding the capture of the names substituted. If P is a process and σ a total function from names to values, we denote by $P\sigma$ the process resulting from applying σ to P.

We find these requirements mild; most calculus to date [3,9,10,12,18] fulfill these constraints. A possible exception is the Join calculus [7] and item (iii) above.

We start by introducing a new class of identifiers, *sites*, distinct from names or any other class of identifiers the base calculus may include. *Located names* are site-name pairs. We let s range over sites, and e over located names. A name a located at site s is denoted by $s{\cdot}a$. We then allow located names to occur in any position in the base calculus where (non-binding occurrences of) names can. The calculus thus obtained constitutes the *first level* of the model. Since site identifiers are introduced anew, there must be no provision in the base calculus for binding located names. As such, at this level, a located name behaves as any other constant in the base calculus.

The *second level* is composed of site-process pairs called *located processes*

(denoted $s : P$), composed via conventional parallel ($N \parallel N$) and (located name) restriction (νeN) operators. The set of networks is given by the following grammar.

$$N \quad ::= \quad s : P \quad | \quad N \parallel N \quad | \quad \nu eN \quad | \quad \mathbf{0}$$

The bindings in networks are as expected: a located name e occurs *free* in a network if e is not in the scope of a νeN; otherwise e occurs *bound*. The set of free located names in a network N, notation $\mathrm{fn}(N)$, is defined accordingly.

Structural congruence allows us to abstract from the static structure of networks; it is defined as the least relation closed over composition and restriction, that satisfies the monoid laws for composition, as well as the following rules taken from Hennessy-Riely [8]. [3]

$$\begin{array}{rl}
(\textsc{Nil}) & s : \mathbf{0} \equiv \mathbf{0} \\
(\textsc{Split}) & s : P_1 \parallel s : P_2 \equiv s : (P_1 \mid P_2) \\
(\textsc{New}) & s : \nu aP \equiv \nu s{\cdot}a(s : P) \\
(\textsc{Extr}) & N_1 \parallel \nu eN_2 \equiv \nu e(N_1 \parallel N_2) \quad \text{if } e \notin \mathrm{fn}(N_1)
\end{array}$$

Rule Nil garbage collects terminated located processes. When used from left to right, the rule Split gathers processes under the same location, allowing reduction to happen; the right to left usage is for isolating prefixed processes to be transported over the network (see rule Move in the reduction relation below). The remaining rules allow the scope of a name local to a process to extrude (rule New) and encompass a network with several located processes (rule Extr).

Non-located names in processes are implicitly located at the site the process occurs at: a name a occurring in a network $s : P$ is implicitly located at site s. When sending names over the network, the implicit locations of names need to be preserved, if we are to abide by the lexical scoping convention. As such, a name a moving from site r to any other site must become $r{\cdot}a$. Similarly a located name $s{\cdot}a$ arriving at site s may drop its explicit location. The remaining names and values need no translation. A *translation of values* from site r to site s is a total function σ_{rs} defined as follows: [4]

$$\sigma_{rs}(a) \stackrel{\text{def}}{=} r{\cdot}a \qquad \sigma_{rs}(s{\cdot}a) \stackrel{\text{def}}{=} a \qquad \sigma_{rs}(v) \stackrel{\text{def}}{=} v$$

Processes prefixed at located names play a crucial role in the model, by moving towards the location of the located name: a process $s{\cdot}aC$ is meant to move to site s. If aC is a message (say $\overline{a}v$ in the asynchronous π-calculus), then $s{\cdot}aC$ denotes a remote message send; if on the other hand aC includes "some code" (say $a(x)P$ in the π-calculus), then $s{\cdot}aC$ denotes a process migration

[3] Rules Nil, Split, and Extr are present in Sewell et al. [16] as well.
[4] The last rule should be applied last.

operation. We thus see that conceptually there is not much difference between a remote message send and a process migration; in section 4 we show that from an implementation point of view the difference is not abysmal either.

The reduction relation for networks is given by the following axiom and rule, plus the familiar rules for composition, restriction, and structural congruence which we omit.

$$(\textsc{Move}) \quad r : s{\cdot}aC \to s : a(C\sigma_{rs}) \qquad\qquad (\textsc{Local}) \quad \frac{P \to Q}{s : P \to s : Q}$$

If prefix $s{\cdot}aC$ is located at site r, then, in order to keep the lexical scope of names, the free names in C must translated according to $C\sigma_{rs}$. So, when sending $s{\cdot}aC$ from r to s we actually transmit $(s{\cdot}aC)\sigma_{rs} = aC\sigma_{rs}$. This is the essence of the axiom Move. Rule Local allows processes in sites to evolve locally.

As an example let us try a remote procedure call in the π-calculus. The client at site s invokes the procedure p at site r with a local argument v, waits for the reply and continues with P. The procedure accepts a request and answers a local name u (somewhere in the body Q of the procedure).

$$\begin{array}{lr}
s : \nu a(\overline{r{\cdot}p}[va] \mid a(y).P) \;\parallel\; r : p(xc).Q \equiv & (\textsc{New,Extr}) \\
\nu s{\cdot}a(s : \overline{r{\cdot}p}[va] \;\parallel\; s : a(y).P \;\parallel\; r : p(xc).Q) \to & (\textsc{Move}) \\
\nu s{\cdot}a(r : \overline{p}[s{\cdot}v\ s{\cdot}a] \;\parallel\; s : a(y).P \;\parallel\; r : p(xc).Q)) \equiv & (\textsc{Split}) \\
\nu s{\cdot}a(s : a(y).P \;\parallel\; r : (\overline{p}[s{\cdot}v\ s{\cdot}a] \mid p(xc).Q)) \to & (\textsc{Local}) \\
\nu s{\cdot}a(s : a(y).P \;\parallel\; r : Q[s{\cdot}v\ s{\cdot}a/xc]) \to^* & \\
\nu s{\cdot}a(s : a(y).P \;\parallel\; r : \overline{s{\cdot}a}[u]) \to\to & (\textsc{Move,Split,Local}) \\
\nu s{\cdot}a(s : P[r{\cdot}u/y]) \equiv s : \nu aP[r{\cdot}u/y] & (\textsc{New})
\end{array}$$

We thus see that a remote communication involves two reduction steps: one to get the message/object to the target site and the other to consume the message/object at the target (cf. [7]); the former is an asynchronous operation, the latter requires a rendez-vous. This reflects actual implementations.

3 Programming

Pick your favorite name-passing programming language, and simply add two new declarations.

export name **in** process
import name **from** site **in** process

There is no need to change the syntax of the base language whatsoever. In particular we never write located names explicitly. The translation into the

base calculus extended with located names is quite simple.

$$[\![\textbf{export } a \textbf{ in } P]\!] \stackrel{\text{def}}{=} [\![P]\!]$$
$$[\![\textbf{import } a \textbf{ from } s \textbf{ in } P]\!] \stackrel{\text{def}}{=} [\![P[s.a/a]]\!]$$

We thus see that the **export** declaration is really unnecessary. Since programs are to be closed, we could take the view that every free name in a program is to be exported. From a programming point of view we however feel that the dual **import**/**export** declarations impose a more disciplined programming style, avoiding, for example, the automatic exporting of names that the programmer forgot to protect with a **new**.

The remainder of this section is devoted to the presentation of several programming examples that attest the flexibility of the model. The new ideas are embodied in our favorite name-passing programming languages: TyCO [17], and Structured Communication-Based Programming [10].

3.1 Java applet server

Our first example illustrates code transmission over the network. The idea is from Fournet et al. [7], but we have taken advantage of objects in TyCO to allow for the downloading of different applets.

In order to set the context for the example we briefly review TyCO [18,17]. TyCO is a name-passing calculus in the line of the asynchronous π-calculus [9] that incorporates, in place of (unlabeled) messages and receptors ($\overline{a}v$, $a(x).P$), labeled messages and objects composed of methods.

$a!l[\tilde{v}]$	$a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$	message/object

In the syntax above, a is a name, $\tilde{v}, \tilde{x}_1, \ldots, \tilde{x}_n$ are sequences of names, and $l, l_1, \ldots, l_n$ are *labels*. Labels constitute a syntactic category distinct from names. Labels $l_1, \ldots, l_n$, and names in each $\tilde{x}_i$, are pairwise distinct. A message $a!l_i[\tilde{v}]$ selects the method l_i in an object $a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$; the result is the process P_i where names in $\tilde{v}$ replace those in $\tilde{x}_i$. These primitives are further combined by the following standard constructs in concurrent programming.

$P_1 \mid P_2$	concurrent composition
new x P	name hiding
def $X_1(\tilde{x}_1) = P_1$ **and** $\ldots$ **and** $X_n(\tilde{x}_n) = P_n$ **in** P	recursion
$X[\tilde{v}]$	instantiation

Contrary to the conventional practice in name-passing calculi, we let the

scope of a **new** extend as far to the right as possible. We single out a label — *val*— to be used in objects with a single method. This allows to abbreviate messages and objects. The **let** constructor is quite useful in getting back results; the syntax is taken from Pict [13].

$a![\tilde{v}]$	abbreviates	$a!\mathit{val}[\tilde{v}]$
$a?(\tilde{x}) = P$	abbreviates	$a?\{\mathit{val}(\tilde{x}) = P\}$
let $x = a!l[\tilde{v}]$ **in** P	abbreviates	**new** r $a!l[\tilde{v}r] \mid r?(x) = P$

This finishes the introduction of all language constructs we shall use in this section; we may now go into our example. An applet server provides for the downloading of k different applets through the k methods of an object. The server locates applet P_j at the name p provided with the invocation of method applet_j. Here is the code to be run at site sumatra.

```
def AppletServer (self) =
   self ? {
      applet1(p) = p?(x)=P1  |  AppletServer[self],
      ...
      appletk(p) = p?(x)=Pk  |  AppletServer[self]}
in export appletserver
in AppletServer[appletserver]
```

Each client creates a fresh name where the applet server is supposed to locate the applet, then invokes the server with this name and, in parallel, triggers the applet.

```
import appletserver from sumatra
in new p appletserver!appletj[p]  |  p![v]
```

Let us see how the server and the client interact. We start by translating the **import**/**export** clauses to obtain

```
sumatra: def ... in AppletServer[appletserver]  ||
client: new p sumatra·appletserver!appletj[p]  |  p![v]
```

Then, the message sumatra·appletserver!applet_j[p] moves to the server (yielding the message appletserver!applet_j[client·p]) with one Move reduction step, one local reduction at the server invokes the applet_j method, and one final Move step migrates the applet client·p?(x)=P_j back to the client, yielding the process: [5]

```
sumatra: def ... in AppletServer[appletserver]  ||
client: new p p?(x)=Pjσ_sumatra client  |  p![v]
```

Notice how the structural congruence rules New and Extr are used (from

[5] Incidentally, three is the number of reduction steps that Mobile Join [7] takes to perform the same operation.

left to right) to allow name p at client to encompass both sites, and then (from right to left) to bring p local to the client again. Notice also that the applet body gets translated to reflect its new site: if P refers to some name a local to the applet server, then $P_j\sigma_{\text{sumatra client}}$ refers to the remote name sumatra·a.

It should be obvious that a client does not need to download the applet to its site; a message appletserver!$applet_j$[s·p] will load the applet at site s.

3.2 Compute server

The next example, inspired by Cardelli [4], distinguishes local from remote computation. A compute server provides two operations, *lexec* and *rexec*, allowing the execution of a given parameterless procedure P at the client site and at the server site, respectively. Here is the code to be run at site borneo.

```
def ComputeServer (self, replay) =
   self ? {
      lexec(p) =
         p![] | ComputeServer[self, p],
      rexec(replyTo) =
         new p replyTo![p] | p![] | ComputeServer[self, p]}
in export computeserver
in ComputeServer[computeserver, _]
```

The method for local execution triggers the procedure located at name p. Once again, exactly where the procedure runs depends on where p is located, and that is in the hands of the client. The method for remote execution provides for the migration of the procedure to the server by creating a new name p (local to the server) and by sending it back to the client. The client is then supposed to locate the procedure at this name while, as in the applet example, the server triggers the procedure. As in the original example [4], the server cheats on clients by storing the latest client procedure in (this time) a local variable. [6]

Here is a possible client.

```
import computeserver from borneo
in new p p?()=P | computeserver!lexec[p] |      – local execution
   let p = computeserver!rexec[] in p?()=P      – remote execution
```

For the local execution, the client creates a new name p where it locates the procedure, and invokes *lexec* with argument p. For the remote execution the client waits for a name from the server and locates the procedure at this name. In both cases the triggering is done by the server. We can see that the difference between the two kinds of execution is centered on where the procedure identifier p is located.

[6] Since the variable replay is local to the server, there is not much use to it. We could however add a *replay* method to ComputeServer: *replay*() = replay![] | ComputeServer[self, replay].

While in the previous example, the applet server defines the procedures (applets) and provides for the uploading, in this example it is the client that defines the procedures to be run. The local execution takes three reduction steps until P is ready to be triggered; the remote execution takes five steps to accomplish the same. The two extra steps involve asking for and getting a name p local to the server, where the procedure is to be located.

3.3 Spawning processes

An important design decision is that "*remote channel creation is only possible with a remote friend.*" Hence "**new** name **at** site" is something we cannot write.[7] The knowledge of a site name must not award the possibility of directly accessing the site's memory. The consequences would be far reaching. In particular, such a construct would allow the spawning of arbitrary processes regardless of the willingness of the server to accept the processes. Spawning a process P at site s without s's consent could be easily written as[8]

new a **at** s a?()=P | a![].

Instead, to model arbitrary migration, we require the collaboration of some friend in the remote location to provide a remote name. Friends can be written as follows.

Friend(self) = self?{*newName*(replyTo) = **new** a replyTo![a] | Friend[self]}

Thus, spawning a process P in a location where we have aFriend can be modeled as

spawn P **at** aFriend $\stackrel{\text{def}}{=}$ **let** p = aFriend!*newName*[] **in** p?()=P | p![].

We have already used this technique in the remote execution method of the compute server (section 3.2) only that there the migrating process is triggered by the server.

An immediate application of this technique allows us to send a computation to a remote server and to get the results (cf. [7]). Here the client defines the request, the request moves to the server, runs there, and sends the result back to the client. Suppose that R is a request that eventually issues a message a![v] with the result v, and e is the name of a friend at the server. Then, we may send R to the remote server, get the result in x and continue with P, by simply writing:

new r **spawn** R **at** e | a?(x)=P.

We can specialize remote friends. Here is one that accepts the migration of arbitrary processes (with the necessary collaboration of the client; see method *rexec* of the compute server, section 3.2), and invokes them.

[7] We stick to the idea of not writing located names explicitly. The counterpart in the (extended) base calculus would be $\nu s a\, P$. Sewell et al. [16] write (**new** $a@s$)P.

[8] Amadio writes **spawn**(s, P) [1]; Hennessy-Riely write $s :: P$ [8], and also **goto**(s, P) [14].

Friend(self) = self?{*migrate*(replyTo) = **new** a replyTo![a] | a![] | Friend[self] }

Since the procedure is triggered at the server, this version saves one remote message passing when compared to the method *newName*. We could go one step forward and stipulate a gateway for each site providing for all the services we could anticipate for the site, as in Amadio [1]. The gateway name would then represent the site itself and we could work with gateway names as if we were dealing directly with sites.

There is also an implementation related reason why we do not want remote name creation. All our remote primitives (but **export**/**import**) are accomplished with a single (asynchronous) remote message passing. To implement remote name creation we would need two remote messages (one asking for the creation, the other replying the name created).

3.4 Migrating a buffer cell

This example uses the friends discussed above. Inspired on Amadio's "migration stack" [1], we have down sized the stack into a one-place buffer cell in order to simplify the migration of the state. Our cell provides for *read*, *write*, and *move* operations. The last operation allows the migration of the whole cell (that is, the cell itself and its value) to a new site. The invoker of the move operation must provide for a friend at the remote location; in return it gets the new location of the cell. We assume that the value the cell is holding possesses a *move* method as well.

```
def Cell (self, value) =
   self ? {
      write(newVal) =
         Cell[self, newVal],
      read(replyTo) =
         replyTo![value] | Cell[self, value],
      move(aFriend, replyTo) =
         let newSelf = aFriend!newName[]
         in let newVal = value!move[aFriend]
         in replyTo![newSelf] | Cell[newSelf, newVal]}
```

3.5 FTP server

Our final example is written in Structured Communication-Based Programming [10] extended with **import**/**export** declarations, thus showing that the ideas of this paper can be embodied into different languages. Before we go into the example we briefly review the syntax of the language.

The idea central to the idiom is a *session*. A session is a series of reciprocal interactions between two parties, possibly with branching and recursive structures, and serves as a unit of abstraction for the structure of interaction. Communications which belong to a session are done via a port specific to that

session. A fresh channel is generated when initiating each session, for the use in communications in the session. To initiate a session we use **request** and **accept** commands.

request $a(k)$ **in** P	**accept** $a(k)$ **in** P	initiation of a session

A **request** first requests, via a name a, the initiation of a session as well as the generation of a fresh channel k, then P would use the channel for later communications. An **accept**, on the other hand, receives the request for the initiation of a session via a, generates a new channel k, which would be used for communications in P. The parenthesis (k) and the keyword **in** shows the binding and its scope. Via a channel of a session, three kinds of atomic interactions are performed: *value passing* (including name passing), *branching*, and *channel passing* (or *delegation*).

$k![e_1 \dots e_n]; P$	$k?(x_1 \dots x_n)$ **in** P	data sending/receiving
$k \lhd l; P$	$k \rhd \{l_1 = P_1, \dots, l_n = P_n\}$	label selection/branching
throw $k[k']; P$	**catch** $k(k')$ **in** P	channel sending/receiving

Data sending/receiving is the standard synchronous message passing. Here e_i denotes an expression such as arithmetic/boolean formulae as well as names. The *branching/selection* is the minimization of method invocation in object-based programming. $l, l_1, \dots, l_n$ are *labels*. Similarly to TyCO, variables $x_1, \dots, x_n$ and also labels $l_1, \dots, l_n$ are pairwise distinct. The *channel sending/receiving*, which we often call *delegation*, passes a channel which is being used in a session to another process, thus radically changing the structure of a session. Sessions are combined via concurrent composition, name hiding, and recursion, as described in section 3.1 for TyCO.

Our example, taken from Honda et al. [10], is composed of an FTP server and a pool of threads. The FTP server establishes a session with a client and, after authenticating the client (code not shown), delegates the session to some idle thread. The server is then free to take another client request. The novelty of the example is that threads may be located at a different machine.

```
def Ftpd (self, ready) =
   accept self(aClient)
   in accept ready(aThread)
   in throw aThread[aClient] | Ftpd[self, ready]
in export ftp
in import ready from threadSite
in Ftpd[ftp, ready]
```

Site ftpServer runs the above code while importing name ready from the site providing for the threads, and exporting name ftp to potential clients.

```
def Thread (ready) =
   accept ready(ftp)
   in catch ftp(aClient)
   in def Actions() =
      aClient ▷ {
         put= aClient?(aFile) in ··· Actions[],
         get= aClient?(aFilename) in ··· Actions[],
         quit= Thread[ready]}
   in Actions[]
in export ready
in Thread[ready]
```

Threads run at threadSite, exporting name ready to potential ftp-servers. Idle threads accept service from the ftp-server and catch client's sessions. The session with the client is then initiated by means of the loop Actions. Here is a client that requests a session with the ftp server, puts a file and quits.

```
import ftp from ftpServer
in request ftp(aSession)
in aSession◁put; aSession![myFile]; aSession◁quit
```

4 Implementation

The model discussed in section 2 can be easily incorporated in the TyCO programming environment.

For the implementation of base processes we rely on the technology we have developed: the TyCO abstract machine [11], TyCOAM for short. Programs in TyCO are first compiled into an intermediate assembly language and then assembled into byte-code files, which in turn are emulated by the TyCOAM.

To emulate a byte-code program, a TyCOAM relies on two distinct address spaces: a heap and a program area. The program area contains static data and the byte-code. The heap is used for dynamic allocation of frames (blocks of contiguous machine words) for data-structures such as messages, objects and channels. Message frames contain the label and the arguments of the message; object frames contain the address of the object's method table in the program area, and the values for the object's free variables. Channels are queues of either messages or objects (or empty) waiting for reduction.

Sites are abstract "places" where computations evolve. We associate a unique TyCOAM with each site. To take advantage of multiprocessors, we do not map sites one-to-one with IP addresses. Instead we allow several sites to coexist at a given IP node. Therefore, a site identifier is a pair *ip-location* where *location* is a small natural number selecting a site within the IP node. To handle communication between distinct sites we endow each IP node with a communication daemon, TyCOd for short. Thus each IP node is formed by an arbitrary number of TyCOAMs plus a TyCOd. Sites at the same IP node run

in parallel (if the architecture allows), or interleaved (in mono-processors). In either case, the scheduling of the TyCOAMs is left either to a thread package or to the local OS kernel. Within and IP node, each site has its own address space in a global shared memory; the TyCOd has access to each of these address spaces. Figure 1 illustrates the architecture of an IP node.

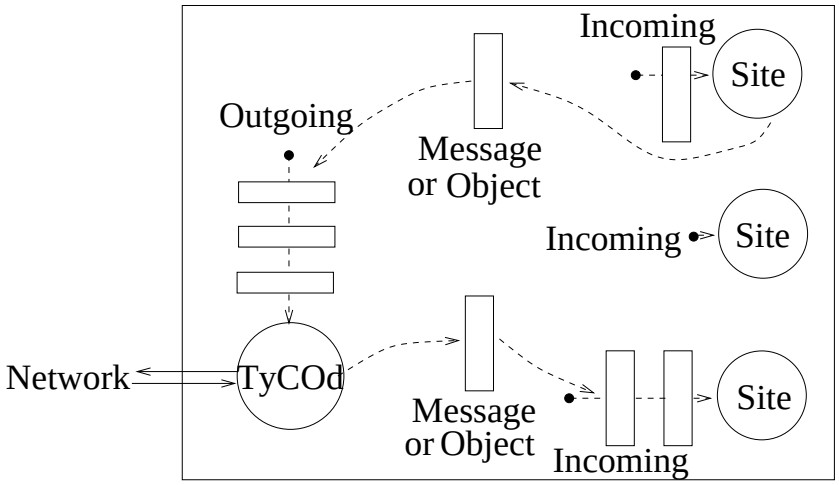

Fig. 1. IP Node Architecture

Each TyCOd maintains a symbol table relating exported names with local channels. All **export**/**import** declarations in a program are processed at launch time: an "**export** name" declaration updates the symbol table with the channel associated with name; an "**import** name **from** site" clause enquires site for the channel of name and binds the result locally.

For remote messages, the TyCO compiler generates a specialized assembly instruction – `remote-message`, instead of the usual `try-reduce-message` which is used for local communication. Similarly, for object migration, a specialized `remote-object` instruction is generated in place of the usual `try-reduce-object`.

A TyCOAM (a site in the figure) executes a `remote-message` instruction by sending to the local TyCOd a request with the target site identifier and the address of the message-frame currently in the heap. The request is placed in a *outgoing* queue maintained by the TyCOd. The daemon eventually processes the request by translating the names in the message frame as defined in section 2, packing the translated frame into an appropriately formated buffer, and then sending the buffer through the network. If the target site is within the same IP node, significant optimizations can be performed. The same approach is taken for `remote-object` instructions. The TyCOd receives a request with the target site identifier and the address of the object-frame. The daemon translates the values (of the free variables of the object) in the frame, uses the address of the object's method table in the frame to extract the byte-code for the object, and sends the translated frame and the byte-code to the TyCOd at the target IP address.

When a remote message or object arrives, the local TyCOd unpacks the buffer into a freshly allocated frame from the heap space of the appropriate site, and places it in the site's *incoming* queue. In the case of an object the byte-code for the methods and the method-table is copied and dynamically linked to the program currently running at the site. Before running a new thread the site checks its *incoming* queue and processes all messages and objects in it.

5 Related Work

Sewell et al. build, on top of the π-calculus, a system that identifies sites, and that allows agents (processes located at a given name) to be themselves located at sites [16]. The runtime system takes care of the current location of agents. The model includes primitives to create a new agent at the current site, to migrate an agent to a site, and to check whether two agents reside at the same site. Our model does not contemplate the notion of agents in this sense.

Fournet et al. introduce a migration primitive that allows a whole running location to move into a new position in the tree of locations (and to trigger some process upon arrival) [7]. In contrast to our proposal where remote names have an explicit syntax, the syntax of Join ensures that names have a unique site at which they are serviced.

Amadio identifies configurations composed of locations, messages, and processes running at locations [1]. Messages include conventional remote messages plus three primitives: to stop a location, to spawn a process at a location, and to check whether a location is alive. As mentioned in the introduction, we have decided not to incorporate these primitives.

Riely and Hennessy identify a CCS based calculus in which processes run at locations. "The language provides operators to kill locations, test the status (dead or alive) of locations, and to spawn processes at remote locations" [14].

Hennessy and Riely's distributed π-calculus [8] is probably the project closest to ours. Among the dissimilarities, Dπ allows the arbitrary spawning of processes at remote locations, and is incapable to send a message directly to a remote location (instead a process that sends the message locally must be spawned at the location).

6 Conclusion

We have presented a model of distribution for process-calculi. The proposal includes among its virtues: the possibility of being embodied in most process calculi (section 2 discusses the assumptions on the base calculus), the extreme simplicity of networks, and the feasible implementation in current hardware. All these features come with a price.

(i) Sites are not first class citizens. Site identifiers, constituting a class distinct from names, need a whole set of operations. There is no point in allowing site names to be passed in messages if we cannot at least perform one the following operations: to create sites locally, to compare sites for equality/inequality, to dynamically form a located name given a site and a name, to test whether a site is alive, to kill a site.

(ii) The model is unable to move a running process (or a whole site) to a different location (cf. [7,16]). We can launch a process at a remote site, but after the process is running there is no means to have it migrated. In particular, we can send a computation to a remote server and get back the result, but not the computation itself (see section 3.3).

We identify four lines for future research.

(i) Enhancing the expressiveness of the language taking into consideration the points identified above.

(ii) The study of the semantic properties of the model proposed.

(iii) The study of type systems to discipline remote computations. It is our believe that a form of distributed subject-reduction is attainable. Also, mixing static with dynamic checking is a promising research direction (cf. [15]).

(iv) The actual implementation of the model into two available hardware architectures: 2 Quad-Pentium Pro machines interconnected with Fast Ethernet, and 8 Dual Pentiums interconnected with Myrinet [2].

Acknowledgments. The first author would like thank Matthew Hennessy, Julian Rathke, and Kohei Honda for fruitful discussions.

References

[1] Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In *COORDINATION'97*, volume 1282 of *LNCS*, pages 374–391. Springer-Verlag, 1997. Full version as Rapport Interne, LIM Marseille, and Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.

[2] Nanette J. Boden et al. Myrinet: A gigabit per second local area network. *IEEE-Micro*, 15(1):29–36, February 1995.

[3] Gérard Boudol. The pi-calculus in direct style. In *24th ACM Symposium on Principles of Programming Languages*, pages 228–241. ACM Press, 1997.

[4] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[5] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155, 1998.

[6] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[7] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.

[8] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. Technical Report 2, Computer Science, University of Sussex, February 1998.

[9] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[10] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[11] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. Compiling process calculi. DCC 98–3, DCC-FC & LIACC, Universidade do Porto, March 1998.

[12] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992.

[13] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. CSCI Technical Report 476, Indiana University, March 1997.

[14] James Riely and Matthew Hennessy. Distributed processes and location failures. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Proceedings of ICALP '97*, volume 1256 of *LNCS*, pages 471–481. Springer, 1997. Full version as Report 2/97, University of Sussex, Brighton.

[15] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. Technical Report 4, Computer Science, University of Sussex, July 1998.

[16] P. Sewell, P. Wojciechowski, and B. Pierce. Location independence for mobile agents. In *Workshop on Internet Programming Languages*, 1998.

[17] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, Department of Computer Science, University of Lisbon, March 1998.

[18] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *1st ISOTAS*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.