Journal of Universal Computer Science, vol. 9, no. 8 (2003), 745-760
submitted: 24/2/03, accepted: 30/5/03, appeared: 28/8/03 © J.UCS

Distributed Typed Concurrent Objects: a Programming Language for Distributed Computations with Mobile Resources

Álvaro Reis Figueira
(DCC-FC & LIACC. Universidade do Porto
arf@ncc.up.pt)

Hervé Paulino
(Departamento de Informatica, Faculdade de Ciencias e Tecnologia,
Universidade Nova de Lisboa
herve@di.fct.unl.pt)

Luís Lopes
(DCC-FC & LIACC, Universidade do Porto
lblopes@ncc.up.pt)

Fernando Silva
(DCC-FC & LIACC, Universidade do Porto
fds@ncc.up.pt)

Abstract: We describe a programming language for distributed computations that supports mobile resources and is based on a process calculus. The syntax, semantics and implementation of the language are presented with a focus on the novel model of computation.

Key Words: Process-Calculus, Distributed Computing, Mobile Resources
Category: D.1.3, D.3.2

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. Research on languages and run-times that support mobile resources 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 asynchronous π-calculus [Honda and Tokoro, 1991, Milner et al., 1992] is commonly used as the base model for concurrent distributed communicating systems and provides a robust theoretical framework upon which researchers can build solid applications. The main abstractions in the π-calculus are *processes*, representing arbitrary computations and, *channels*, representing places where processes synchronize and exchange data.

Recent extensions of these models with *locations*, representing places in a network where processes evolve, are allowing scholars to glimpse the complexity of distributed systems with mobile resources - see [Cardelli and Gordon, 1998] and [Fournet et al., 1996, Vasconcelos et al., 1998]. Underlying these models is the general concept of *mobility*, that is, the ability for resources to dynamically change their location or access rights, as the system evolves. Mobility comes in two flavors: *weak mobility*, meaning code movement between locations, and; *strong mobility*, meaning that entire computations move through the network of locations [Fuggetta et al., 1998].

The main advantages of using process-calculi for the development of distributed systems are: (a) the calculi provide a natural programming model as their main abstractions deal with the notions of communication and distribution; (b) their semantics are well understood thus significantly diminishing the usual gap between the semantics of the language and that of its implementation, and; (c) they are scalable in the sense that high-level constructs can be readily obtained from encodings in the base calculus.

As may be deduced from the above comments, our approach is distinct from other systems using CORBA [CORBA, 1995], DCOM [COM, 1995] or Java/RMI, although we share some of the goals. We are mainly interested in: (a) developing systems that are provably correct, with simple, well defined semantics, and; (b) we want computations that are *network aware*, i.e, resources can be local or remote and the distinction is explicit in the syntax. In other words we do not adopt the view that the system should provide the elusion of a single, local, address space. Locations should be part of the language abstractions.

The remainder of the paper is organized as follows. The next section briefly introduces the Distributed TyCO process calculus, its syntax and semantics. Section 3 describes the programming language syntax and, section 4 describes the run-time system. The paper ends with the conclusions and references for future work.

2 Distributed Typed Concurrent Objects

The programming language we present in this paper is called Distributed Typed Concurrent Objects (DiTyCO) [Vasconcelos et al., 1998]. The language is an implementation of a process calculus in the lines of the asynchronous π-calculus. The main abstractions of the (centralized portion) of the calculus are channels (a kind of mailbox), objects (collections of methods that wait for incoming messages at channels) and asynchronous messages (method invocations targeted to objects held in channels). It is also possible to use process definitions, parameterized on a set of variables, that may be instantiated anywhere in the program (this allows for unbounded behavior).

The abstract syntax for the (centralized) core language is the following:

$P ::= \mathbf{0}$	terminated process
$\quad\mid\; P \mid P$	concurrent composition
$\quad\mid\; \textsf{new}\ x\ P$	new local variable
$\quad\mid\; x!l[\tilde{v}]$	asynchronous message
$\quad\mid\; x?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$	object
$\quad\mid\; \textsf{def}\ X_1(\tilde{x}_1) = P_1\ \ldots\ X_n(\tilde{x}_n) = P_n\ \textsf{in}\ P$	definition
$\quad\mid\; X[\tilde{v}]$	instantiation
$\quad\mid\; \textsf{if}\ v\ \textsf{then}\ P\ \textsf{else}\ Q$	conditional execution

where x represents a variable, v a value (a variable or a channel), X a process definition and, l a method label.

From an operational point of view, centralized DiTyCO computations evolve for two reasons: object-message reduction (i.e., the execution of a method in an object in response to the reception of a message) and, instantiation of definitions. These actions can be described more precisely as follows:

$$x?\{\ldots, l(\tilde{x}) = P, \ldots\} \mid x!l[\tilde{v}] \rightarrow \{\tilde{v}/\tilde{x}\}P$$

Here, the message $x!l[\tilde{v}]$ targeted to channel x, invokes the method l in an object held in the same channel, denoted as $x?\{\ldots, l(\tilde{x}) = P, \ldots\}$. The result is the body of the method, P, running with the parameters $\tilde{x}$ substituted by the arguments $\tilde{v}$. For instantiations we have something very similar:

$$\textsf{def}\ \ldots\ X(\tilde{x}) = P\ \ldots\ \textsf{in}\ X[\tilde{v}] \mid Q \rightarrow \textsf{def}\ \ldots\ X(\tilde{x}) = P\ \ldots\ \textsf{in}\ \{\tilde{v}/\tilde{x}\}P \mid Q$$

Here, a new instance $X[\tilde{v}]$ of the definition X is created. The result is a new process with the same body, P, as the definition but with the parameters $\tilde{x}$ substituted for the arguments $\tilde{v}$ given in the instantiation.

This kernel language constitutes a kind of assembly language upon which higher level programming abstractions can be implemented as derived constructs. For example, we may implement a derived construct for synchronous method calls in the following way. First, a synchronous method call involves sending an extra argument, r, to the call that works as a "reply" channel. The method, upon completing its execution will send back an acknowledgement message to this channel. Thus, after invoking the method on an object at channel x we must setup an object at channel r to wait for the reply and then continue with Q. This can be written as:

$$\textsf{new}\ y\ x!l[\tilde{v}\ y] \mid y?\{ack() = Q\}$$

This process runs in parallel with:

$$x?\{\ldots, l(\tilde{x}\ z) = P \mid z!ack[], \ldots\}$$

to yield:

$$\begin{aligned}
&\textsf{new}\ y\ x!l[\tilde{v}\ y] \mid y?\{ack() = Q\} \mid x?\{\ldots, l(\tilde{x}\ z) = P \mid z!ack[], \ldots\} \equiv \text{(commut.)}\\
&\textsf{new}\ y\ y?\{ack() = Q\} \mid x!l[\tilde{v}\ y] \mid x?\{\ldots, l(\tilde{x}\ z) = P \mid z!ack[], \ldots\} \to \text{(reduction)}\\
&\textsf{new}\ y\ y?\{ack() = Q\} \mid \{\tilde{v}\ y/\tilde{x}\ z\}P \mid y!ack[] \equiv \text{(commutativity of } \mid \text{)}\\
&\textsf{new}\ y\ y?\{ack() = Q\} \mid y!ack[] \mid \{\tilde{v}\ y/\tilde{x}\ z\}P \to \text{(reduction at } y\text{)}\\
&\textsf{new}\ y\ Q \mid \{\tilde{v}\ y/\tilde{x}\ z\}P
\end{aligned}$$

Hence, process Q only runs *after* the method l is invoked in an object located at channel x. Therefore, we may define a new operator for synchronous method invocation **call** using the base language:

$$\textsf{call}\ x!l[\tilde{v}]\ \textsf{in}\ P \equiv \textsf{new}\ y\ x!l[\tilde{v}y] \mid y?\{ack() = P\}$$

The full calculus grows from the centralized version by adding a new layer of abstraction representing a network of *locations*, identified by names s, where processes are running.

N	$::=$	$\mathbf{0}$	terminated network
	$\mid$	$N \parallel N$	concurrent composition
	$\mid$	**new** $x@s\ N$	new local variable
	$\mid$	**def** $D@s$ **in** N	definition
	$\mid$	$s[P]$	location with running process

This additional layer does not, however, introduce new reduction operations in the calculus. In fact, reduction can only be performed locally at locations and they remain either communications or instantiations as described above.

As can be observed from the above syntax, all resources are lexically bound to the locations they are created on. Thus, a message or object located at some channel $x@s$ must first move to location s to in order to reduce. Similarly, an instantiation of a definition $X@s$ must move to location s in order to reduce.

The lexical scope on resources together with the requirement of local reduction induce the following rules for resource migration:

(Message Migration)

$$r[x@s!l[\tilde{v}]] \to s[x!l[\tilde{v}\sigma_{rs}]]$$

(Object Migration)

$$r[x@s?M] \to s[x?M\sigma_{rs}]$$

(Remote Instantiation)

$$\textsf{\textbf{def}}\ X@s(\tilde{x}) = P\ \textsf{\textbf{in}}\ r[X@s[\tilde{v}]] \to \textsf{\textbf{def}}\ X@s(\tilde{x}) = P\ \textsf{\textbf{in}}\ s[X[\tilde{v}\sigma_{rs}]]$$

To preserve the lexical bindings of resources, every time one moves to another location, all its free identifiers (references for resources it uses) are translated on-the-fly. This is represented here by the transformation σ_{rs} meaning "translation of identifiers when moving from location r to location s". More formally, σ_{rs} is defined as:

$$\sigma_{rs}(x@s) = x \qquad \sigma_{rs}(x) = x@r \qquad \sigma_{rs}(v) = v$$

where, the last case is always applied last.

One final word is required on: (a) lexical scope, and; (b) local reduction, since they are ultimately design goals for the language.

Lexical scope is an important property since it provides the compiler and run-time system with important information on the origin of a resource. This is important namely for safety reasons (e.g., does the resource come from a trusted location ?) and for implementation reasons (e.g., where do we allocate the data-structures for it ? Do they move around in the network ?).

Local reduction is also of the utmost importance. In our model, client-server interactions for example occur within a location. This is in contrast with the standard client-server model where interactions require maintaining remote sessions open and the exchange of many messages drastically reducing the available bandwidth of a network. In the novel paradigms for Web Computing [Fuggetta et al., 1998], client applications move to server locations where they interact through local sessions. They return to their original location after the local session is complete.

3 Programming in DiTyCO

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

export x P	**import** x **from** s **in** P
export $X(\tilde{x}) = Q$ **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. The semantics associated with imported channels or definitions is, as we have seen, *code shipping*. 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.

$$
\begin{aligned}
[\![s[\textbf{export}\ x\ P] \parallel N]\!] &\stackrel{\text{def}}{=} \textbf{new}\ x@s(s[[\![P]\!]] \parallel [\![N]\!]) \\
[\![s[\textbf{export}\ X(\tilde{x}) = Q\ \textbf{in}\ P] \parallel N]\!] &\stackrel{\text{def}}{=} \textbf{def}\ X@s(\tilde{x}) = Q\ \textbf{in}\ (s[[\![P]\!]] \parallel N) \\
[\![\textbf{import}\ x\ \textbf{from}\ s\ \textbf{in}\ P]\!] &\stackrel{\text{def}}{=} [\![P\{x@s/x\}]\!] \\
[\![\textbf{import}\ X\ \textbf{from}\ s\ \textbf{in}\ P]\!] &\stackrel{\text{def}}{=} [\![P\{X@s/X\}]\!]
\end{aligned}
$$

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

The first example defines an AppletServer implemented 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 channel p lexically bound to the client site. Each client creates a fresh channel where the applet server is supposed to locate the applet, then invokes the server with this channel and, in parallel, triggers the applet. The program source code is presented in figure 1.

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

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

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

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

Notice that the applet body gets translated to reflect its new site: if P refers to a channel x local to the applet server, then $\mathsf{P}\sigma_{\mathsf{server\ client}}$ refers to the remote

Server	**Client**
def AppletServer (self) =	**import** appletserver **from** Server
self ? {	**in new** p appletserver!applet$_j$[p] |
applet$_1$(p) =	p![v]
p?(x)= P$_1$ | AppletServer[self]	
...	
applet$_k$(p) =	
p?(x)= P$_k$ | AppletServer[self]	
}	
in export appletserver	
in AppletServer[appletserver]	

Figure 1: *Code for the Applet Server example.*

channel x@server. Note that the message appletserver!applet$_j$[p@site] migrates the code to a site site; thus, clients may download the applet to any site.

The second example, inspired in the SETI (Search for Extra-Terrestrial Intelligence) program. Seti@home was developed by the SETI managers as a way to deal with the vast computational power required to process data obtained by the program's radio-telescopes. The DiTyCO program is described in figure 2.

This program introduces a new concept, the uploading of DiTyCO definitions. The concept allows programs to create instances of remote definitions. The client site may supply local channels as arguments (Install@setiServer[handle]) and waits for the server site to instantiate the definition and ship back the code. In the example we have, translating the import/export rules:

def Install@setiServer(self)= ... Go@setiServer(self)= ... **in**
setiServer [**def** Database(self)= ... **in** Database[database]] ||
setiClient[**new** handle Install@setiServer[handle]]

The client creates a fresh channel that is passed as an argument to the Install definition located at the server. As it is passed as argument, handle's location must be known to the server, thus obliging the channel to become known to the entire DiTyCO network under the identifier handle@setiClient (in the base calculus, a set of structural congruence rules makes this possible). The client runs Install[handle@setiClient], detects that the definition for Install is at server and ships the instantiation to that location.

After the instantiation is received at the server, one local reduction creates a new instance of Install. As usual, the transformation σ is applied to all free

Seti Server	Seti Client

new database
def
  Database(self) = self ? {
    newData(data) = ...
    newChunk(replyTo) = ...
  }
in
export
  Install(self) = self ? {
    ⟨install⟩; Go[self]
  }
  Go(self) = self ? {
    let data = database!newChunk[]
    in ⟨process⟩; Go[self]
  }
in Database[database]
	import Install **from** setiServer
	in new handle Install[handle]

Figure 2: *Code for the SETI@home example.*

identifiers moving through the network.

def Install@setiServer(self)= ... Go@setiServer(self)= ... **in**
new handle@setiClient
setServer[...] ||
setiClient[⟨install⟩$\sigma_{\text{setiServer setiClient}}$; Go[handle]]

Now the installation procedure can run and configure all that is necessary for the program to start. Once this is done the remaining code requires a new instantiation of a remote definition, Go, and therefore the uploading process is repeated. The resulting code contains the newChunk[] method invocation on the remote channel database.

def Install@setiServer(self)= ... Go@setiServer(self)= ... **in**
new handle@setiClient
setiServer[...] ||
setiClient[**let** data = database@setiServer!newChunk[] **in**
⟨process⟩$\sigma_{\text{setiServer setiClient}}$; Go[handle]]

Next, at the client, a new channel data@setiClient is created and the method newChunk at the server database is called with it. Once the method newChunk is executed, it places a data block at data@setiClient, that will be processed by the code in ⟨process⟩.

def Install@setiServer(self)= ... Go@setiServer(self)= ... **in**
new handle@setiClient, data@setiClient
setiServer[...] ||
setiClient[⟨process⟩$\sigma_{\mathsf{setiServer\ setiClient}}$; Go[handle]]

After the local execution of ⟨process⟩ the program then loops endlessly, fetching and processing new data chunks.

4 The Implementation of Distributed TyCO

In this section we describe the architecture of the DiTyCO run-time system, its basic functioning and how it evolved from the TyCO virtual machine (developed for the TyCO calculus [Lopes et al., 1999]).

The Software Architecture

Despite maintaining a logical organization in the form of a flat network topology, the implementation of DiTyCO has three levels: **sites**, **nodes** and **network**. 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 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. Resource mobility occurs between sites, with the network and the nodes being simply service providers. This structure is illustrated in figure 3.

The Network provides a global name service to sites. Explicitly exported identifiers, as well as site names are registered in a *Network Name Service* (NNS). Conceptually, the service maintains two tables, one for sites (*SiteTable*) and another for exported identifiers (*Export Table*). 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.

SiteTable: SiteName $\mapsto$ SiteId $\times$ IpAddress

The key for the *Export Table* 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.

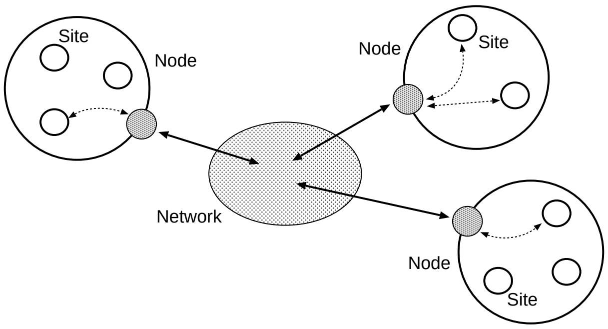

Figure 3: *DiTyCO Architecture.*

ExportTable: SiteName$\times$IdName $\mapsto$ HeapId

The network address for an identifier – (IP,SiteId,HeapId) – is composed by its unique HeapId, the site identifier, SiteId, and its IP location.

Nodes are composed of a pool of sites running concurrently plus a proxy for communication. There is one DiTyCO node per IP node. This architecture is illustrated in figure 4.

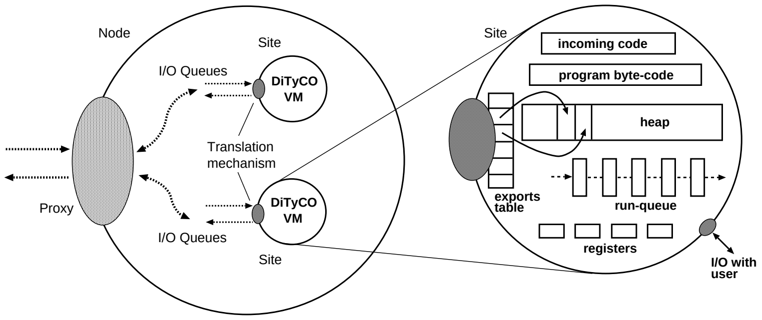

Figure 4: *The Node and Site architecture.*

A DiTyCO node is implemented as a Unix process, which launches threads for handling inter-node communications, and sites for executing DiTyCO programs. The threads share the address space of the node. New sites are created when a new program is submitted for execution and destroyed when the program finishes.

The node's proxy 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:

- the site places a packaged resource with a destination network reference in the outgoing-queue;
- the local proxy gets the resource from the queue, gets the destination IP from the network reference and sends the resource to the proxy of the node where the remote site is located, and finally;
- the remote proxy takes the resource and places it in the incoming-queue of the remote site where it will be picked up by the local DiTyCO Virtual Machine.

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

Sites are implemented as threads. They support resource mobility by extending the single-threaded TyCO virtual machine [Lopes et al., 1999].

The TyCO virtual machine is a compact register based machine that features: a *program area* where the code is kept; a *heap area* where dynamic data-structures are allocated, and; a *run-queue* to keep executable tasks and their corresponding environment bindings. The programs are compiled into an intermediate assembly code, which in turn is compiled into hardware independent byte-code. The mapping between the assembly and the final byte-code is almost one-to-one. Nested structures in the source program are preserved in the final byte-code to allow the efficient dynamic selection of byte-code blocks that have to move between sites. This design has proved to be quite compact and efficient when compared with related languages such as Pict [Pierce and Turner, 1997], Oz [Mehl et al., 1995] and Join/JoCaml [*JC-team*, 1999].

In Distributed TyCO the basic virtual machine (figure 4) is extended in the following ways to support resource mobility:

- **References.** Internal references in the virtual machine may now hold *local references* or *network references*. A local reference is a reference for a frame in the heap of the local site. A network reference, on the other hand, is "a reference" 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.

  The mapping between local and network references is done per-site with the help of an *export table*. This table maps local heap references into its corresponding network address. When a resource moves to a remote site, the free variables in the resource 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 the 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.

- **Instructions.** Some instructions must be added to the virtual machine whereas some others must have their semantics changed. First, new instructions are required to implement the constructs **import** and **export**, that interact with the NNS via the local proxy. Second, the instructions for processing code resources such as: objects, messages and instantiations must be changed to account for the possibility that these may be associated with remote channels or definitions. When processing and object, say, the system dynamically checks whether its channel is a local reference (in which case the usual reduction procedure will apply) or if it is a network reference. In this last case, the machine will pack the byte-code for the object and its free variable bindings and, via its node proxy, will send it to a remote node where it will be integrated in the destination site. The semantics for processing messages and instantiations is similar, always involving: (a) checking the reference for the channel or definition; (b) performing local reduction or resource movement accordingly.
- **Data-Structures.** We add two queues for mobile resources to the basic TyCO virtual machine to allow the exchange of information between the site and the local proxy. These require mutual exclusive access.

These small changes in the TyCO virtual machine allow the full implementation of the functionality of sites.

Weak Mobility

Weak mobility refers to the movement of stateless computational components in a network. In this section we describe with more detail how every aspect of the system's architecture is involved in the movement of these resources.

Let us assume we are executing a program at site r and an object is to be placed at channel which lies in site s. Then, according to the semantics of the system, this object must move to site s. Therefore, site r builds an inter-node message which contains that object and puts it in the node's outgoing queue, to be picked by the node's proxy and be sent to the node that holds site s. This inter-node message is built based on: (a) the frame, holding the object's free identifiers; (b) the object's byte-code; (c) some book-keeping information.

Just before being put on the node's outgoing queue, the frame part has its free identifiers translated, according to the local export table, such that local identifiers are transformed into network addresses. All those steps are exemplified in figure 5.

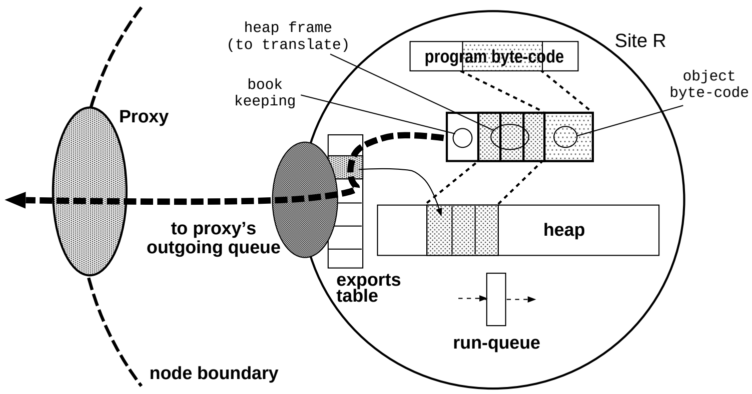

Figure 5: *Sending an object.*

The local proxy then picks this internal message, prepares it for inter-node communication and transmits it to the proper node where site s is located. Eventually, if the message is bigger than a specified maximum size, according to the DiTyCO inter-node transmition protocol, it has to be split into several packets which will be transmitted in order.

Upon reception on the other node by the local proxy, the message is re-assembled (if it arrived fragmented), and is placed in the incoming queue of site s. After the last DiTyCO-thread that was in execution in site s has been processed, all pending messages in the site's incoming queue are read and processed.

Now, the frame part of the message is retranslated with the help of the local export table, to reflect the new location, and is placed on the local heap. The byte-code for the object, also part of the message, is placed in the *remote byte-code area*. At this point the object can be processed locally. These operations are illustrated in figure 6.

The case of migrating messages is simpler since there is no byte-code to migrate. Only the message arguments need to be translated. The remote in-stantiation of definitions is similar to the message case (we only translate the arguments for the instantiation).

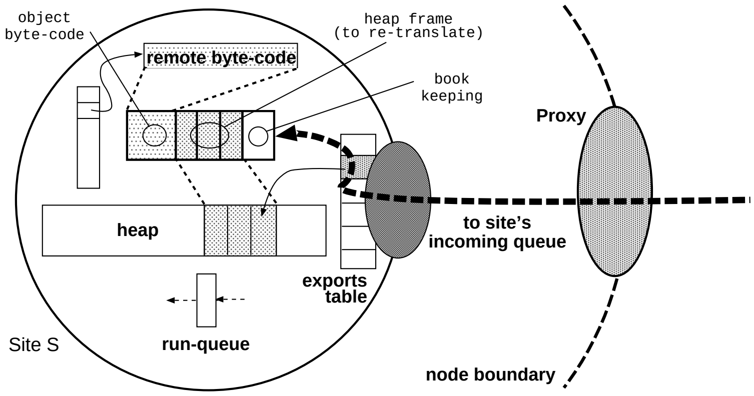

Figure 6: *Receiving an object.*

Strong Mobility

The term *strong mobility* denotes the movement of full computations (code and state) from one place to another in a network. In Distributed TyCO, this process is *subjective*, that is, a site decides on its own when it moves to another location in the network. This is accomplished, at the programming language level, through the inclusion of a new primitive **go**, written:

go newName[@newLocation]

If the right hand side is not present in the argument then, the system assumes that it is just a change in the site's name and only an update in the NNS is issued. If we do have a location in the argument then the computation is suspended at the local site and moves to the other machine where the site is recreated under the new name and restarted. This operation is asynchronous and the NNS is also updated with the new location of the site. The steps in a **go** operation can be summarized as follows:

- Original Location:
  1. suspend execution of site;
  2. pack code and state;
- Network:
  1. packaged computation moves between proxies;
- Destination Location:
  1. unpack the site;

2. resume execution of site;

3. update the site's new location in NNS.

At the virtual machine level, strong mobility is supported by providing methods that gather all components of the state of the computation into a package ready to be moved to the remote site through mediation of the local proxy.

The nodes themselves see their functionalities increased since they now may send and receive packed sites. When receiving such a packed site the node, after doing some checks, must unpack it and restore its execution.

At the network level, strong mobility requires the use of the NNS to keep track of the sites as they move through the network and to redirect any mobile resources destined to these moving sites. Figure 7 presents an example of how the NNS helps with resource mobility under a dynamically changing network topology. First, at (1), site A moves to a new node and (2) the proxy updates

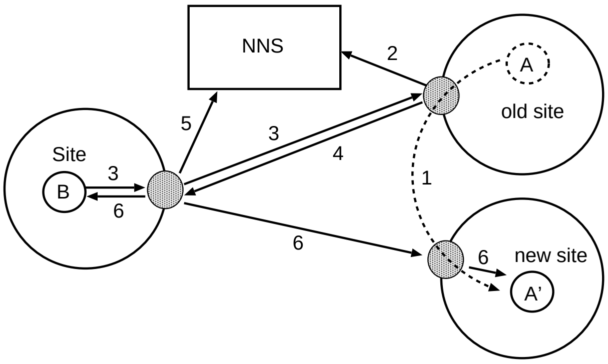

Figure 7: *Message Delivery Mechanism.*

the NNS information for the new location of the site. When another site B tries to move a resource to the old location (3) we have a problem. Remember that some channels in this site may be lexically bound to site A but are not aware that the site has moved. The result of such an attempt is that the message is returned to the proxy of site B with an error code (4), indicating that the site is no longer at the original location. The proxy of node B then gets the new site location from the NNS (5) and resends the message to the new location (6). It also sends a message to site B to update its information on the location of site A. In this way, the topology of the computation is maintained dynamically in the network wide service provided by the NNS.

5 Conclusions and Future Work

We have introduced a programming and execution model for distributed computations with support for resource mobility that is both intuitive and, we feel, provides adequate abstractions for coding distributed applications. The model is based on a process calculus framework which makes it amenable to formal verification.

Currently, the first DiTyCO prototype with support for both weak and strong mobility is in the final stages of the implementation. We have not dealt with important issues such as fault-tolerance, termination detection and security since we are still at an early stage in this work. The above mentioned problems are quite difficult in a distributed setting and will be the focus of future work.

Acknowledgements
The work described in this paper was supported by projects MIMO and Mikado, references POSI/CHS/39789/2001 and IST-2001-32222, respectively.

References

[Cardelli and Gordon, 1998] Cardelli, L. and Gordon, A. (1998). Mobile Ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'98)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag.

[COM, 1995] COM (1995). The COM Specification. The Microsoft COM home page. `http://www.microsoft.com/com/default`

[CORBA, 1995] CORBA (1995). The Common Object Request Broker: Architecture and Specification, Revision 2.0. `http://www.omg.org/corba/corbiiop.htm`

[Fournet et al., 1996] Fournet, C., Gonthier, G., and et al. (1996). A Calculus of Mobile Agents. In *International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag.

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

[Honda and Tokoro, 1991] Honda, K. and Tokoro, M. (1991). An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag.

[*JC-team*, 1999] *JC-team* (1999). The JoCaml System home page. `http://pauillac.inria.fr/jocaml`

[Lopes et al., 1999] Lopes, L., Silva, F., and Vasconcelos, V. (1999). A Virtual Machine for the TyCO Process Calculus. In *Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 244–260. Springer-Verlag.

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

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

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

[Vasconcelos et al., 1998] Vasconcelos, V., Lopes, L., and Silva, F. (1998). Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science.