A Concurrent Programming Environment with Support for Distributed Computations and Code Mobility

Luís Lopes, Álvaro Figueira, Fernando Silva
DCC-FC & LIACC, Universidade do Porto
Rua do Campo Alegre, 823, 4150 Porto, Portugal
{lblopes,arf,fds}@ncc.up.pt

Vasco T. Vasconcelos
DI-FC, Universidade de Lisboa
Campo Grande, 1700 Lisboa, Portugal
vv@di.fc.ul.pt

Abstract

We propose a programming model for distributed concurrent systems with mobile objects in the context of a process calculus. Code mobility is induced by lexical scoping on names. Objects and messages migrate towards the site where their prefixes are lexically bound. Class definitions, on the other hand, are downloaded from the site where they are defined, and are instantiated locally upon arrival. We provide several programming examples to demonstrate the expressiveness of the model. Finally, based on this model we describe an architecture for a run-time system supporting concurrent, distributed computations and code mobility.
Keywords: Distributed Computing, Process-Calculus, Concurrency, Code Mobility, Implementation.

1 Introduction

Over the last few years, the increase in speed of both personal computers and network connections has fostered an ever growing research interest in distributed computing. Moreover, research on computation or code mobility has become one of the leading edges of computer science, with vast applications in *web* languages, intelligent mobile agents, cryptography, and high-performance computing, to name a few.

The introduction of the π-calculus [11, 17] and other related process calculi, in the early nineties, as a model for concurrent distributed communicating systems provided a theoretical framework upon which researchers could build solid results. The main abstractions in these calculi are processes, representing arbitrary computations and, names, representing places where processes synchronize and exchange data. Recent extensions of these models allowed us for the first time to glimpse, in a rigorous way, the complexity of distributed systems with mobile resources [3, 6, 9, 23, 25].

Underlying these models is the general concept of *mobility*, that is the ability for processes, objects or computations to dynamically change their location, site, or access rights, as the system evolves. Mobility comes in two flavors: *weak* mobility, meaning code movement between computations, and; *strong* mobility, meaning the movement of entire computations through the network [10].

The advantages of a process-calculus approach to the development of concurrent distributed systems are manifold:

1. the calculi provide a natural programming model as their main abstractions deal with the notions of communication and concurrency;

2. they usually have very small kernel definitions, with well understood semantics providing ideal frameworks for language definitions. This significantly diminishes the usual gap between the semantics of the language and that of its implementation;

3. they are scalable in the sense that high level constructs can be readily obtained from encodings in the kernel calculus;

4. they commonly feature type systems that allow not only to check the type safeness of programs but also to collect important information for code optimization.

Our approach in the development of a concurrent programming language and run-time system for distributed systems is distinct from other works using CORBA [2], DCOM [1] or Java/RMI, although we share some of the goals. The main differences can be summarized as follows:

1. we are interested in developing systems provably correct, with relatively simple, well defined semantics;

2. our computations are *network aware*, i.e, names can be local or remote (belonging to other sites) and the distinction is explicit in the syntax. The system does not provide the illusion of a single, local, address space;

3. we are interested in exploring the concurrency and fine grained parallelism in our model with high-performance, low-cost clusters of PCs, interconnected with Giga-switch

network technology;

4. we provide inter-platform support in heterogeneous networks by using emulated byte-code for implementation technology, much like Java/RMI.

Distributed TyCO – DiTyCO, the model and system architecture we propose in this paper, is based on a form of the π-calculus called Typed Concurrent Objects – TyCO [24]. The model features, first class objects, asynchronous method invocations, and class definitions. The calculus formally describes the concurrent interaction of ephemeral objects through asynchronous communication. Synchronous communication is implemented by sending continuations in messages. Classes are processes parameterized on a sequence of variables. Unbounded behavior is modeled through explicit instantiation of recursive classes.

Support for a distributed setting is introduced with the notion of location or site: places where TyCO computations occur. Names local to a site are lexically bound to it as in Obliq [5]. Code movement is triggered by the lexical scope of names prefixing method invocations, objects or class definitions. The migration of prefixed processes is deterministic, point to point, and asynchronous. Synchronization only happens locally, at reduction time.

The DiTyCO [23] model presents a *flat organization of sites*, that reflects the architectures of current implementations of high-performance networks, namely Giga-Switches [7, 8]. We feel that, the site organization should map the low level hardware architecture as closely as possible to allow an efficient implementation of the model.

The amount of parallelism/concurrency generated during the execution of programs is very large and very fine grained, typically a few tens of byte-code instructions per thread. To exchange items with such granularity without loosing performance we need a very low latency network and high-bandwidth. We use the concurrency in our model to effectively hide the existing communication latency by performing fast context switches between local threads. Thus we are using as our target architecture, low-cost, off-the-shelf, clusters of PCs interconnected with a high speed network.

At the programming level, DiTyCO grows from TyCO mainly by introducing two simple constructs for making names and classes visible to the network (**export**) and for making network names and classes available to local processes (**import**). From an implementation point of view the DiTyCO source code is compiled into byte-code for an extended TyCO virtual machine [15]. Each site is implemented as a thread that runs an extended TyCO virtual machine with private registers and memory areas. A site is created to run a DiTyCO program and is freed when the program ends.

The outline of the paper is as follows. The next section briefly introduces the TyCO process calculus, its syntax and semantics; section 3 introduces distribution and code mobility in TyCO. Section 4 describes the programming model induced by the extended calculus, with examples. Section 5 describes the hardware platform we are aiming at and the rationale for this decision. It then continues to describe the software architecture of our implementation of DiTyCO, and related compilation, emulation and performance issues. The paper ends with a brief overview of related work, conclusions and future work.

2 The TyCO Calculus

The focus of this paper is on the extension of the TyCO process calculus and its language implementation to support distribution and code mobility. However, the main ideas that support this work can be easily embodied in any name-passing calculus that satisfies a few mild pre-conditions (e.g. [4, 11, 12, 17, 24]).

TyCO [24] (Typed Concurrent Objects) is a name-passing calculus in the line of the asynchronous π-calculus [11] that uses labeled asynchronous messages and objects composed of methods as the fundamental processes. Messages and objects are prefixed by names that represent their locations. In the sequel we introduce the syntax and semantics of TyCO. The discussion is much abbreviated due to space constraints. For a fully detailed description the reader may refer to the language definition [22].

The basic syntactic categories are *names*, ranged over by a, b, x, y, u, v; *labels*, ranged over by l, k, and; *class variables*, ranged over by X, Y. Let $\tilde{x}$ denote the sequence $x_1 \cdots x_n$, with $n \geq 0$, of pairwise distinct variables. Then the set of processes, ranged by P, Q is given by the following grammar:

P	$::=$	$\mathbf{0}$	terminated process
	$\mid$	$P \mid P$	concurrent composition
	$\mid$	$\textbf{new}\ \tilde{x}\ P$	local channel declaration
	$\mid$	$x\,!\,l[\tilde{v}]$	asynchronous message
	$\mid$	$x?M$	object
	$\mid$	$X[\tilde{v}]$	instance of class
	$\mid$	$\textbf{def}\ D\ \textbf{in}\ P$	definition of classes

A message $x!l_i[\tilde{v}]$ selects the method l_i in an object $x?M$ where M is a collection of methods of the form $\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$; the result is the process P_i where the names in $\tilde{x}_i$ are replaced by the names $\tilde{v}$, denoted $P_i\{\tilde{v}/\tilde{x}_i\}$. This form of reduction is called *communication*.

$$\text{(COMMUNICATION)}$$
$$x\,!\,l_i[\tilde{v}] \mid x?\{l_1(\tilde{x}_1) = P_1\,, \ldots,\, l_n(\tilde{x}_n) = P_n\} \rightarrow P_i\{\tilde{v}/\tilde{x}_i\}$$

Similarly, an instance $X_i[\tilde{v}]$ selects, from a class definition D of the form $X_1(\tilde{x}_1) = P_1$ **and**...**and** $X_k(\tilde{x}_k) = P_k$, the class X_i; the result is the process P_i where the names in $\tilde{x}_i$ are replaced by the names in $\tilde{v}$. This form of reduction is called *instantiation*.

$$\text{(INSTANTIATION)}$$
$$\textbf{def } X(\tilde{x}) = P \textbf{ in } X[\tilde{v}] \to \textbf{def } X(\tilde{x}) = P \textbf{ in } P\{\tilde{v}/\tilde{x}\}$$

Finally, to simplify the syntax somewhat, we let the scope of a **new** extend as far to the right as possible, and we single out a label — *val*— to be used in objects with a single method.

$x![\tilde{v}]$	abbreviates	$x!\mathit{val}[\tilde{v}]$
$x?(\tilde{y}) = P$	abbreviates	$x?\{\mathit{val}(\tilde{y}) = P\}$

We sketch a small example of a single element polymorphic cell. We define a class with two attributes: the self name and the value v of the cell. The class has two methods one for reading the current cell value and another to change it. The recursion at the end of each method keeps the cell alive after a reduction.

```
def Cell(self, v) =
    self ? {
        read(r) = r![v]  | Cell[self, v],
        write(u) = Cell[self, u]
    }
in new x Cell[x, 9]  | new y Cell[y, true]
```

TyCO features a (Damas-Milner) polymorphic type-system and the definition of the cell is polymorphic on the attribute v. This is why we can instantiate a Cell with an integer and another with a boolean attribute. For example, the process

```
new z x![z]  | z?(w)=print[w]
```

when placed in parallel with the above cells, interacts with the integer cell located at x invoking the method *read* in that cell. The reply is a message directed to z with the cell's value 9. Finally, this message would interact with the process z?(w)= *print*[w] to invoke its only method to print the value 9.

3 Distribution and Code Mobility

Using TyCO as our base calculus we proceed to define the next layer: *networks*. We introduce a new class of identifiers, *sites*, distinct from any class of identifiers in the base calculus. We let r and s range over the set of sites. *Located names*, are site-name pairs; we write $s.x$ for name x located at site s. Similarly, *located class variables*, are site-class variable pairs, written $s.X$, denoting the class variable X located at site s. Names and class variables are collectively known as *located identifiers* or simply *identifiers*.

We then allow located identifiers to occur in any position in the base calculus where (non-binding occurrences of) identifiers can. The calculus thus obtained constitutes the *first layer* of the model. Since site identifiers are introduced anew, there must be no provision in the base calculus for binding located identifiers. As such, at this level, a located identifier behaves as any other constant in the base calculus. The *second layer* is composed of site-process pairs, denoted $s[P]$, and called *located processes*, composed via conventional parallel, restriction, and definition operators. Thus, the set of networks is given by the following grammar.

N	$::=$	$\mathbf{0}$	terminated network
	$\mid$	$s[P]$	located process
	$\mid$	$N \parallel N$	concurrent composition
	$\mid$	$\textbf{new } s.x\ N$	scope restriction
	$\mid$	$\textbf{def } s.D \textbf{ in } N$	class definition

The bindings in networks are as expected: a located name $s.x$ occurs *free* in a network if $s.x$ is not in the scope of a **new** $s.x$; otherwise $s.x$ occurs *bound*. The set of free located names in a network, notation $\mathrm{fn}(N)$, is defined accordingly. Similarly, a located class variable $s.X$ occurs free in a network N if $s.X$ is not in the scope of a **def** $s.D$, for X one of the X_i defined in D. The set $\mathrm{ft}(N)$ of free located class variables in networks 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 parallel composition, as well as the following rules.

$$
\begin{array}{rl}
\text{(NIL)} & s[\mathbf{0}] \equiv \mathbf{0} \\
\text{(SPLIT)} & s[P_1] \parallel s[P_2] \equiv s[P_1 \mid P_2] \\
\text{(NEW)} & s[\textbf{new } x\ P] \equiv \textbf{new } s.x\ s[P] \\
\text{(DEF)} & s[\textbf{def } D \textbf{ in } P] \equiv \textbf{def } s.D \textbf{ in } s[P] \\
\text{(GCN)} & \textbf{new } s.x\ \mathbf{0} \equiv \mathbf{0} \\
\text{(GCD)} & \textbf{def } s.D \textbf{ in } \mathbf{0} \equiv \mathbf{0} \\
\text{(EXN)} & N_1 \parallel \textbf{new } s.x\ N_2 \equiv \textbf{new } s.x\ (N_1 \parallel N_2) \\
 & \text{if } s.x \notin \mathrm{fn}(N_1) \\
\text{(EXD)} & N_1 \parallel \textbf{def } s.D \textbf{ in } N_2 \equiv \textbf{def } s.D \textbf{ in } (N_1 \parallel N_2) \\
 & \text{if } \mathrm{bt}(D) \cap \mathrm{ft}(N_1) = \emptyset
\end{array}
$$

Rule NIL garbage collects terminated located processes, whereas rules GCN and GCD garbage collect unused names and definitions, respectively. 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 (messages, objects, instantiations) to be transported over the network (see rules SHIP in the reduction relation below). There are also rules that allow the scope of a name (rule NEW) or of a class definition (rule DEF) local to a process, to extrude and encompass a network with several located processes (rules EXN and EXD).

Simple names in processes are implicitly located at the site where the process occurs; a name x or a class variable X occurring in a located process $s[P]$ is implicitly located at site s. When sending identifiers over the network their implicit locations need to be preserved, if we are to abide by the *lexical scoping convention*. A name x is uploaded to a site s; a class variable X is downloaded from a site s. A *translation of identifiers* from site r to site s is a total function σ_{rs} defined as follows:

$$\sigma_{rs}(x) \stackrel{\text{def}}{=} r.x \qquad \sigma_{rs}(s.x) \stackrel{\text{def}}{=} x \qquad \sigma_{rs}(s'.x) \stackrel{\text{def}}{=} s'.x$$
$$\sigma_{rs}(X) \stackrel{\text{def}}{=} r.X \qquad \sigma_{rs}(s.X) \stackrel{\text{def}}{=} X \qquad \sigma_{rs}(s'.X) \stackrel{\text{def}}{=} s'.X$$

We are now ready to define the reduction relation over networks. The first rule, LOC, allows processes in sites to evolve locally.

$$(\text{LOC}) \quad \frac{P \to Q}{s[P] \to s[Q]}$$

This rule, plus the familiar rules for parallel composition, restriction, definitions and structural congruence which we omit, form the backbone of the reduction relation. We now discuss the three remaining axioms that introduce weak mobility.

Processes prefixed at located identifiers play a crucial role in the model. They determine the semantics associated with code movement. A process $s.x!l[\tilde{v}]$ represents a remote method invocation on an object located at a name x at site s. The message moves to site s. On the other hand, a process $s.x?M$ denotes an object that must be located at name x at site s. In other words the located prefix name induces a code migration operation between the current site and site s. Note that conceptually there is not much difference between a remote method invocation and an object migration; in section 5 we show that from an implementation point of view the difference is not abysmal either. Thus, we see that lexical scope on names induces *code shipping* semantics for method invocations and objects. The axioms are as follows:

$$(\text{SHIPM}) \quad r[s.x!l[\tilde{v}]] \to s[x!l[\tilde{v}\sigma_{rs}]]$$
$$(\text{SHIPO}) \quad r[s.x?M] \to s[x?M\sigma_{rs}]$$

Note that, if the method invocation $s.x!l[\tilde{v}]$ (respectively, object $s.x?M$) is located at site r, then, in order to keep the lexical scope of names, the free names in $l[\tilde{v}]$ (respectively M) must be translated accordingly to $l[\tilde{v}\sigma_{rs}]$ (respectively $M\sigma_{rs}$). So, when sending $s.x!l[\tilde{v}]$ (respectively $s.x?M$) from r to s we actually transmit $l[\tilde{v}\sigma_{rs}]$ (respectively $M\sigma_{rs}$). This is the essence of the axioms SHIP.

As an example let us try a remote procedure call in TyCO. 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).

$s[\textbf{new}\ a\ (r.p\,!\,[va] \mid a?(y) = P)] \ \| \ r[p?(xr) = Q] \equiv$
applying: NEW, EXN
$\textbf{new}\ s.a\ (s[r.p\,!\,[va]] \,\|\, s[a?(y) = P] \,\|\, r[p?(xr) = Q]) \to$
applying: SHIPM
$\textbf{new}\ s.a\ (r[p\,!\,[s.v\ s.a]] \,\|\, s[a?(y) = P] \,\|\, r[p?(xr) = Q]) \equiv$
applying: SPLIT
$\textbf{new}\ s.a\ s[a?(y) = P] \ \| \ r[p\,!\,[s.v\ s.a] \mid p?(xr) = Q] \to$
applying: LOC
$\textbf{new}\ s.a\ s[a?(y) = P] \ \| \ r[Q\{s.v\ s.a/xr\}] \to^*$
Q reduces
$\textbf{new}\ s.a\ s[a?(y) = P] \ \| \ r[s.a\,!\,[u]] \to\equiv\to$
applying: SHIPM, SPLIT, LOC
$\textbf{new}\ s.a\ s[P\{r.u/y\}] \equiv$
applying: NEW
$s[\textbf{new}\ a\ P\{r.u/y\}]$

We thus see that a remote communication involves two reduction steps: one to get the method invocation/object to the target site and the other to consume the message/object at the target (cf. [9]); the former is an asynchronous operation, the latter requires a rendez-vous. This reflects actual implementations.

Another kind of remote interaction occurs when a site s finds an instantiation of the form $r.X[\tilde{v}]$. A class variable X prefixed with a site name r indicates that X was originally defined at site r. The class definitions (classes in particular) have to be downloaded from site r to site s. The following axiom does the trick.

$$(\text{FETCH}) \quad \frac{X \in \text{dom}(D)}{\textbf{def}\ r.D\ \textbf{in}\ s[r.X[\tilde{v}]] \to \textbf{def}\ r.D\ \textbf{in}\ s[\textbf{def}\ D\sigma_{rs}\ \textbf{in}\ X[\tilde{v}]]}$$

Thus, contrary to lexical scope in names, the lexical scope of a class variable induces *code fetching* semantics in a way reminiscent of languages such as Java. Also note that, again, when copying D to s we must keep the lexical bindings for the free variables of D, so we actually download $D\sigma_{rs}$. We opt to download D instead of just the definition for X in it since often X will be a mutually recursive definition involving other classes in D.

Below we present an example of this kind of interaction. We move a piece of code from site s to site r; the code contains a class variable X local to r. The definition for X

is downloaded from site r thereafter.

$r[$**def** $X(x) = P$ **in** $s.a?() = X[b]] \parallel s[a![]] \equiv$
applying: DEF, EXD
def $r.X(x) = P$ **in** $(r[s.a?() = X[b]] \parallel s[a![]]) \rightarrow$
applying: SHIPO
def $r.X(x) = P$ **in** $(s[a?() = r.X[r.b]] \parallel s[a![]]) \rightarrow$
applying: SPLIT, LOC
def $r.X(x) = P$ **in** $s[r.X[r.b]] \rightarrow$
applying: FETCH
def $r.X(x) = P$ **in** $s[$**def** $X(x) = P\sigma_{rs}$ **in** $X[r.b]] \rightarrow$
applying: LOC
def $r.X(x) = P$ **in** $s[$**def** $X(x) = P\sigma_{rs}$ **in** $P\sigma_{rs}\{r.b/x\}]$

4 Programming

The programming model associated with the framework described in the previous section is rather simple requiring just two new constructs.

export new x P **import** x **from** s **in** P
export def D **in** P **import** X **from** s **in** P

A site uses the **export** construct to provide identifiers to other sites in a network. In other words **export** is used to declare the external interface of a site. Other sites in the network use these exported identifiers for local computations with the help of the **import** construct. As we have seen however, the semantics associated with imported names is *code shipping* whereas for imported class variables it is *code fetching*. The syntax of the base language remains unchanged, since we never write located identifiers explicitly. The translation of the above constructs into the base calculus extended with located identifiers is straightforward.

$$[\![s[\textbf{export new}\ x\ P] \parallel N]\!] \overset{\text{def}}{=} \textbf{new}\ s.x(s[[\![P]\!]] \parallel [\![N]\!])$$
$$[\![\textbf{import}\ x\ \textbf{from}\ s\ \textbf{in}\ P]\!] \overset{\text{def}}{=} [\![P\{s.x/x\}]\!]$$
$$[\![s[\textbf{export def}\ D\ \textbf{in}\ P] \parallel N]\!] \overset{\text{def}}{=} \textbf{def}\ s.D\ \textbf{in}\ (s[[\![P]\!]] \parallel [\![N]\!])$$
$$[\![\textbf{import}\ X\ \textbf{from}\ s\ \textbf{in}\ P]\!] \overset{\text{def}}{=} [\![P\{s.X/X\}]\!]$$

The remainder of this section is devoted to a couple of programming examples in the extended language, DiTyCO, to attest the simplicity and flexibility of the model.

The first example is adapted from Fournet et. al. [9] and illustrates an applet server. It models the download of applet byte-code over the network of sites. We give two possible implementations based on, respectively, *code fetching* and *code shipping* semantics.

The first program defines the applet server as a collection of class definitions. The client site uses code fetching to select a given class (that is, the target applet) from the server. When the client creates an instance of, say, Applet$_j$ it triggers the code movement from the server site to the client site, since Applet$_j$ is lexically bound to the server site. Once the code arrives at the client site, the instantiation is performed as in the base calculus. The program is as follows.

SERVER SITE

```
export def  Applet_1(x) = P_1
and          ...
and         Applet_k(x) = P_k
in           ...
```

CLIENT SITE

```
import Applet_j from server
in Applet_j[v]
```

The second program defines the applet server as a class whose methods, once invoked, ship the code for an applet. At the server site, the invocation of a method *applet*$_j$ causes the applet P$_j$ to be shipped to the name p lexically bound to the client site. 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. The program is as follows.

SERVER SITE

```
def AppletServer (self) =
   self ? {
      applet_1(p) = p?(x)=P_1  |  AppletServer[self],
       ...
      applet_k(p) = p?(x)=P_k  |  AppletServer[self]
   }
in export new appletserver
   AppletServer[appletserver]
```

CLIENT SITE

```
import appletserver from server in
new p appletserver!applet_j[p]  |  p![v]
```

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

```
new server.appletserver
server[def ... in AppletServer[appletserver]]  ||
client[new p server.appletserver!applet_j[p]  |  p![v]]
```

Then, the message server.appletserver!*applet*$_j$[p] moves to the server (yielding the message appletserver!*applet*$_j$[client.p]) with one SHIPM reduction step, one local reduction at the server invokes the *applet*$_j$ method, and one final SHIPO step migrates the applet client.p?(x)=P$_j$ back to the client, yielding the process:

new server.appletserver
server[**def** ... **in** AppletServer[appletserver]] ||
client[**new** p p?(x)=$\mathsf{P}_j\sigma_{\text{server client}}$ | p![v]]

Notice how the structural congruence rules NEW and EXN are used (from 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 a name x local to the applet server, then $\mathsf{P}\sigma_{\text{server client}}$ refers to the remote name server.x. It should be obvious that clients may download the applet to any site; a message appletserver!*applet*$_j$[site.p] migrates the code to site.

The next example is inspired in the SETI (Search for Extra-Terrestrial Intelligence) research program. Seti@home was developed by the SETI research team as a way to deal with the vast computational power required to process data obtained by the program's radio-telescopes. Here we model an application that only requires a single command from a remote client to be downloaded from the SETI server site. Hence forward, the application runs "forever" at the client site processing data chunks from SETI's database. The code for both server and client sites is given below. Note that, the process **let** $x = a\,!\,l[\tilde{v}]$ **in** P abbreviates **new** r $a\,!\,l[\tilde{v}r] \mid r?(x) = P$, thus hiding the reply-to name r.

SETI SITE

```
new database
export def Install() = ⟨install⟩; Go[]
and Go() = let data = database!newChunk[]
           in ⟨process⟩; Go[]
in
database ? {
   newData(data) = ...
   newChunk(replyTo) = ...
}
```

CLIENT SITE

```
import install from seti
in Install[]
```

The code in the client just instantiates a copy of the Install class at the seti server site. The FETCH rule migrates a copy of the code for Install to the client site. Free identifiers in the code get translated to expose the lexical binding. Thus, after an application of FETCH we have the following situation at the client the following process.

def (Install() = ... **and** Go() = ...)$\sigma_{\text{seti client}}$ **in** Install[]

After a local instantiation, the installation procedure (denoted as ⟨install⟩) runs and configures all that is necessary for the program to start. The system starts by calling method newChunk. The **let** abbreviation describes a remote method invocation, implemented via two (asynchronous) remote messages. The reply to this method call, from the SETI database, is a chunk of data to be processed locally at the client. After this step we get the following situation:

def (Install() = ... **and** Go() = ...)$\sigma_{\text{seti client}}$
in (⟨process⟩; Go[])$\sigma_{\text{seti client}}$

The client gets the data chunk and does the required number crunching (denoted by ⟨process⟩). Afterwards the program at the client will run "forever" inside this Go loop.

5 The Implementation of DiTyCO

In this section we describe the architecture of the DiTyCO run-time system and specifically how it interfaces with the TyCO virtual machine developed for the calculus [15]. We start by describing the kind of hardware platform we are interested in and then proceed to map DiTyCO on top of it.

The Hardware Platform

Our test-bed hardware for DiTyCO consists of a cluster of four dual-processor PCs interconnected with a 1Gb/s Myrinet switch assembled under project Dolphin. Each processor runs the open source, freeware, Linux operating system with support for Symmetric Multi-Processing. Each PC is additionally connected through a Fast-Ethernet (100Mb/s) link to the external network (figure 1). This hardware platform has already been used to develop a prototype for a high-performance explicitly parallel programming language [21].

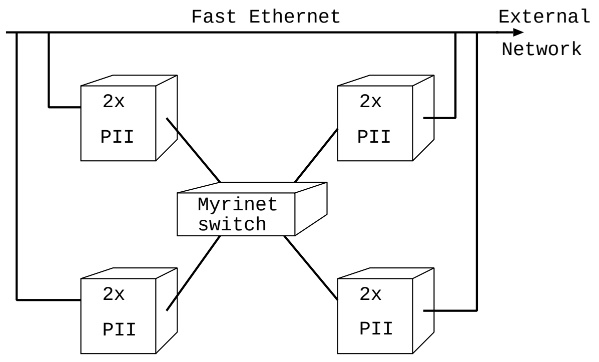

Figure 1. The Hardware Platform.

Our interest in this kind of hardware platform steams from the following reasons:

1. item the parallelism generated by the TyCO virtual machine is very fine grained, typically a few tens of byte-code instructions per thread;
2. to exchange items with such granularity without loosing

performance we need a very low latency network, as provided by these giga-switches. Switches are quite efficient at point-to-point communication as packets do not have to hop through several intermediate nodes before reaching their destinations;

3. the large volume of parallelism, and consequently of messages and code that is sent over the network, requires a network with high bandwidth to achieve good performance;

4. the use of multiprocessing nodes is very important since it allows to perform optimizations in the case of local (within a node) communication. In this case, code movement or message sending can be implemented with a single shared-memory reference exchange;

Last but not least, the fine-grained, pervasive concurrency in our model allows us to effectively hide the existing communication latency by performing fast context switches to other, non-blocked, threads.

The Software Architecture

Our implementation of DiTyCO has three levels: **sites**, **nodes** and **networks**. Sites form the basic sequential units of computation. Nodes have a one-to-one correspondence with physical IP nodes and may have an arbitrary number of sites computing either concurrently or in parallel. This intermediate level does not exist in the formal model. It makes the architecture more flexible by allowing multiple sites at a given IP node. Finally, the network is composed of multiple DiTyCO nodes connected in a static IP topology. Message passing and code mobility occurs at the level of sites, and at this level the communication topology changes dynamically. This structure is illustrated in figure 2.

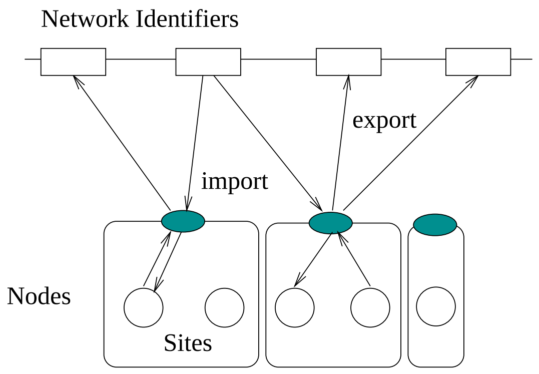

Figure 2. DiTyCO architecture.

1. SITES are the basic units of the implementation. They are implemented as threads, each running a re-engineered TyCO virtual machine [15]. The TyCO virtual machine (TyCOVM) was developed as a run-time system for the TyCO calculus. The architecture includes: a *program area* where the byte-code is kept; a *heap area* for dynamic data-structures such as names, messages and objects; a *run-queue* to keep runnable byte-code blocks and their corresponding environment bindings; a *local variable table* where the bindings of local variables are kept, and; a *stack* for evaluating builtin expressions.

The emulator itself is an implementation of the *name-passing* computational model of TyCO briefly described in section 2. Programs are compiled into an intermediate virtual machine assembly. This in turn is compiled into hardware independent byte-code. The mapping between the assembly and the final byte-code is almost one-to-one. The nested structure of the source program is preserved in the final byte-code. This allows the efficient dynamic selection of byte-code blocks that have to be moved between sites. This design has proved to be quite compact and efficient when compared with related languages such as Pict [18], Oz [16] and Join/JoCaml [13] (see [15, 14] for details).

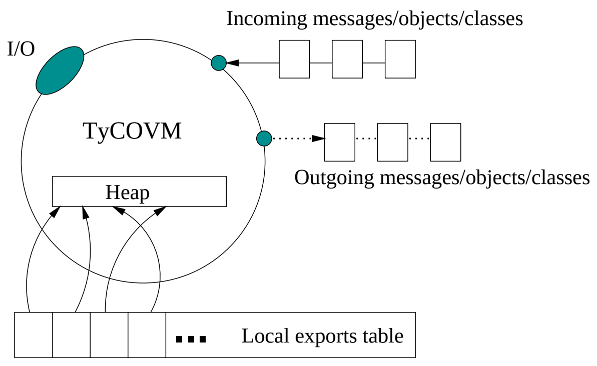
Incoming messages/objects/classes
I/O
TyCOVM
Outgoing messages/objects/classes
Heap
Local exports table

Figure 3. Site architecture.

Sites are implemented as extended TyCOVMs (figure 3). The extensions are required to support remote interaction through an intermediate node-wide communication daemon. In the sequel we give a description of the changes.

Local vs. Network References. Variables may now hold, besides *local references*, *network references*. A local reference is a pointer to the heap of the local site. A network reference, on the other hand, is "a pointer" to a data structure allocated in the heap of some remote site. Network references have a hardware independent representation that keeps information on the remote variable, its site, and IP address.

Network reference: $(\mathsf{HeapId}, \mathsf{SiteId}, \mathsf{IpAddress})$

Local reference: $\mathsf{0x04faa40c}$

Mapping between Local and Network References. An *export table* is needed to map network references into heap pointers for all local variables that leave the site. When the site sends a message, object or class code to another remote site, the free variables in these processes are translated in two steps. Local variables leaving a site are translated into network references. All other variables or data are left untouched. The mapping between the translated local references and their corresponding network references is kept in a local export table. When the data reaches the destination site, the second step of the translation is performed. All variables in the process lexically bound to the destination site are translated into local pointers using that site's export table. Again, the other references or data are left untouched.

New Virtual Machine Instructions. New instructions are required to provide (respectively obtain) the network references for exported (respectively imported) identifiers. This is done by inquiring a network name service that implements the network level of the model (see below). The two additional instructions – `export` and `import` – perform this function. `export` registers a local identifier in the network name service thus making it available to other sites. `import` consults the network name service to find the network reference for an imported identifier. Once the information is available to the local virtual machine remote interaction is possible.

Re-implementation of Instructions for Communication. New semantics are required for the instructions that handle communication in the TyCO virtual machine. These are `trobj` and `trmsg`. The `trobj x` virtual machine instruction tries to reduce an object with some available message located at the position held in variable `x`. Now `x` may hold a pointer to the local heap or a network reference. In the first case, the reduction happens within the site and is implemented just as in the TyCO machine. In the second case `x` is a network reference for a name in the heap of some remote site. The byte-code for the object and the bindings for the free variables (after having been translated) are packaged into a buffer and placed on the outgoing-queue addressed to the remote site. The `trmsg x` instruction is the dual of the above for messages, and the approach is similar. If `x` is a network reference the label and the arguments of the message (after having been translated) are packaged into a buffer and placed in the outgoing-queue addressed to the remote site.

Re-implementation of Instructions for Instantiation. The instruction `instof X` of the virtual machine, creates an instance of a class whose byte-code is located at some program label `X`. Again, we have to consider two cases. If `X` is a local reference, the byte-code for the class is in the program area and reduction proceeds as in the TyCO machine. However, if `X` is a network reference, then it represents a piece of byte-code that lies in some remote site's program area. In this case, an asynchronous message requesting the byte-code is placed in the outgoing-queue addressed to the remote site. Some time later, the reply message with the packaged byte-code is received in the incoming-queue. The code is then dynamically linked to the local program and the reduction proceeds locally. Notice that the synchronous request can be effectively overlapped with computation by rapidly switching context to another local thread.

Queues for Incoming/Outgoing Data. These are required to allow sites to receive/send messages, objects and class definitions dynamically from/to other sites. Incoming processes are placed in the incoming-queue of a site by the local communication daemon – TyCOd. The queue is read periodically by the local TyCOVM and after some processing the incoming processes are used in local computations. Likewise, the TyCOVM places outgoing messages, objects or class definitions in the outgoing-queue to be picked up by the local TyCOd.

I/O Port. An I/O port is required for each site for input/output operations so that users may selectively provide data to running programs or receive data from them.

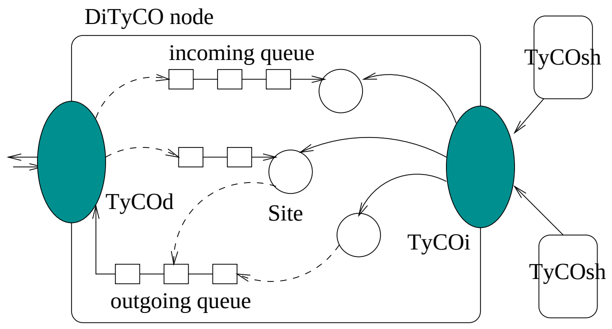

Figure 4. Node architecture.

2. NODES are composed of a pool of sites running concurrently, a dedicated communication daemon (TyCOd), and a user interface daemon (TyCOi). There is one DiTyCO node per IP node. This architecture is illustrated in figure 4.

A DiTyCO node is implemented as a Unix process. The sites, the communication daemon (TyCOd), and the user interface daemon (TyCOi) are implemented as threads sharing the address space of the node. Each site executes a DiTyCO program. New sites are created when a new program is submitted for execution and destroyed when the program exits. Users submit new programs for execution in a node using a shell program called TyCOsh. The user requests are handled by a node manager daemon, the TyCOi.

The TyCOd daemon is responsible for all the data exchange between sites in the network. Interactions between sites may be local, when sites belong to the same node, or remote when the sites belong to different nodes. Local interactions are optimized using shared memory. Remote interactions involve three steps:

1. the site places a packaged process with a destination net-

work reference in the outgoing-queue;

2. the local TyCOd gets the process from the queue, gets the destination IP from the network reference and sends the process to the TyCOd of the node where the remote site is located, and finally;

3. the remote TyCOd takes the process and places it in the incoming-queue of the remote site where it will be picked up by the local TyCOVM.

In addition to forwarding processes for sites, the TyCOd also handles requests from local sites to the network name service. These requests occur when instructions such as `export` and `import` are executed.

3. NETWORKS allow sites to make identifiers publicly available to other sites. Explicitly exported identifiers, as well as site names are registered in a *Network Name Service*. Conceptually, the service maintains two tables, one for sites and another for exported identifiers. Each tuple of the site table includes the lexeme that identifies the site in the source programs (the key attribute), a site identifier (a natural number) and the IP address where the site is located.

$$\mathsf{SiteTable}: \mathsf{SiteName} \mapsto \mathsf{SiteId} \times \mathsf{IpAddress}$$

The key for the table of exported identifiers is compound and uses the lexemes for the identifier and the site it belongs to. Besides the key, each tuple also contains a unique natural number heap identifier.

$$\mathsf{IdTable}: \mathsf{SiteName} \times \mathsf{IdName} \mapsto \mathsf{HeapId}$$

The network address for an identifier is composed by its unique HeapId, the site identifier and its IP location. So, the network reference for an identifier named appletserver at the site server is translated as the tuple:

$$(\mathsf{IdTable}(\mathsf{server},\mathsf{appletserver}), \mathsf{SiteTable}(\mathsf{server}))$$

Currently, in this first implementation, the network name service is centralized and all sites know its location in advance. This will change, as the system matures, into a distributed network name service. This is a fundamental development for reasons of both redundancy (for failure recovery) and performance.

6 Related Work

Work in distributed object-oriented systems has focused in two main areas according to the kind of approach taken in their development.

From a software technology viewpoint, several systems have been proposed that support object distribution in an heterogenous network environment. The main idea is to give the illusion of locality to clients despite the fact that the server objects may be physically allocated somewhere else in a network. The most common systems capable of such functionalities are Microsoft's Distributed Component Object Model (DCOM [1]), OMG's Common Object Request Broker Architecture (CORBA [2]) and JavaSoft's Java/Remote Method Invocation (Java/RMI).

Another approach is to study distributed (object-oriented) systems from theoretical point of view, using frameworks such as process calculi, and try to evolve from this work into working prototypes. There are several proposals for calculi that model distribution, weak (code) mobility and, strong (computation) mobility. Implementations, however, are still very scarce. We briefly describe some of the proposals.

In Mobile Join, Fournet et. al. [9], extend the join-calculus with the notion of *location* defined as a set of processes with a local environment surrounded by a membrane. They are located at sites and can be atomically moved across a web-wide tree of locations. Distribution and mobility are directly supported by explicit primitives in the calculus.

Distributed-π, by Riely and Hennessy [19], is an extension of the π-calculus that models dynamically evolving networks of distributed processes. It incorporates notions of remote execution, migration, and site failure. Located names are explicit in the syntax and their use is controlled by permissions associated with the names.

Nomadic-π, by Sewell et.al. [20] introduces explicit primitives for mobility and distribution into the π-calculus. Processes are executed at sites (locations). They identify two fragments of the resulting calculus: the first lower-level with location dependent primitives while a second fragment is entirely location independent. The semantics of the location independent fragment is then defined based on the lower level fragment.

In Ambients, Cardelli and Gordon [6] introduce a process calculus that models the notions of ambients, agents, actions and capabilities. The calculus is then extended with asynchronous communication and variable binding primitives to model interaction. Mobility is described in terms of the movement of agents and their associated ambients and capabilities through a network. Ambients are bounded places where agents compute. They are self contained. Ambients encounter obstacles while moving through a network and use capabilities to allow others to perform local operations without revealing their true identity.

Seal [25] is a process calculus suited for modeling Internet applications and programming languages. The mobile units of the calculus are called *seals*. The three main abstractions are: *locations* that model, for example, boundaries of networks, IP nodes, address spaces; *processes* modeling flow of control such as threads and OS processes, and; *resources* that model physical resources such as network, memory and peripherals.

7 Conclusions and Future Work

We have introduced a programming model for distributed computations with code mobility that is both intuitive and provides adequate abstractions for coding distributed applications. The model is based on a process calculus framework which makes it amenable to formal verification. Also, many common features in process calculi such as type systems can be used to greatly improve the quality of the generated code in the compilation process. The fine grained parallelism that is generated by the compiler and the asynchronous computational model provides greater flexibility in overlapping communication with local computations. The abstract machine must schedule new threads while the remote operation is being executed. This can be implemented efficiently as each thread is rather small and carries a small state.

Our test-bed hardware consists of a low-cost cluster of PCs, interconnected with a 1Gb/s Myrinet network switch, built in the context of project Dolphin. We consider this platform ideal for experimenting with our model given the constraints in the amount and granularity of the generated parallelism.

Currently, the first DiTyCO prototype is in the final stages of the implementation. We have developed a type checking scheme that ensures that no type mismatch or protocol errors occur in remote interactions. The scheme combines both static and dynamic type checking. On the other hand, we need to introduce fault-tolerance and termination detection into the system. We want to be able to detect site failures, to reconfigure the computation topology and to try to terminate computations cleanly.

Acknowledgements Álvaro Figueira is supported by Ph.D. grant PRAXIS/BD/13971/97. This work was partially supported by projects Dolphin (contract PRAXIS/2/2.1/TIT/1577/95) and DiCoMo (contract PRAXIS/P/EEI/12059/98). The authors wish to thank António Ravara for his insights on the formal model.

References

[1] The COM Specification. *http://www.microsoft.com/oledev/-olecom/title.htm*, 1995.

[2] The Common Object Request Broker: Architecture and Specification, Revision 2.0. *http://www.omg.org/corba/-corbiiop.htm*, July 1995.

[3] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: the Spi-Calculus. In *CCS'97*, 36–47. The ACM Press, 1997.

[4] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[5] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[6] L. Cardelli and A. Gordon. Mobile Ambients. In *FoSSaCS'98*, *LNCS* 1378, 140–155. Springer-Verlag, 1998.

[7] A. C. et al. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *PP'97*, 1997.

[8] N. B. et al. Myrinet: A Gigabit per second Local Area Network. *IEEE-Micro*, 15(1):29–36, February 1995.

[9] C. Fournet, G. Gonthier, and et al. A Calculus of Mobile Agents. In *CONCUR'96*, *LNCS* 1119, 406–421. Springer-Verlag, 1996.

[10] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[11] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *ECOOP'91*, *LNCS* 512, 141–162. Springer-Verlag, 1991.

[12] K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-Based Programming. In *ESOP'98*, *LNCS* 1381, 122–138. Springer-Verlag, 1998.

[13] The JoCaml Home Page. *http://pauillac.inria.fr/jocaml*.

[14] L. Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, Faculty of Science, University of Porto, Portugal, 1999. Available from: *http://www.ncc.up.pt/∽lblopes/*

[15] L. Lopes, F. Silva, and V. Vasconcelos. A Virtual Machine for the TyCO Process Calculus. In *PPDP'99*, *LNCS* 1702, 244–260. Springer-Verlag, 1999.

[16] M. Mehl, R. Scheidhauer, and C. Schulte. An Abstract Machine for Oz. Technical report, German Research Center for Artificial Intelligence (DFKI), 1995.

[17] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.

[18] B. Pierce and D. Turner. Pict: A Programming Language Based on the Pi-Calculus. TR CSCI 476, Computer Science Department, Indiana University, 1997.

[19] J. Riely and M. Hennessy. A Typed Language for Distributed Mobile Processes. In *POPL'98*, ACM Press, 1998.

[20] P. Sewell, P. Wojciechowski, and B. Pierce. Location Independence for Mobile Agents. In *Workshop on Internet Programming Languages*, 1998.

[21] F. Silva, H. Paulino, and L. Lopes. Di_pSystem: a Parallel Programming System for Distributed Memory Architectures. In *EuroPVM/MPI'99*, *LNCS* 1697, 525–532. Springer-Verlag, 1999.

[22] V. Vasconcelos and R. Bastos. Core-TyCO - The Language Definition. TR-98-3, DI / FCUL, 1998.

[23] V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *HLCL'98*, *ENTCS* 16(3), 19–34, Elsevier Science, 1998.

[24] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *ISOTAS'93*, *LNCS* 742, 460–474, Springer-Verlag, 1993.

[25] J. Vitek and G. Castagna. Seal: A Framework for Secure Mobile Computations. In *Workshop on Internet Programming Languages*, 1999.