

Álvaro Pedro de Barros Borges Reis Figueira

**Design and Implementation of a
Distributed System with Mobility
based on a Process Calculus**



Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2004

Álvaro Pedro de Barros Borges Reis Figueira

Design and Implementation of a Distributed System with Mobility based on a Process Calculus



*Tese submetida à Faculdade de Ciências da
Universidade do Porto para obtenção do grau de Doutor
em Ciência de Computadores*

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

2004

Agradecimentos

Quero expressar os meus agradecimentos ao Fernando, meu orientador “oficial”, e ao Luís, meu orientador “não-oficial”, a grande entrega e disponibilidade de ambos para seguirem o meu trabalho, por me proporem alternativas válidas, pelas várias leituras e pelas incansáveis correcções da tese. A ambos o meu muito obrigado.

A conclusão deste trabalho, no entanto, só foi possível pelo apoio que tive no meu núcleo familiar. A Carmen e o meus Pais foram o suporte que necessitei. Foram a força, a convicção e a motivação que permitiram que isto continuasse e, sobretudo, que terminasse. Esta tese é muito mais do que um trabalho científico. É sobretudo uma demonstração de amor e confiança em mim por aqueles que me estão muito próximos.

Queria agradecer igualmente aos meu colegas no Departamento de Ciência de Computadores o seu interesse e preocupação tantas vezes manifestado. Em particular tenho de agradecer ao Luis Filipe, ao Rui Prior, ao Sérgio Crisóstomo, ao Mário e à Ana Paula por terem tornado possível uma redução do meu trabalho docente. Quero igualmente agradecer ao pessoal do 213, Ricardos, Michel e Eduardo, o interesse e a boa disposição! Uma referência importantíssima aos responsáveis pela Administração da Rede, em particular, ao Hugo Ribeiro por estar sempre disponível para dizer as “palavras mágicas”.

Finalmente, à minha restante família por ter esperado. Por me perdoarem estas ausências todas, e por continuarem a gostar de mim.

Desejo também expressar o meu agradecimento à Fundação para a Ciência e Tecnologia, pelo seu apoio na fase inicial deste trabalho, sob a forma de bolsa de doutoramento (PRAXIS XXI/BD/13971/97).

Abstract

The last few years have seen a change in programming paradigms for computer networks. This is a result of an ever growing demand for software support for mobile devices with intermittent connectivity and a high degree of autonomy. One of the main difficulties in developing software for these systems is being able to verify its correctness. Compliance with a sound formal specification is the best way to ensure that the software is safe and will perform as expected.

This thesis addresses the design and implementation of DiTyCO, a programming language and run-time system based on a process calculus with the notions of distribution and resource mobility. The starting point for this work is the TyCO (Typed Concurrent Objects) process calculus which is a form of the asynchronous π -calculus with objects and method invocations interacting on shared communication channels as main abstractions. The use of a process calculus has the advantage of providing a provably correct specification for the implementation of the corresponding language and run-time system.

DiTyCO uses a flat network architecture with three software layers: a network name server to resolve shared network references, a proxy server at each IP node where DiTyCO computations may occur, and multiple virtual machines running DiTyCO programs. The language uses a lexical scope discipline to keep track of resources in the network. Resources such as objects, messages or definitions can move between virtual machines as a result of their lexical scope (weak mobility). Another form of mobility is also implemented that allows running DiTyCO programs to move between IP nodes in the network (strong mobility).

Resumo

Nos últimos anos temos assistido a uma mudança de paradigmas de programação para computação em rede. Tal mudança decorre de uma crescente procura de software para dispositivos móveis que suporte uma conectividade intermitente e um alto grau de autonomia. Uma das maiores dificuldades no desenvolvimento de software para tais sistemas é a possibilidade de se verificar a sua correcção. A melhor forma de garantir que o software produzido seja seguro e execute de acordo com o que dele se espera é seguir uma especificação sólida e formal.

Esta tese trata da especificação e implementação do DiTyCO, uma linguagem de programação e respectivo sistema de execução, baseados num cálculo de processos com as noções de distribuição e mobilidade de recursos. O seu ponto de partida é o cálculo de processos TyCO (Typed Concurrent Objects), uma variante do cálculo π , tendo como abstrações fundamentais objectos e invocações de métodos em canais de comunicação partilhados. O uso de um cálculo de processos facilita a verificação de que a nossa implementação da linguagem e sistema de execução está correcta relativamente à especificação formal.

O sistema DiTyCO usa uma arquitectura plana de rede com três camadas de software: numa camada um servidor de nomes para fazer a correspondência entre endereços lógicos e endereços físicos; numa outra, nós IP distintos, cada um com o seu proxy, suportam múltiplas computações; e na última, múltiplas máquinas virtuais, agrupadas por nó, e executando os programas DiTyCO. A linguagem usa uma disciplina de âmbito léxico como forma de manter o rasto dos recursos na rede. Recursos, tais como objectos, mensagens e definições podem mover-se entre diferentes máquinas virtuais como resultado do seu âmbito léxico (mobilidade fraca). Uma outra forma de mobilidade foi também implementada permitindo, que programas DiTyCO em execução se movam na rede entre nós IP diferentes (mobilidade forte).

Résumé

Dans les dernières années, on a assisté à un changement des paradigmes de programmation pour la computation en réseau. Un tel changement est dû à la croissante demande de software pour dispositifs mobiles capable de supporter une connectivité intermittente et un haut niveau d'autonomie. Une des plus grandes difficultés dans le développement de software pour ces systèmes est la capacité de vérifier sa correction. La meilleure manière de garantir que le software produit sera sûr et exécute ses tâches d'accord avec ce qu'on attend de lui est de suivre une spécification solide et formelle.

Cette dissertation s'occupe de la spécification et de l'implémentation du DiTyCO, qui est un langage de programmation et le correspondant système d'exécution, ayant comme base un calcul de processus avec les notions de distribution et de mobilité des ressources. Son point de départ est le calcul de processus TyCO (Typed Concurrent Objects), une variante du calcul π , ayant comme abstractions fondamentales des objets et des invocations de méthodes en canaux de communication partagés. L'utilisation d'un calcul de processus nous rend plus facile de vérifier si notre implémentation du langage et du système d'exécution est correcte par rapport à la spécification formelle.

Le système DiTyCO utilise une architecture plate de réseau avec trois couches de software : dans une des couches, un serveur de noms pour faire la correspondance entre les adresses logiques et les adresses physiques; dans une autre, des noeuds IP distincts, chacun avec son propre proxy, où se produisent des multiples computations DiTyCO; et dans la dernière couche, où multiples machines virtuelles, groupées par des noeuds, exécutent les programmes DiTyCO. Le langage utilise une discipline de champ d'action lexicale comme façon de maintenir la trace des ressources dans le réseau. Ces ressources, qui sont des objets, des messages et des définitions, peuvent se mouvoir entre différentes machines virtuelles, comme le résultat de son champ d'action lexicale (mobilité faible). Une autre façon de mobilité a été aussi implémentée, permettant

que des programmes DiTyCO en exécution peuvent se mouvoir dans le réseau entre des noeuds IP différents (mobilité forte).

Contents

Abstract	5
Resumo	7
Résumé	9
List of Tables	17
List of Figures	21
1 Introduction	23
1.1 A Novel Model of Computation	23
1.2 Motivation	24
1.3 Towards a Process-Calculus Base	24
1.4 The Asynchronous π -Calculus	27
1.4.1 The π -calculus Syntax	27
1.4.2 Basic Principles of the Operational Semantics	28
1.5 The TyCO-calculus and language	29
1.6 Distribution and Mobility	30
1.7 Goals of this Thesis	31

1.8	Thesis Outline	31
2	Typed Concurrent Objects	33
2.1	Introduction	33
2.2	The TyCO Programming Language	34
2.2.1	Syntax	34
2.2.2	Operational Semantics	35
2.3	Derived Constructs	37
2.4	Examples	38
2.5	Implementation	41
2.5.1	Memory Architecture	41
2.5.2	The Program Area	42
2.5.3	The Heap	42
2.5.4	The Run-Queue	45
2.5.5	Machine Registers	46
2.5.6	The Engine	47
2.5.7	The Garbage Collector	47
2.6	The TyCO Compiler	47
2.6.1	The Instruction Set	49
2.6.2	Basic Compilation Blocks	50
2.6.3	Example Compilations	52
2.7	Chapter Overview	55
3	The DiTyCO Programming Language	57
3.1	Motivation	57

3.2	Introducing DiTyCO	58
3.3	The Calculus and its Semantics	59
3.3.1	Syntax	59
3.3.2	Structural Congruence	60
3.3.3	Reduction and Mobility	62
3.4	The Programming Model	64
3.5	Programming Examples	65
3.5.1	The Bank Account	65
3.5.2	The SETI@Server	67
3.5.3	Network Paradigms	69
3.6	Outline of Changes in the Compiler	70
3.7	Chapter Overview	71
4	The Virtual Machine Implementation	73
4.1	General Description	73
4.2	The Site Layer	75
4.2.1	Memory Architecture	76
4.2.2	The Local Exports Table	76
4.2.3	Representing Names, Strings and Values	78
4.2.4	The Incoming Queue	79
4.2.5	Remote Program Area	79
4.2.6	The I/O Port	79
4.2.7	New instructions	80
4.3	The Node Layer	84

4.3.1	Memory Architecture	84
4.3.2	The DiTyCO shell	85
4.3.3	The DiTyCO interface	87
4.3.4	The Proxy	88
4.3.5	Internal Queues	91
4.4	The Network Layer	92
4.4.1	The Network Name Server	92
4.4.2	Translation of Names	94
4.4.3	Packing and Unpacking for Communication	95
4.4.4	Fragmenting and Assembling Messages	97
4.5	An Example	99
4.6	Chapter Overview	102
5	Mobility	103
5.1	Introduction to Mobility	103
5.2	Mobility in DiTyCO	105
5.3	<i>Shipping</i> versus <i>Fetching</i>	107
5.3.1	Characteristics of <i>ship</i> and <i>fetch</i>	107
5.3.2	Implementation Issues	108
5.4	Weak Mobility	110
5.4.1	The Applet Server Example	110
5.4.2	The Migration Mechanism	112
5.5	Implementing Weak Mobility	113
5.5.1	The Machine View of the Applet Server	113

5.5.2	Initial Setup Phase	114
5.5.3	Local Site to Local Proxy	118
5.5.4	Local Proxy to Remote Proxy	121
5.5.5	At the Server Site	123
5.5.6	Back at the Client	125
5.6	Strong Mobility	126
5.7	Implementing Strong Mobility	127
5.7.1	Packing a site for migration	127
5.7.2	The Site Migration Mechanism	129
5.7.3	Resource Delivery	130
5.8	Chapter Overview	135
6	Related Work	137
6.1	Basic Issues	137
6.2	Survey on the State of the Art	138
6.2.1	The $D\pi$ calculus	139
6.2.2	The π_{ll} -calculus	140
6.2.3	The Distributed Join-calculus	140
6.2.4	The Seal-calculus	143
6.2.5	X-Klaim and Klava	144
6.2.6	Safe Mobile Ambients	145
6.2.7	Nomadic Pict	146
6.2.8	Obliq	147
6.2.9	Agent TCL	148

6.2.10	Telescript	149
6.3	Chapter Overview	150
7	Conclusions	153
7.1	Overview of the Research	153
7.2	Summary of Contributions	155
7.3	Future Work	156
7.4	Final Remarks	157

List of Tables

2.1	The Register set.	46
4.1	The Exports Table.	77

List of Figures

2.1	The architecture of the TyCO virtual machine.	41
2.2	Frames and Words in the Heap	42
2.3	A Channel in the Heap	43
2.4	A Message frame.	43
2.5	Types of Arguments.	44
2.6	An Object frame.	44
2.7	An Instance frame.	45
2.8	A frame for a String.	45
2.9	A Run-Queue frame.	45
3.1	CS, RE and COD programming paradigms.	70
4.1	Communication between sites in a network	74
4.2	Sites building their interface.	75
4.3	The DiTyCO Virtual Machine.	76
4.4	DiTyCO arguments	78
4.5	The Remote Program Area.	80
4.6	A DiTyCO-node in action.	85
4.7	The DiTyCO-node Threads	85

4.8	DT-node Shared Memory	87
4.9	The DiTyCO proxy	88
4.10	Internal Message Queue	92
4.11	A Network Address Representation.	95
4.12	A message translated and packed	96
4.13	An object translated and packed	97
4.14	An object to be sent to another site.	98
4.15	Box holding an object.	98
4.16	The fragmentation procedure of an object in a Box.	99
4.17	A channel and an object are created at Site A	100
4.18	An imported channel, a local channel and a message at Site B	100
4.19	Enqueuing an object at the local channel in Site B	100
4.20	Message frame packed and translated.	101
4.21	Creation of a reduction frame in the run-queue at Site A	101
4.22	Creation of a reduction frame in the run-queue at Site B	102
5.1	Site communication.	112
5.2	Assembly code for the Applet Server example.	114
5.3	Heap configuration at site Server after export and newd.	115
5.4	Heap configuration in site Server after instf.	115
5.5	Assembly code for the definition.	116
5.6	Heap configuration in site Server after instantiation and objf.	117
5.7	Initial Heap configuration stages at site Client	117
5.8	Message packed and translated in Box.	118

5.9	Message reception, reassembling and enqueueing at site Server .	122
5.10	Message unpacked and re-translated at site Server .	123
5.11	Assembly code for the Applet Server example.	124
5.12	Heap at site Client after receiving the applet.	125
5.13	The DiTyCO Virtual Machine.	128
5.14	Different memory areas in a packed site.	130
5.15	Transition diagram for the site state within a node.	131
5.16	Resource Delivery Mechanism.	133
5.17	Example using the resource delivery mechanism.	134

Chapter 1

Introduction

1.1 A Novel Model of Computation

Over the last few years, the increase in speed of both personal computer 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 application in *web* languages, intelligent mobile agents, cryptography, and high-performance computing, to name a few.

The recent availability of distributed systems and applications accessible to the general public has triggered the interest of end users, markets, and the industry, motivating the development of new classes of applications. The pervasiveness and ubiquity of computer processing and communication has changed the way networks are exploited. The role of the Internet (and networks in general) is progressively changing from a plain communication media to a large distributed platform where innovative and unforeseen services are evermore supported.

The widespread of portable computers such as notebooks, palm-tops, PDAs and even mobile phones gave birth to an additional demand for transference of computations from one place to another, to resume computations in another machine, to reestablish its bindings when reconnected to the network and then resume remote communications. Most of all, nowadays internet computations need several forms of *code mobility*.

1.2 Motivation

In parallel with the emergence of programming for the *world-wide-web* there is the need to provide traditional computing systems with capacities to manage a series of new issues as remote communications, distribution, migration of computations and the additional concurrency raised by the combination of these issues. While, factors like concurrency, distribution and code mobility seem to be highly desirable, the inherent extra complexity introduced hampers the full development of concurrent, distributed and mobile-code systems.

Indeed, maintaining the programming discipline inherited from the imperative paradigms, and simply adding more features into a system (like remote communication, distribution and event-driven actions, taken in the form of concurrency handling), necessarily will lead to an increased error prone system. The essential programming equation, stating that the implementation should be equal to the specification, remains an elusive goal. Programming for the Internet is on one hand an excessive difficult task, and on the other hand only available to heavy-weight systems which, due to their complexity are prone to bugs and semantic errors.

In fact, apart from the Java applets in Web browsers (a rather limited application of code mobility), we can safely say that there are presently very few commercial distributed applications that exploit mobile code in a real-world domain. This lack of evidence hampers the widespread acceptance of the mobile-code technology and poses a fundamental challenging question to the whole research area, on how to surpass this extra complexity in building the demanded applications?

In this thesis, we exploit the characteristics of a new generation of programming languages with built in notions of concurrency, distribution and code mobility, which are based on process-calculi. We focus on the TyCO-calculus and language and propose the *Distributed Typed Concurrent Objects* (DiTyCO) language and run-time system.

1.3 Towards a Process-Calculus Base

Being able to analyze, verify and prove correct the most critical part of a system is a natural wishful goal for any system developer. Here, *correction* means compliance of

implementation with the specification. Network protocols and distributed computations are highly concurrent. Testing can often be useful to find common errors, but it is not capable of guaranteeing correctness. The verification of a system can, to some extent, be automatically done using tools such as theorem provers. However, this process is a difficult one since the gap between the programming language and the one used by the standard theorem provers is often large. The process is also error-prone and the verification produced usually is not reusable as the system grows.

One apparent solution is to verify the actual code expressed in the used programming language rather than some abstraction of it. Unfortunately, the traditional programming languages (i.e. using the imperative style) are not suitable for formal verification as has been argued, despite being easier to translate into efficient code [35]. Therefore, we should use a programming language with a syntax and semantic rules simple enough, which however, can be efficiently implemented. The, above mentioned characteristics raised the attention of the research community to process-calculi as the basis to develop such languages.

Process calculi offer a small set of operators accompanied by operational semantics, which altogether provide a clean design path for the construction of programming languages. The expressive power of a language can then be increased by adding higher level constructs abstractions such as data-structures, objects and classes. Usually, these idioms are built by adding a layer of convenient syntactic sugar on the top of the small core. Translations of such high-level idioms (like objects, functions and branch structures) have already been given rigorous treatment, for example in [76, 77].

Process calculi denote processes and actions in concurrent systems by using algebraic expressions and algebraic operators. They are built around three basic principles [70]: modeling interaction via communication in terms of message passing rather than shared variables; using a small set of basic primitives to specify behavior of the system; deriving useful algebraic laws for manipulating expressions written using these primitives. However, the development of pure *concurrent languages* remained an unreachable goal until the basis of the concurrency theory, the characterization of concurrent systems and their semantics within a rigorous framework was first established.

Robin Milner gave a major contribution in this field by introducing an algebra called *Calculus of Communicating Systems* (CCS) [58], in which processes interact by syn-

chronizing on shared names, though not exchanging data. The CCS view of modeling concurrency, considering processes as algebra terms and their interactions as operations on these terms constituted the basis of the now commonly called *process calculi*.

The basic principle of CCS is illustrated in the expression below in which processes Q and P synchronize their execution on name a seen as an output from Q and input from P . The consequence of this synchronization, written using the symbol \rightarrow , is the free execution of Q and P independently of each-other.

$$\bar{a}.Q \mid a.P \rightarrow P \mid Q$$

The work pursued by Milner, together with Parrow and Walker, culminated in a new proposed process calculus, named the π -calculus [59, 60, 61]. In its original form, a process could either send or receive a single name, to or from, another process. Therefore, the new capacity of the π -calculus was to transmit data during the CCS-like synchronizations. This was achieved augmenting the prefix of processes as in:

$$\bar{a}(b).Q \mid a(x).P \rightarrow P\{b/x\} \mid Q$$

The expression $\{b/x\}P$ means the process term obtained from P by replacing all free occurrences of x by b , renaming as necessary to avoid collisions. This new calculus constituted the *monadic* π -calculus, meaning that only single names are passed through the communication links between processes. Soon, this minor difficulty was surpassed through a simple encoding transforming the calculus into a *polyadic* form [63]:

$$\bar{a}(\tilde{b}).Q \mid a(\tilde{x}).P \rightarrow P\{\tilde{b}/\tilde{x}\} \mid Q$$

In its original form the π -calculus is called *synchronous* because messages do *block* the execution of processes. This means that in the output process $\bar{a}(\tilde{b}).Q$, process Q suspends until the message $\bar{a}(\tilde{b})$ is read.

Later in the nineties Honda and Tokoro [45] and Boudol [18] presented another version of the polyadic π -calculus which had an equivalent asynchronous formulation. In this form output processes do not carry continuations and thus have the form $\bar{a}(\tilde{b})$. In this form the reduction occurs like this:

$$\bar{a}(\tilde{b}) \mid a(\tilde{x}).P \rightarrow P\{\tilde{b}/\tilde{x}\}$$

The *asynchronous* π -calculus is commonly accepted as the basic calculus for most of the other calculi, systems and research that followed.

1.4 The Asynchronous π -Calculus

The π -calculus has two main abstractions - *processes* and communication *channels* (identified by unique *names*). The calculus allows communication between concurrent processes by synchronizing the output of a process with the input of the other process on the same channel. In network architectures and distributed systems (e.g. as described in [24]), processors share only a communication network. Therefore, the π -calculus *channel* can be seen as an abstraction of the physical communication network, providing the communication path between processes. On the other hand, a *process* here, is a running program, a process (loosely) in the sense of operating systems, consisting of the execution thread together with the respective environment. Communication is accomplished when one process sends a message to a channel and another process receives the message from that channel. The need to specify a channel name denoting the message destination address resembles Internet protocols for message delivery between remote processes lying in different hosts. Despite this situation, in the π -calculus, however, there is no notion of physical machines, as channels are linked directly to processes. The Amoeba [67] distributed operating system provided a closer approach to process calculi models by transmitting messages directly to processes or to communication ports attached to processes, without the need to specify machine hosts.

1.4.1 The π -calculus Syntax

To better describe the syntax, we present the basic grammar. In such, *names* of channels belong to an infinite set, ranged over by x, y . Processes, ranged over by P, Q are defined as:

P, Q	::=	$()$	Terminated process
		$P \mid Q$	Concurrent composition
		$\bar{x}(y)$	Output v on channel x
		$x(y).P$	Input from channel x
		$x^*(y).P$	Replicated input from x
		new x in P	New channel

The term $()$ represents a process that cannot perform any action. The term $P \mid Q$ means that processes P and Q are concurrently executing. A process of the form $\bar{x}(y)$ sends the name y through channel x . The process $x(y).P$ waits to receive a name on channel x , which then is substituted by y in P upon the reception, and continues execution with P . The *restriction operator* **new** x ensures that a fresh channel name x is created. This is particularly useful to avoid name collision even if there is another channel named x as well. Alpha renaming is used in the calculus in bound names to avoid name capture.

In the π -calculus there can be many outputs on the same channel competing for the same input, or many inputs competing for the same output. Despite, this situation, only one will succeed, introducing nondeterminism in the computations. For example, in the following expression, process P can receive either a or b on x .

$$\bar{x}(a) \mid \bar{x}(b) \mid x(y).P$$

The ability of π -calculus names to be communicated between processes, which in turn can be used for communication, allows the creation of systems with evolving connectivity structures.

1.4.2 Basic Principles of the Operational Semantics

According to Milner [58], the operational semantics can be defined in two steps: first by giving a definition of a *structural congruence* relation such that a reducible process can be rearranged to enable reduction; and then, with the definition of a binary *reduction* relation (written \rightarrow) to actually reduce.

This *reduction* relation is fulcral in the π -calculus, as it is in other calculi, for example the lambda-calculus. The expression $P \rightarrow Q$ is read as “ P reduces to Q ”, meaning that P contains (at least) two parallel subprocesses that can communicate on the same channel to become the corresponding subprocesses of Q .

The full description of the π -calculus operational semantics can be found in [64] which includes the *structural congruence* relation, *reduction semantics*, *renaming of bound variables*, *scoping* rules, and *binding* operations .

Many extensions have been proposed to the π -calculus. Probably, the most natural one is to allow tuples of names to be sent (creating the *polyadic* π -calculus), or more general data structures such as booleans, integers and strings. Another important extension was the proposal of a *recursion* operator on a process.

Equipped with this theoretical framework, researchers introduced the notion of observational, or behavioral equivalence. The basic idea is to reason about the behavior of communicating processes, for instance, based on *bisimulation* (as described in [61, 75]). Honda and Tokoro [45], and Amadio, Castellani and Sangiorgi [10] present two different approaches to bisimulation for the asynchronous π -calculus. More on the π -calculus can be found in Milner's book [64] and in the tutorials [63, 77].

1.5 The TyCO-calculus and language

The TyCO-calculus [87, 88] is a version of the asynchronous π -calculus featuring *builtin objects*, *asynchronous messages* and *definitions* as the fundamental entities. TyCO objects are similar to objects in the ζ -calculus [5] in the sense that they are sets of labeled methods. Messages can be interpreted as asynchronous method invocations on the objects. Definitions are processes abstracted on variables allowing, for example, classes to be modeled. TyCO also introduces unbounded behavior which can be modeled through instantiation of recursive definitions. Moreover, the TyCO-calculus provides a clean model for an object-based language with a sound theoretical framework.

Besides these important characteristics, the TyCO-calculus is also the base of the TyCO language, for which an implementation exists [55]. The underlying abstract machine, proven to be *fair*, *sound* and *deadlock-free*, could also be used as the starting point for our implementation.

Such work suited perfectly our objectives of finding a sound basis in which we could elaborate on top, adding the notions of distribution and studying the issues concerning code mobility.

1.6 Distribution and Mobility

The last few years have seen much interest in systems and applications which communicate in a network and using different forms of mobility, such as processes, objects, devices and agents (a collection of papers on each is published in [65]). Nowadays, the support for connecting devices while they physically move, and for processes, objects or agents which visit remote sites and bind to local resources seem to be a normal and undeniable demand. Examples of such needs vary from collaborative applications used in portable computers to maintain communication with a group of people, to the migration of processes to a stable part of the network carrying their computations, or even to the use of mobile agents acting on behalf of users to collect network information which can later be reported.

Traditionally, *process migration* meant the move of an executing process from one CPU to another, for instance to balance the machine load. The early process migration mechanisms were supported by the operating system, as in MOSIX [13] and in Sprite [68], in which processes running in any node could access remote resources indistinctly from local ones. The Condor system [52] proposed an approach in which UNIX processes migrate without the intervention of the kernel. However, only processes that do not use signals or any form of inter-process communication are eligible for migration. Other operating system's supporting process migration as Charlotte [11] and Amoeba [82, 83] provide transparent communication between processes irrespective of their location.

The Emerald object-based language [46] innovated with two important properties in the mobility field: the language introduced support for both the notions of location and mobility, and; the unit of distribution is very fine-grained – the object. The Network Objects distributed programming system [16] enables the transference of objects either by copying, or by reference, despite that they do not move and have no associated thread of control. The Emerald and the Network Objects were probably the precursors of the Java system. The Java programming language and environment [38, 51] and the Java Remote Method Invocation (RMI) [94] offer an environment which is the same in different architectures, due to its universal virtual machine. This allows code to be moved from one virtual machine to another independently of the underlying architecture. The RMI provides objects (i.e. code and associated state) to be passed

from different virtual machines and even pre-compiled classes (which are ensured to be type-safe). However, in Java, it is not possible to move threads of execution.

The strongest form of code mobility is characterized by the *mobile agent* which is an executing computation that can migrate between machines and act independently on behalf of users or other agents. The agent mobility combines features of object migration (encapsulated code and state) and process migration (migration of a thread of execution).

1.7 Goals of this Thesis

This thesis uses the TyCO-calculus and programming language as the starting point to create the Distributed Typed Concurrent Objects (DiTyCO) calculus, language and run-time system.

The main objectives with this work on DiTyCO and its implementation are:

- to formalize a calculus with builtin notions of distribution and mobility;
- to design a concurrent, object-based, programming language in which programs for distributed computations could be easily implemented;
- to create a run-time environment for the language based on communicating virtual machines;
- to provide an implementation of weak and strong mobility in this programming model.

1.8 Thesis Outline

The thesis is structured as follows: chapter 2 describes the TyCO process calculus, the associated programming language, compiler and run-time system; chapter 3 introduces the DiTyCO calculus, its operational semantics and the programming language with examples; chapter 4 describes the architecture and implementation of the DiTyCO system, including the virtual machine, the node service, the name server and the

compiling mechanism; chapter 5 presents a discussion on code mobility and presents DiTyCO's weak and strong mobility features and operating mechanisms; chapter 6 reviews research work that relates directly to the subject of this thesis; finally, chapter 7 elaborates on the main conclusions extracted from the work presented in this thesis and gives future research directions.

Chapter 2

Typed Concurrent Objects

We begin by describing the Typed Concurrent Objects calculus (TyCO), its syntax and operational semantics. Then, we introduce the TyCO programming language [87] and see how new constructs are encoded in the base calculus. In order to ease the understanding of the language, we use some program examples to introduce its main features. We then describe the language's virtual machine and its implementation [55]. Finally, the compilation scheme used is also presented.

2.1 Introduction

The TyCO calculus was initially proposed by Vasconcelos [90] as a form of the asynchronous π -calculus featuring builtin objects, asynchronous messages and process definitions as fundamental abstractions. This calculus describes process interactions by means of communication using asynchronous method invocation and instantiation of process definitions. Objects are ephemeral and unbounded behavior is modeled through explicit instantiation of recursive definitions. Standard communication is asynchronous, however, synchronous communication can be implemented using explicit reply names.

Objects in TyCO are collections of labeled methods and messages are asynchronous method invocations. Both messages and objects are placed in communication channels (actually queues). Whenever an object and a message meet in a channel, reduction takes place.

Lopes et. al. [89] proposed a new programming language and run-time system for the TyCO calculus. The language is implicitly typed, polymorphic, concurrent, and object-based. It is a very low-level kernel language with a syntax close to the calculus itself and with a few derived constructs and primitive data-types. Lopes [55] presented the design and implementation of the TyCO programming language, including a virtual machine to run programs in TyCO. The virtual machine is implemented as a byte-code emulator.

2.2 The TyCO Programming Language

In this section we describe the syntax and semantics of the TyCO process calculus. The static semantics of the calculus is provided by a type-inference system, during compile time, which supports polymorphism in process definitions. The operational semantics is given as usual for process calculi [58] as a set of reduction rules plus a structural congruence relation.

2.2.1 Syntax

We start by describing the syntactic categories used in the definition of the TyCO calculus:

Constants may be booleans, integers, floating-point numbers or strings and are ranged over by c .

Variables are interchangeably used as constants or locations where communication occurs in a given process context, and are ranged over by x, y . Let \tilde{x} denote the sequence $x_1 \cdots x_n$, with $n \geq 0$, of pairwise distinct variables.

Values are either constants or variables, and range over by u, v .

Expressions are builtin logical, algebraic, relational and string operations, ranged over by e . Similarly to variables, \tilde{e} denotes the sequence $e_1 \cdots e_n$, with $n \geq 0$, of expressions. The evaluation of an expression e is a value v .

Process definitions are ranged over by X and identify processes parameterized in a set of variables.

Labels are ranged over by l and are used to select methods within objects.

Processes are the topmost category of the calculus and are ranged over by P, Q .

Their structure is given by the following grammar:

P	$::=$	$\mathbf{0}$	Terminated process
		$P \mid P$	Concurrent composition
		$\mathbf{new} \ x \ P$	New local variable
		$x!l[\tilde{e}]$	Asynchronous message
		$x?M$	Object
		$\mathbf{def} \ D \ \mathbf{in} \ P$	Process definition
		$X[\tilde{e}]$	Instantiation
		$\mathbf{if} \ e \ \mathbf{then} \ P \ \mathbf{else} \ Q$	Conditional execution
M	$::=$	$\{l_1(\tilde{x}_1) = P_1, \dots, l_n(\tilde{x}_n) = P_n\}$	Methods
D	$::=$	$X_1(\tilde{x}_1) = P_1 \ \mathbf{and} \ \dots \ \mathbf{and} \ X_n(\tilde{x}_n) = P_n$	Recursive definition
e	$::=$	$e \ op \ e$	Binary operator
		$op \ e$	Unary operator
		(e)	Expression
		v	Value
op	$::=$	$\mathbf{not} \mid \mathbf{and} \mid \mathbf{or}$	Boolean Operators
		$+ \mid - \mid * \mid /$	Numerical Operators
		$== \mid \backslash = \mid < \mid > \mid <= \mid >=$	Relational Operators
		$\mathbf{cat} \mid \mathbf{len}$	String Operators

2.2.2 Operational Semantics

The variables \tilde{x} are bound in P in a process definition $X(\tilde{x}) = P$, in a method $l(\tilde{x}) = P$ or, in a scope restriction $\mathbf{new} \ \tilde{x} \ P$. The set of *free variables* that occur in a process P , denoted by $\text{fv}(P)$, and the set of *bound variables* occurring in a process P , denoted by $\text{bv}(P)$ are defined accordingly. In TyCO the scope of the **new** instruction extends as far to the right as possible. The substitution of values \tilde{y} for free occurrences of variables \tilde{x} in a process P , $P\{\tilde{y}/\tilde{x}\}$, is only possible if the lengths of \tilde{y} and \tilde{x} are the same. In a process definition, variables are bound by the declaration itself.

As in Milner [58] the computational rules of the calculus are divided in two parts: the *structural congruence* rules and the *reduction* rules. Structural congruence rules allow

the re-writing of processes into semantically equivalent expressions until they may be simplified using reduction rules. The relation \equiv , defined using the following rules, is taken as the smallest congruence relation over processes.

$$\begin{aligned}
[\text{ALPHA}] \quad & P \equiv Q \quad \text{if } P \equiv_\alpha Q \\
[\text{GROUP}] \quad & P \mid \mathbf{0} \equiv P, \quad P \mid Q \equiv Q \mid P, \quad P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
[\text{NEW}] \quad & \mathbf{new } x \mathbf{0} \equiv \mathbf{0}, \\
& \mathbf{new } x \mathbf{new } y P \equiv \mathbf{new } y \mathbf{new } x P, \\
& \mathbf{new } x P \mid Q \equiv \mathbf{new } x (P \mid Q) \quad \text{if } x \notin \text{fv}(Q) \\
[\text{DEF}] \quad & \mathbf{def } D \mathbf{in } \mathbf{0} \equiv \mathbf{0}, \\
& \mathbf{def } D \mathbf{in } \mathbf{new } x P \equiv \mathbf{new } x \mathbf{def } D \mathbf{in } P \quad \text{if } x \notin \text{fv } D \\
& (\mathbf{def } D \mathbf{in } P) \mid Q \equiv \mathbf{def } D \mathbf{in } (P \mid Q) \quad \text{if } \text{ft}(D) \cap \text{ft}(Q) = \emptyset \\
& \mathbf{def } D \mathbf{in } \mathbf{def } D' \mathbf{in } P \equiv \mathbf{def } D \mathbf{and } D' \mathbf{in } P \quad \text{if } \text{dom}(D) \cap \text{ft}(D') = \emptyset \\
[\text{PERM}] \quad & \{l_1(\tilde{x}_1) = P_1, l_2(\tilde{x}_2) = P_2\} \equiv \{l_2(\tilde{x}_2) = P_2, l_1(\tilde{x}_1) = P_1\} \\
& X_1(\tilde{x}_1) = P_1 \mathbf{and } X_2(\tilde{x}_2) = P_2 \equiv X_2(\tilde{x}_2) = P_2 \mathbf{and } X_1(\tilde{x}_1) = P_1
\end{aligned}$$

The computation proceeds either through communication or by instantiation. Communication involves a *redex*, that is, a pair of message and object interacting on a shared channel.

The message $x!l_i[\tilde{v}]$ targeted to channel x , invokes the method l_i in an object $x?\{l_1(\tilde{x}) = P_1, \dots, l_n(\tilde{x}) = P_n\}$ at channel x . The result is the body of the method, P_i , running with the parameters \tilde{x} substituted by the arguments \tilde{v} ($P_i\{\tilde{v}/\tilde{x}\}$). This reduction rule, called *communication*, is written as:

$$\begin{aligned}
& [\text{COMMUNICATION}] \\
& x!l_i[\tilde{v}] \mid x?\{l_1(\tilde{x}_1) = P_1, \dots, l_n(\tilde{x}_n) = P_n\} \rightarrow P_i\{\tilde{v}/\tilde{x}_i\}, \quad 1 \leq i \leq n
\end{aligned}$$

Instantiations of process definitions are another form of reduction. The result of creating the instance $X[\tilde{v}]$ of a definition $X = (\tilde{x})P$ is the process $P\{\tilde{v}/\tilde{x}\}$. This can be written as:

$$\begin{aligned}
& [\text{INSTANTIATION}] \\
& \mathbf{def } X(\tilde{x}) = P \mathbf{and } D \mathbf{in } X[\tilde{v}] \rightarrow \mathbf{def } X(\tilde{x}) = P \mathbf{and } D \mathbf{in } P\{\tilde{v}/\tilde{x}\}
\end{aligned}$$

2.3 Derived Constructs

The TyCO programming language grows directly from the TyCO calculus. It includes all constructors from the calculus and some extra derived constructs to ease programming. All the new constructs can be encoded in the base calculus.

We start by describing the label `val` which can be used in objects with a single method. This allows the programmer to simplify the code for messages and objects that can only refer to a unique method.

$$\begin{array}{lll} x![\tilde{v}] & \text{abbreviates} & x!\mathit{val}[\tilde{v}] \\ x?(\tilde{y}) = P & \text{abbreviates} & x?\{\mathit{val}(\tilde{y}) = P\} \end{array}$$

The construction **if** e **then** P derives directly from the standard calculus construction **if** e **then** P **else** Q by introducing the *terminated process* in the else branch:

$$\mathbf{if } e \mathbf{ then } P \equiv \mathbf{if } e \mathbf{ then } P \mathbf{ else } 0$$

This kernel language constitutes a small assembly language upon which higher level programming abstractions can be implemented as derived constructs [86]. For example, we may implement a derived construct for synchronous method calls as follows. First, a synchronous method call involves sending an extra argument, say y , to the callee to be used as a “reply” channel. The remote method should trigger an acknowledgement message on this channel. Synchronization occurs only when this message reduces at channel y with an object holding the continuation process. The execution then proceeds with the current process P . This can be written as:

$$\mathbf{new } y \ x!l[\tilde{v} \ y] \mid y?\{\mathit{val}() = P\}$$

This process runs in parallel with an object at x :

$$x?\{l(\tilde{x} \ z) = Q \mid z!\mathit{val}[], \dots\}$$

to yield:

new y $x!l[\tilde{v} y] \mid y?\{val() = P\} \mid x?\{l(\tilde{x} z) = Q \mid z!val[], \dots\} \equiv (\text{applying: GROUP})$
new y $y?\{val() = P\} \mid x!l[\tilde{v} y] \mid x?\{l(\tilde{x} z) = Q \mid z!val[], \dots\} \rightarrow (\text{applying: COMM})$
new y $y?\{val() = P\} \mid \{\tilde{v} y/\tilde{x} z\}Q \mid y!val[] \equiv (\text{applying: GROUP, NEW})$
new y $y?\{val() = P\} \mid y!val[] \mid \{\tilde{v} y/\tilde{x} z\}Q \rightarrow (\text{applying: COMM})$
new y $P \mid \{\tilde{v} y/\tilde{x} z\}Q$

Thus, we may define a new operator, **call**, for a synchronous method invocation using the base language:

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

In this equivalence the operational behavior is similar to what we described previously that is, process P is only executed after the message is reduced.

We can extend this example to get a **let** constructor, useful in getting back results from synchronous calls, and whose syntax is taken from Pict [71]:

$$\mathbf{let} \ y = x!l[\tilde{e}] \ \mathbf{in} \ P \equiv \mathbf{new} \ z \ x!l[\tilde{e}y] \mid z?val(y) = P$$

$$\mathbf{let} \ y = X[\tilde{e}] \ \mathbf{in} \ P \equiv \mathbf{new} \ z \ X[\tilde{e}y] \mid z?val(y) = P$$

Finally, there is also a more general pattern matching constructor. The syntax is taken from ML [62]:

$$\mathbf{match} \ x!l[\tilde{e}] \ \mathbf{with} \ M \equiv \mathbf{new} \ y \ x!l[\tilde{e}y] \mid y?M$$

$$\mathbf{match} \ X[\tilde{e}] \ \mathbf{with} \ M \equiv \mathbf{new} \ y \ X[\tilde{e}y] \mid y?M$$

where $M \equiv \{l_1(\tilde{x}_1) = P_1, \dots, l_n(\tilde{x}_n) = P_n\}$

2.4 Examples

In this section we present some examples illustrating the TyCO language syntax and programming style. In the first example we model a cell data-type which can hold a

polymorphic value. The cell object has two methods: `read` to get the state of the cell and `write` to change it.

```
def Cell(self, value) = {
  self ? {
    read (replyTo)  = replyTo ! val [value] | Cell[self, value],
    write(newValue) = Cell[self, newValue]
  } in new x Cell[x,7] | let w = x ! read[] in io ! put[w]
```

In this example, we first create a channel `x` where the cell will be placed; then we instantiate it with value 7; and then the value is read through the invocation of the method `read` which returns the value in `w` and is printed out through the primitive I/O channel of TyCO.

The next example computes the n -th Fibonacci number using the usual recursive definition. We illustrate in this example the use of the `let` constructor.

```
def Fib ( n, replyTo ) =
  if ( n ≥ 2 ) then
    replyTo ! [1]
  else
    let v1 = Fib[n-1]
      v2 = Fib[n-2]
    in replyTo ! [v1+v2]
in
  let v = Fib[30]
  in io ! put[" Fib[30]="] | io ! put[v]
```

The next example implements the Sieve of Erathostenes, for computing prime numbers. All prime numbers found are added to a growing chain, one for each prime found. The sieve filters the numbers as they are generated in `Ints`.

```
def Ints (n, m, sieve) =
  sieve![n] | if n < m then Ints[n+1, m, sieve]
and Sieve(self, myGrain, nextSieve) =
```

```

self ? (n) =
    if (n % myGrain) ≠ 0 then
        nextSieve![n] |
        Sieve[self, myGrain, nextSieve]
and LastSieve(self, myGrain) =
    self ? (n) =
        if (n % myGrain) ≠ 0 then
            io!put[n] |
            new newSieve Sieve[self, myGrain, newSieve] |
            LastSieve[newSieve, n]
        else
            LastSieve[self, myGrain]
in io!put[2] | new sieve Ints[2, 10000, sieve] | LastSieve[sieve, 2]

```

The last example models a bank account that allows a bank client to have access to his account and perform standard banking operations like `withdraw`, `deposit` and `balance`.

```

def Account ( self, balance ) = {
    self ? {
        deposit ( amount ) =
            io ! put[balance+amount] | Account[self,balance+amount],
        balance ( reply ) =
            reply ! write[balance] | Account[self,balance],
        withdraw ( amount ) =
            io ! put[balance-amount] | Account[self,balance-amount]
    }
} in new Account[a, 1000]
new b b ? {write ( x ) = io ! put[x] } | ... | a ! balance[b] | ...

```

In the example, the account is modeled by creating a definition that, when instantiated, places an object in a channel, maintaining the state of the account. Persistency is achieved by recursion in the method invocations. Initially, the balance is set to 1000. The client part is modeled by invoking methods in the object created. To get the answer from the bank, the client creates an object that receives the value of the balance and displays it.

2.5 Implementation

The TyCO virtual machine (Tvm) is the basis of the run-time system of the TyCO programming language. Its design observed the following principles: efficiency, scalability and, functionality to support concurrent programming. Regarding the implementation of the Tvm, it is emulator based, uses a heap for allocating blocks of words called *frames* that represent dynamic data-structures. The program executed by the Tvm is in the form of a byte-code.

Next, we describe in more detail the architecture of the Tvm and its instruction set.

2.5.1 Memory Architecture

The architecture of the Tvm is organized in four main memory areas (see figure 2.1): a *program area* where the byte-code is kept; a contiguous memory area for the *heap* and *run-queue*, growing in opposite directions; an *operand stack*, and; *local name array* to keep track of local names. The memory space for the heap and run-queue is subject to occasional garbage collection operations. The Tvm uses a set of *registers* to keep the state of the execution.

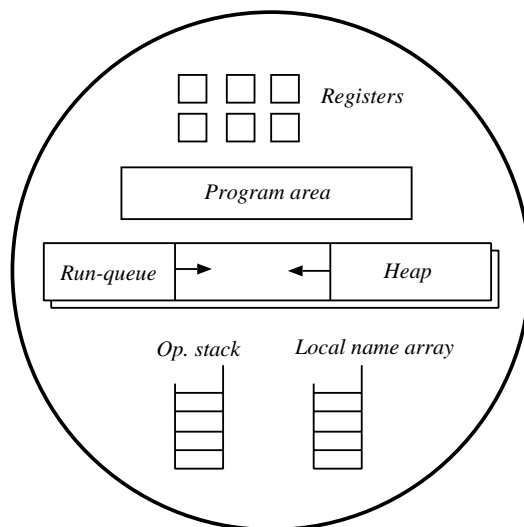


Figure 2.1: The architecture of the TyCO virtual machine.

2.5.2 The Program Area

The Program area hosts the program byte-code to be executed. This area never changes during program execution. Methods of objects are kept in the byte-code and are referred to by using a pointer in the object frame allocated in the heap. Process definitions are also stored in the byte-code and referred from a pointer in the respective heap frame. Access to the program area is done by displacement of the PC register.

2.5.3 The Heap

The heap is an array of 32 bit *words*, used to store dynamic data structures such as channels, messages, objects and instances. During program execution the heap is populated with groups of words, organized as *frames*. Those frames may vary in size. The number of words used in a frame depends on the instruction being executed. A frame in the heap is formed by a special word which we call *the Descriptor* plus a number of heap words according to the structure being represented. The descriptor holds a bitmap that specifies information relative to the frame, namely the size of the frame, the type of frame, and important information for the garbage collector. Figure 2.2 illustrates the organization of frames just described.

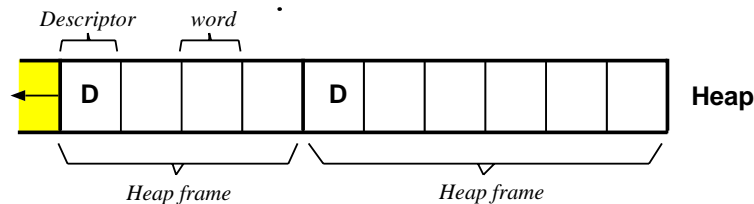


Figure 2.2: Frames and Words in the Heap

The frames kept in the heap are manipulated by specialized instructions and their size is determined by context. The current TyCO implementation has five different heap frames: channels, messages, objects, instances and strings.

Channels are frames that hold communication queues and that implement the base calculus channels. Those frames consist of a descriptor (to hold the frame state) and two pointers that indicate the first and the last item in the queue. The queue at any given moment either is empty or holds only messages or only objects. The definition for the state of a channel is described as: i) empty; ii) with messages; iii) with objects.

This state is kept in the descriptor word, **D**, of the channel frame. Figure 2.3 illustrates a channel in the heap with two resources (objects or messages) enqueued.

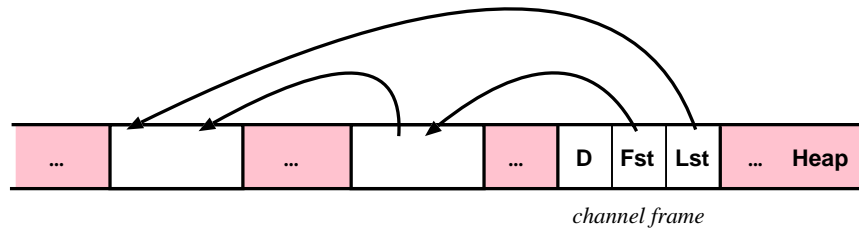


Figure 2.3: A Channel in the Heap

Messages are represented by frames of variable size. Internally their structure is composed by a label name referring the method it is invoking, plus a variable number of arguments as illustrated in figure 2.4. The arguments may be: *channels*, or *primitive values*. Message frames may be queued in a channel using the **Nxt** (next) pointer field. The size of a message frame depends on the number of arguments included in the message.

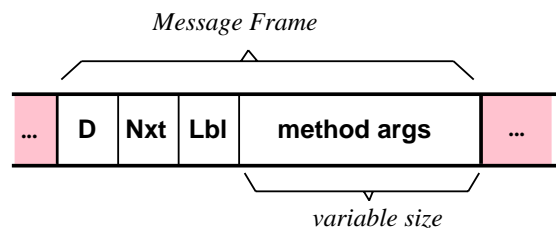


Figure 2.4: A Message frame.

The TyCO implementation places primitive values directly in the arguments within a message frame. For channels and strings these are references to other frames in the heap. To distinguish the types of the arguments in a message frame, bit 0 of the corresponding word must be used as a tag. If the bit is zero, we are in the presence of a value, otherwise we are in the presence of a pointer, either to a channel or to a string.

Heap words with bit 0 set are called *tagged heap words*. These situations are illustrated in figure 2.5. Pointers to channels and strings can be distinguished by their respective

frame descriptor.

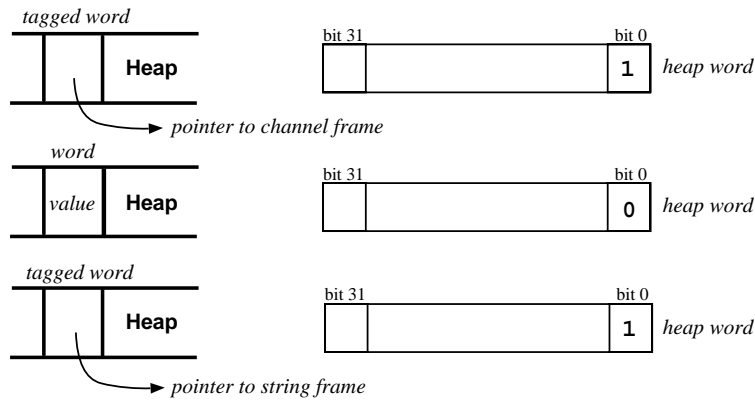


Figure 2.5: Types of Arguments.

Objects are represented in the heap by a frame of variable size, which depends on the number of free variables in the body of the object. Together with the descriptor, the object frame holds a pointer to the byte-code (targeting the location of the method table), plus the variable number of bindings for the free variables that exist in the object's methods. Object methods are accessed through the `Tbl` pointer to the method table, plus an offset given by the message label.

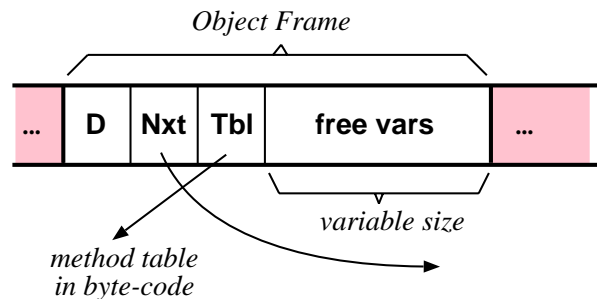


Figure 2.6: An Object frame.

Instances are also stored using a heap frame. This frame holds a descriptor, a reference for the piece of byte-code implementing the definition, and a sequence of arguments. Thus, this frame is of variable size, depending on the number of arguments. Figure 2.7 illustrates an instance heap frame.

Strings need also to be stored in a heap frame. The frame consists of a descriptor and the words necessary to store the string. Figure 2.8 illustrates a string frame in the heap.

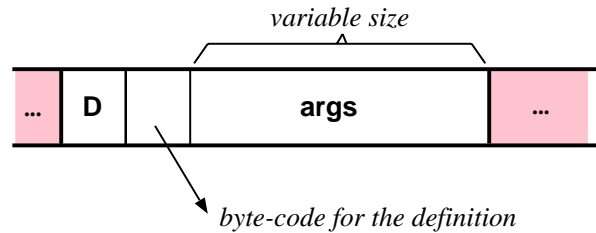


Figure 2.7: An Instance frame.

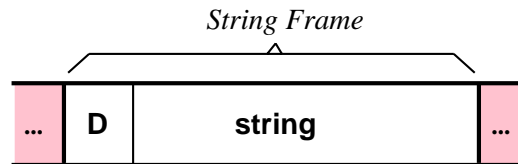


Figure 2.8: A frame for a String.

2.5.4 The Run-Queue

The Run-Queue (RQ) keeps fixed size frames named *tasks*. Tasks correspond to program execution blocks. A RQ frame is 3-word long and is used to hold information on a task and its execution environment. The first word holds a pointer to the next byte-code instruction to be processed; the second, points to the frame holding the parameters and; the third word points to the frame holding the free variables. Figure 2.9 illustrates a RQ frame as just described.

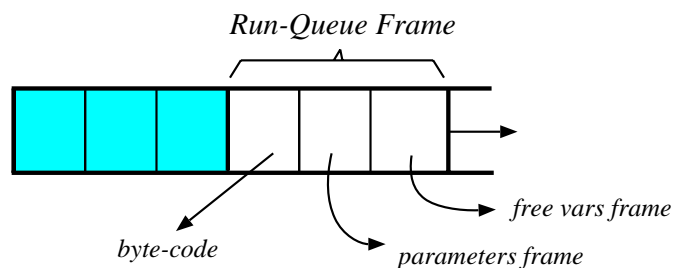


Figure 2.9: A Run-Queue frame.

There are only two situations that lead to a new frame being created in the run-queue: when either communication or instantiation occurs. For both situations the same kind of frame is created in the run-queue: the first word of the frame holds a pointer to the byte-code for the method or definition; the second RQ-frame word holds a pointer to the message frame, and the third RQ-frame word holds a pointer to the object frame,

for the specific case of communication. Globally, for both cases, the second RQ-frame word holds a pointer to the parameters frame, and the third RQ-frame word holds a pointer to the frame holding the free variables.

During program execution new *tasks* are added to the RQ. It is possible to put many *tasks* before the first one gets executed. The RQ is managed through the use of two registers – SQ and EQ – that mark respectively the start and the end of the run-queue.

A new *task* is triggered for execution when the current *task* ends its execution. The last instruction of a thread is responsible for checking if there are more *tasks* to be executed, and in that case, to prepare the next one to be executed.

Since the run-queue and heap grow in opposite directions, it may happen that the system runs out of memory space to allocate new frames. In such a case, the Tvm triggers the *garbage collector*.

2.5.5 Machine Registers

The Tvm uses a set of registers to keep track of the virtual machine state. It uses eleven registers as listed in the table 2.1.

Register	Description
PC	Program Counter
HP	Heap Pointer
SQ	Start of Run Queue
EQ	End of Run Queue
CC	Current Channel
CF	Current Frame
OF	Other redex component Frame
FV	Free Variable bindings
PM	ParaMeter bindings
OS	top of Operand Stack
R	Array of local variables

Table 2.1: The Register set.

Register PC is updated on each instruction being executed to point to the next instruction that must be executed. Register HP is used to mark the first free position in the heap. Registers SQ and EQ mark the start and end of the run-queue respectively. Register CC holds the current channel being used, while register CF holds temporary frames for reduction. Register CF is also used for placement of arguments or free-variables in frames. OF is used as the “other-frame” during reduction. The free variables at some point are located by register FV, and the parameters by register PM. A stack is used for operations on builtins and register OS points the top of this stack. All locally created channels are addressed through an array of registers whose base pointer is given by the register R.

2.5.6 The Engine

The emulator starts with the main task, executing blocks of byte-code instructions. The execution of a task may generate other tasks that are placed in the run-queue waiting for execution. Upon completion of a task, the emulator picks the next one from the run-queue. This procedure loops until there are no more tasks in the RQ to be executed and the system halts.

2.5.7 The Garbage Collector

The TyCO garbage collector implements a copy based mechanism in which there is a fresh memory space equal in size to the heap, where active frames are copied to. A frame is active if it is possible to reach it by following pointers from the frames that exist in the run-queue.

2.6 The TyCO Compiler

The TyCO compilation scheme is based on a two-step procedure in which an intermediate code, actually an assembly code, is produced before final generation of byte-code. The complete compilation procedure is done by using a pipe which takes the output produced by the TyCO compiler into the TyCO assembler. This intermediate step may be used for many operations that include:

- support for visualization and debugging of the programs;
- an extra level of optimizations of the generated code;
- allowing other languages to be compiled directly to the assembly, and then use the TyCO byte-code assembler, to profit from the run-time system;
- as a possible base representation for proof-carrying code, or a typed assembly code.

Byte-code implementations separate the program code from the code to run the programs. This produces a program with a hardware independent representation that can be run by an emulator dependent of its hardware platform. Byte-code implementations are usually slower than their binary counterparts. The overheads are mainly introduced from the emulation mechanism. However, there are substantial advantages for using a byte-code format, which are:

- the byte-code is hardware independent, a very important property for mobile and distributed programs to be able to run on different architectures;
- the byte code is smaller than the correspondent binary code. This is due to the lack of code for the virtual machine;
- possibility to make use of proof carrying code techniques for remote code verification and JIT computation;
- by separating the code for programs from the code to implement the virtual machine, it is possible to do optimizations based on a particular implementation and on the hardware platform.

The TyCO compilation mechanism, from the source code into the assembly (intermediate language) preserves the nested structure of processes and scope boundaries, in the sense that a nested process sequence in the source code corresponds to a nested sequence of assembly instructions.

Messages and objects are expanded as they appear to the compiler. Process definitions are labeled and then referred to throughout the program by these code labels. The access to methods in an object is done by using the method table.

2.6.1 The Instruction Set

The instruction set is very reduced and was intended for minimal size and simple layout. The set is composed of instructions for heap allocation, data movement, control flow, reduction, scheduling and primitive operations.

Heap Allocation Instructions. These instructions create a frame in the heap of a size specified by an argument and, initialize it according to the type of instruction being executed. This initialization is expressed by the *descriptor* of the frame which has information about the type of the frame itself, as well as on the size of the frame.

- msgf** n, l allocates a frame for a message with label l and n arguments.
- objf** n allocates a frame for an object with n free variables.
- instf** n, l allocates a frame for keeping n free variables of a process definition stored in a code label l .
- new** r_k creates a new channel in the heap and keeps a reference to it in the register $R[k]$.

Data Movement Instructions. These instructions move data, one word at a time, within the heap and between the heap, the operand stack and the registers.

- put** w copies the word w directly to the the frame currently being assembled, in position pointed by register CF.
- push** w moves w to the top of the operand stack.
- pop** moves the word at the top of the operand stack to the frame currently being assembled, in position pointed by register CF.

Control Flow Instructions. These instruction allow the execution to diverge from the sequential order.

- iff** l jumps to the code label l if the value at the top of the operand stack is (boolean) false, otherwise the execution continues with the next instruction.
- switch** l_1, \dots, l_n jumps to the code at label l_k where k is the value at the top of the operand stack, and n is the number of code label arguments of the instruction.

jump l jumps unconditionally to the code label l .

Reduction Instructions. These instructions implement communication and generate a new task that is placed in the run-queue.

trobj c_i reduces the object pointed to by register CF with the message pointed to by register OF in channel c ($c \in \{r, a, f\}$).

trmsg c_i is the dual. It reduces the message pointed to by CF with the object at OF in channel c ($c \in \{r, a, f\}$).

instof n, l creates a new task with a code label l and n arguments allocated in the run-queue. Register OF points to the heap frame with the free variables and CF is made to point to the first available word for the arguments which are copied to the run-queue by subsequent **put** or **pop** instructions.

Scheduling Instructions. The instruction **ret** is used at the end of each task to schedule a new task for execution. It first checks whether the run-queue is empty, in which case the virtual machine halts. If the run-queue is not empty it dequeues a task by updating the registers: PC for the new byte-code to be executed and, FV and PM to hold the environment bindings for the free variables and parameters, respectively.

Instructions on Builtin Data-types. These instructions operate on the builtin data-types supported by the virtual machine. The operands are taken from the top of the operand stack and the result, when defined, is placed at the top of the operand stack. There is a set of instructions that operate on *boolean values*, a set to operate on *integers* and *floats* performing arithmetic operations, a third set, to implement *relational operators*, and another to manipulate *strings*.

2.6.2 Basic Compilation Blocks

In this section we illustrate how the basic TyCO processes are unfolded and compiled into the assembly language. We give the correspondence between the TyCO code and the respective assembly for the basic constructs in TyCO: messages, objects, definitions and instances.

Messages:

```

channel ! methodname [v1,...,vn]      msgf n,methodname
                                       put v1
                                       put ...
                                       put vn
                                       ...
                                       trmsg ci

```

After creating the message frame, the space allocated for its arguments is filled with some **put**'s, one for each argument. The construction ends with a *try to reduce the message* instruction (**trmsg**), at a given channel. For the object case things are not very different.

Objects:

```

channel ? {                               objf n
      l1(x1) = P1,                         put ...
      ...                                  put ...
      ln(xn) = Pn                          ...
}                                           trobj ci = {
                                           { l1, ..., ln }
                                           l1 = {
                                           ... % code for P1
                                           }
                                           ...
                                           ln = {
                                           ... % code for Pn
                                           }
                                           }
                                           ...

```

The translation starts by creating the frame for the object and then by filling it with the free variables that occur in the definition of the object. Then, the **trobj** instruction tries to reduce the object at the referred channel. At that point, the compiler inserts the method table and the definitions for each object, in order of appearance. Each method name is referred by the label. The Definition case is similar to the message case:

Definitions:

```

def X(v1,...,vn) = P                       def X(n) = {
                                           ... % code for P
                                           }

```

Here, we see how close the assembly is to the source TyCO code. The definition is compiled directly into the assembly.

Instantiations:

```

X[v]          instof n,x
              put v1
              ⋮
              put vn

```

Instruction **instof** has, as arguments, the number of parameters used in the instantiation and the label for the respective definition. This instruction appears in the assembly code, in the same relative position as it does in the TyCO code.

2.6.3 Example Compilations

In this section we present some examples of well-known programs, coded in TyCO and compiled to the intermediate assembly language to illustrate the compilation into the TyCO's virtual machine. We present non-optimized code to improve the readability. The examples are the same as presented in section 2.4. In the assembly code, arguments to instructions are divided into three distinct classes: i) variables local to the current task (r_i); ii) free variables that occur in the body of the current task (f_i), and; iii) arguments of a method invocation or arguments for a definition instantiation (a_i).

Example 1: the Cell data-type Here we take the example presented earlier in this chapter of a Cell data-type. We initialize the cell with the value 7 and then print it out.

```

def Cell( self, value ) = self ? {
  read( r ) = r![value] | Cell[self, value],
  write( v ) = Cell[self, v]
}
in new x Cell[x,7] | let w = x ! read[] in io ! put[w]

```

The code produced by the compiler is then the following:

```

main = {
  def Cell1(2) = {
    objf      2                % create object with 2 free vars
    put       a0
    put       a1
    trobj     a0 = {          % place the object at self
      { read, write }
      read = {
        msgf   1,0           % message with cell value
        put    f1
        trmsg  a0
        instof 2,Cell1       % re-instantiate the cell
        put    f0
        put    f1
      }
      write = {
        instof 2,Cell1       % re-instantiate the cell with new value
        put    f0
        put    a0
      }
    }
  }
  newc      r0                % create channel x
  instof    2,Cell1
  put       r0
  put       7
  newc      r1                % create channel w
  objf      0
  trobj     r1 = {          % the let reduction
    { ioput }
    ioput = {
      push    a0
      iout
    }
  }
  msgf      1,0
  put       r1
  trmsg     r0                % read invocation
}

```

Example 2: the Sieve of Erathostenes In this example we show the assembly produced when compiling the code presented in section 2.4.

```

main = {
  def Ints1(3) = {
    msgf      1,0           % sieve ! [n]
    put       a0
    trmsg     a2
    push      a0
    push      a1
    lti       % if n < m then ...
    iff       10
    instof    3,Ints1       % Ints[n + 1, m, sieve]
    push      a0
    push      1
    addi
    pop
    put       a1
  }
}

```

```

10:   put      a2
    }
    def Sieve2(3) = {
      objf      3          % self ? (n) = ...
      put      a2
      put      a1
      put      a0
      trobj     a0 = {
        { m0_0 }
        m0_0 = {
          push   a0
          push   f1
          modi   % compute (n % myGrain)
          push   0
          negi
          iff    l1
          msgf   1,0      % nextSieve ! [n]
          put    a0
          trmsg  f0
11:   instof   3,Sieve2
          put   f2
          put   f1
          put   f0
        }
      }
    }
    def LastSieve3(2) = {
      objf      2          % self ? (n) = ...
      put      a1
      put      a0
      trobj     a0 = {
        { m1_0 }
        m1_0 = {
          push   a0
          push   f0
          modi   % compute (n % myGrain)
          push   0
          negi
          iff    l2
          push   a0
          iout
          newc   r0          % new newSieve
          instof 3,Sieve2    % Sieve[self, myGrain, newSieve]
          put    f1
          put    f0
          put    r0
          instof 2,LastSieve3 % LastSieve[newSieve, n]
          put    r0
          put    a0
          jump   l3
12:   instof   2,LastSieve3 % LastSieve[self, myGrain]
          put   f1
          put   f0
13:   }
        }
      }
    }
    push      2
    iout
    newc      r0          % new sieve

```

```
    instof    3,Ints1                % Ints[2, 10000, sieve]
    put      2
    put      10240
    put      r0
    instof    2,LastSieve3           % LastSieve[sieve, 2]
    put      r0
    put      2
}
```

2.7 Chapter Overview

In this chapter we gave an overview on the TyCO language and implementation. We focused on the formal specification of the language namely its semantics. This forms the basis of our work, the design and implementation of a distributed TyCO calculus and language - the subject of the next chapter.

Chapter 3

The DiTyCO Programming Language

In this chapter we introduce the Distributed Typed Concurrent Objects (DiTyCO) programming language, an extension of TyCO presented in the previous chapter. The motivation behind this work is to introduce distributed computations and mobile resources in the base TyCO model. The extensions of the calculus are based on the lexical scope of a program acting on a network. We introduce new constructors in the programming language to allow distribution and mobility, and illustrate their use by presenting some programming examples.

3.1 Motivation

To create a programming language capable of coping with the new trend in software communication, distribution and mobility, it is important to base implementations on provably correct specification such as those provided by process calculi. We started from the TyCO calculus which is a stabilized model in the area of concurrent calculus and language implementations. TyCO has solid theoretical results in the field of process concurrency and it constitutes an ideal starting point to evolve and add notions of distribution and mobility.

Even though, the TyCO calculus is prepared to deal with concurrency, the corresponding TyCO language does not have built in its kernel any primitive to deal

with concurrency in an *inter-process* way. That is, there is no provision for remote communication and no notion of networks location. Communication in TyCO calculus is only local.

In TyCO there are no instructions to send or receive messages from another TyCO virtual machine (Tvm). Every TyCO program must run in a single Tvm which has no capability to communicate with another Tvm. Therefore, concurrency at the level of program communication is not possible.

The construction of DiTyCO aimed at extending TyCO with the necessary features to make it able to deal with distribution and mobility, while retaining its concurrent properties, the object-based programming style and trying to minimize, the changes in the original TyCO language.

3.2 Introducing DiTyCO

We extend TyCO with the notion of *site* – an abstraction for a place where a computation evolves. We assume a *lexical scope* discipline for programs in such a way that names are always bound to the site they are created in. In this extended model, conventional (name-passing) computations only happen at sites (to avoid global synchronization). Code movement is triggered by the lexical scope of variables prefixing the corresponding processes: method invocations, objects or definitions. The migration of prefixed processes is deterministic, point to point, and asynchronous. Synchronization only happens locally, at reduction time.

The model has two logical levels: processes and networks (cf. [8, 44, 73]). Local computations happen at sites, as prescribed by the semantics of the base calculus. Remote computations occur between prefixed processes at different sites in a network and involve message passing between processes before reduction may take place. Therefore, the new model builds the second level, on top of TyCO.

The DiTyCO [29, 54, 89] model presents a *flat organization of sites*, that reflects the architectures of current implementations of high-performance networks, namely Giga-Switches [27, 28]. 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.

Due to the builtin notions of concurrency, to its characteristics of distribution and code mobility, DiTyCO is prepared to be used as a language in an environment full of events and for interactions between processes. This kind of environment resembles the situation of nowadays processes, lying in computers connected to large networks, communicating with others and prepared to exchange information along the network.

Our approach in the development of a concurrent programming language and run-time system for distributed systems is distinct from other well known works as CORBA [3], DCOM [2] or Java/RMI [1], although we share some of the goals. The main differences can be summarized as follows:

- we are interested in developing systems that can be provably correct, with relatively simple semantics;
- 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;
- we provide inter-platform support in heterogeneous networks by using emulated byte-code for implementation technology (which in this cases resembles Java/RMI).

3.3 The Calculus and its Semantics

The full DiTyCO calculus grows from the centralized version by adding a new layer of abstraction representing a network of *locations*, where processes are running. We will hereafter refer to such locations as *sites*.

3.3.1 Syntax

We let s range over the set of sites. *Located channels*, are channel-site pairs; we write $x@s$ for channel x located at site s . Similarly, *located definitions*, are definition-site pairs, written $X@s$, denoting the definition X located at site s . Sites, channels and definitions are collectively called *names*. *Located names* stand for the above mentioned compound names. We allow located names to occur in any position in the base calculus

where (non-binding occurrences of) identifiers can. 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 calculus thus obtained constitutes the *first layer* of the model.

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 [89].

N	$::=$	$\mathbf{0}$	Terminated network
		$N \parallel N$	Concurrent composition
		$\mathbf{new} \ x@s \ N$	New located channel
		$\mathbf{def} \ D@s \ \mathbf{in} \ N$	New located definition
		$s[P]$	Site with running process

Notice that processes P (see section 2.2.1) now evolve within sites. Therefore, this new grammar constitutes an extension to the previous one.

The bindings in networks are as expected: a located name $x@s$ occurs *free* in a network if $x@s$ is not in the scope of a $\mathbf{new} \ x@s$; otherwise $x@s$ occurs *bound*. The set of free located names in a network, notation $\text{fn}(N)$, is defined accordingly. Similarly, a located definition $X@s$ occurs free in a network N if $X@s$ is not in the scope of a $\mathbf{def} \ D@s$, for X one of the X_i defined in $X_i = (\tilde{x}_i)P_i$ in D . The set $\text{ft}(N)$ of free located definitions in networks is defined also accordingly.

3.3.2 Structural Congruence

Structural congruence is important to allow for a generalization of rules that must be applied in the calculus, allowing us to abstract from the static structure of networks in order to obtain redexes. The structural congruence rules for DiTyCO are:

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

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 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 channel x or a definition 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 by the *lexical scoping convention*. A channel x is uploaded to a site s ; a definition X is uploaded to a site s . A *translation of identifiers* from site r to site s is a total function σ_{rs} defined as follows:

$$\begin{aligned}
\sigma_{rs}(x) &\stackrel{\text{def}}{=} x@r & \sigma_{rs}(x@s) &\stackrel{\text{def}}{=} x & \sigma_{rs}(v) &\stackrel{\text{def}}{=} v \\
\sigma_{rs}(X) &\stackrel{\text{def}}{=} X@r & \sigma_{rs}(X@s) &\stackrel{\text{def}}{=} X
\end{aligned}$$

Where the last rule is applied last.

Basically, we may have *located names* and *network names*. Whenever a located name is in its origin site, it drops the specification of the site, as it is implicit. On the contrary, names located in a different site from their origin, must preserve the full specification of their location (channel plus site). Finally, when names are out of any site, they must retain the full format as network names.

This additional layer does not however introduce new reduction operations in the calculus. In fact, reduction can only be performed locally at sites and it remains either communication or instantiation as described in TyCO.

3.3.3 Reduction and Mobility

In order to deal with the notion of located processes, and to relate the evolution of computations in sites where they occur, we must create new rules for the calculus, as well as adapt other ones.

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

$$\frac{[\text{LOCAL}] \quad P \rightarrow Q}{s[P] \rightarrow s[Q]}$$

This rule, plus the rules for parallel composition, restriction, definitions and structural congruence expressed above, forms the backbone of the reduction relation. We now discuss the four remaining axioms that introduce a kind of mobility, in which parts of the computation are moved into another location to be executed in that new place.

Processes prefixed with located names play a crucial role in the model. They determine the semantics associated with code movement. A process $x@s!l[\tilde{v}]$ represents a remote method invocation on an object located at a channel x at site s . The respective semantics moves the message to site s . A process $x@s?M$ 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; from an implementation point of view the difference is not that different either. Thus, we see that lexical scope on names induces a kind of *code shipping* semantics for method invocations and objects. The axioms for the message and object case are as follows:

[SHIPM]

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

[SHIPO]

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

Rule SHIPM prescribe that if the method invocation $x@s!l[\tilde{v}]$ (respectively, rule SHIPO for object $x@s?M$) 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 $x@s!l[\tilde{v}]$ (respectively $x@s?M$) from r to s we actually transmit $l[\tilde{v}\sigma_{rs}]$ (respectively $M\sigma_{rs}$). This is the essence of the axioms SHIP.

Let us illustrate the use of the previous axioms by presenting an example of a remote procedure call in DiTyCO. 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).

$$\begin{aligned}
& s[\mathbf{new} \ a \ (p@r![va] \mid a?(y) = P)] \parallel r[p?(xr) = Q] \equiv (\text{applying: NEW, EXN}) \\
& \mathbf{new} \ a@s \ (s[p@r![va]] \parallel s[a?(y) = P] \parallel r[p?(xr) = Q]) \rightarrow (\text{applying: SHIPM}) \\
& \mathbf{new} \ a@s \ (r[p![v@s \ a@s]] \parallel s[a?(y) = P] \parallel r[p?(xr) = Q]) \equiv (\text{applying: SPLIT}) \\
& \mathbf{new} \ a@s \ s[a?(y) = P] \parallel r[p![v@s \ a@s] \mid p?(xr) = Q] \rightarrow (\text{applying: LOCAL}) \\
& \mathbf{new} \ a@s \ s[a?(y) = P] \parallel r[Q\{v@s \ a@s/xr\}] \rightarrow^* \ (\text{Q reduces}) \\
& \mathbf{new} \ a@s \ s[a?(y) = P] \parallel r[a@s![u]] \rightarrow \equiv \rightarrow (\text{applying: SHIPM, SPLIT, LOCAL}) \\
& \mathbf{new} \ a@s \ s[P\{u@r/y\}] \equiv (\text{applying: NEW}) \\
& s[\mathbf{new} \ a \ P\{u@r/y\}]
\end{aligned}$$

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. [33]); the former is an asynchronous operation, the latter requires a *rendez-vous*, which reflects the actual implementation.

Another kind of remote interaction occurs when a site s finds an instantiation of the form $X@s[\tilde{v}]$. A definition X prefixed with a site name r indicates that X was originally defined at site r .

[SHIPD]

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

In this case the arguments for the instantiation will move to the original site of the definition, and will be instantiated and executed there. Therefore, we also have code *shipping* which is different from other approaches as in Java, where the code is *fetched* (downloaded from the origin).

3.4 The Programming Model

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

export $x P$

import $x \mathbf{from} s \mathbf{in} P$

export $X(\tilde{x}) = Q \mathbf{in} P$

import $X \mathbf{from} s \mathbf{in} P$

These appear always at the topmost level of a program. 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:

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

To include these new constructs we need to augment the DiTyCO grammar, which can be done by extending TyCO processes with top level declarations. Let S range over by the set of sites, and w range over by the set of names (channels and definitions):

$$\begin{array}{lll}
S & ::= & \mathbf{export} \ w \ \mathbf{in} \ S & \text{Export interface} \\
& | & \mathbf{import} \ w \ \mathbf{from} \ s \ \mathbf{in} \ S & \text{Import interface} \\
& | & P & \text{TyCO process}
\end{array}$$

3.5 Programming Examples

In this section we present some examples to illustrate programming in DiTyCO, particularly in what concerns real concurrency modeled by several processes interchanging information, distribution and code mobility. These examples are useful to assess the language expressiveness.

3.5.1 The Bank Account

The first example is an adaptation of the example presented in section 2.4 concerning a bank account. Initially the example was presented as a piece of code modeling a bank account through a persistent object. The client was modeled by invoking methods (`withdraw`, `deposit` and `balance`) on the account object.

```

CentralBank[
    export bankChannel
    def Account(account, amount) = {
        account ? {
            deposit(x) = Account[account, amount + x],
            withdraw(x) = Account[account, amount - x],
            balance(r) = r![amount] | Account[account, amount]
        }
    } in
    def Bank(self) = {
        self ? {
            newClient(amount, replyTo) = new x Account[x, amount] |
                replyTo![x] |
                Bank[self]
        }
    } in
    Bank[bankChannel]
]

```

Note that we have a bank capable of creating as many client accounts as the machine memory concedes. The client only needs to contact the Bank via its public channel `bankChannel`, invoking the only method `newClient`, with an initial amount. The account is then created and the client receives in return the access point to that account. A program for the client side may be for instance:

```

Client[
    import bankChannel from CentralBank in
    new myAccount
    bankChannel ! newClient[myAccount, 1000] |
    ...
]

```

Here, the client is now able to use the account created and invoke the available methods to manage it. For example, imagine that the client makes a deposit of 50, withdraws 70, and checks the account's balance (not necessarily by this order). To print out the balance, we create an object placed in channel `y`, which will reduce with the message returned from `CentralBank`:

```

...
new y y ? { (v) = io ! put[v] } |
myAccount ! deposit[50] | myAccount ! withdraw[70] | myAccount ! balance[y]
...

```

Throughout this example it is clear the resemblance of a typical client server paradigm. Moreover, we are modeling a remote evaluation paradigm. The client uses the bank to make the computations. This is achieved by simple message passing.

3.5.2 The SETI@Server

The next example is 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.

Seti Server	Seti Client
<pre> export Install(self) = self ? { <install>; Go[self] } self![] in export Go(self) = self ? { let data = database!newChunk[] in <process>; Go[self] } self![] in new database def Database(self) = self ? { newData(data) = ... newChunk(replyTo) = ... } in Database[database] </pre>	<pre> import Install from setiServer in new handle Install[handle] </pre>

This program introduces the concept of uploading definitions. This feature 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 under the identifier `handle@setiClient`. 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 identifiers moving through the network:

```
def Install@setiServer(self)=... Go@setiServer(self)=... in
new handle@setiClient
setiServer[. . .] ||
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⟩σsetiServer setiClient ; Go[handle]]

```

After the local execution of $\langle \text{process} \rangle$ the program then loops endlessly, fetching and processing new data chunks.

3.5.3 Network Paradigms

In the last programming examples we presented two common paradigms of network programming: the *client-server* (CS) and the *code on demand* (COD). Those, may be sequentially organized in a growing scale of complexity and difficulty in programming [22]. The *client-server* is usually related with message passing, in the sense that data is passed to the server, which does the computation on it and returns the computed result back to the client. All the communication between the client and the server is performed by simple message transfer.

The *remote evaluation* (RE) paradigm is more complex because one site sends to another site the know-how, that is, the code to perform some service on its behalf. The result of this remote evaluation is to be returned back to the requesting site.

The *code on demand* sets a higher degree of complexity in network programming. In this paradigm, what is transferred between two different sites is a request of remote code to be executed locally, and the returned code. Usually, the code is part of a remote program, that upon the request, has to be packed and delivered to the requesting site.

These three paradigms are depicted in figure 3.1:

In chapter 5 we will refer to a new programming paradigm (*Mobile Agent*) which is based on a program moving from DT-node to DT-node (different IP locations). This situation, called *strong migration*, is possible by capturing the execution state, migrating the whole program data and, resuming in the other site the state and data structures. Strong mobility was added as an experiment in DiTyCO.

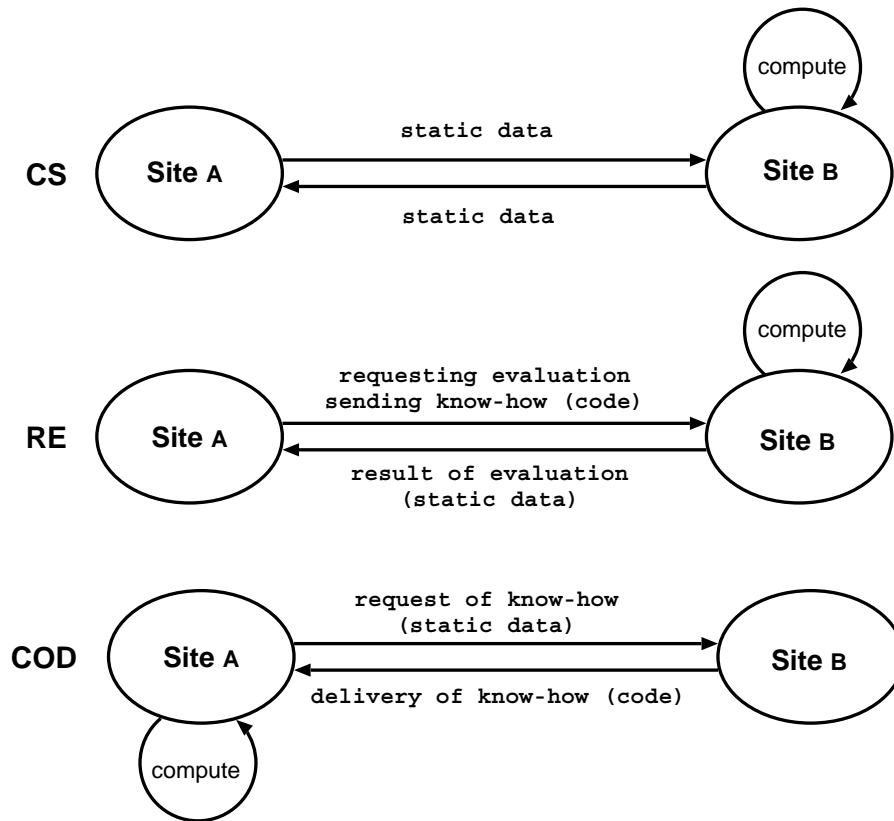


Figure 3.1: CS, RE and COD programming paradigms.

3.6 Outline of Changes in the Compiler

In DiTyCO, the compiling mechanism is basically the same as in TyCO, it still includes a two-phase compiling process including compilation to an assembly intermediate language and then, compilation into byte-code. However, to deal with the new features in DiTyCO, like distribution and mobility, the compiler had to be modified. The two new constructs `export` and `import` were added to the language and must also be included in the assembly language, as well as in the byte-code.

In the assembly language, some instructions had to be adapted to deal with remote channels and remote definitions, namely in the treatment of reduction instructions. Some other instructions were added to the language. The new instructions were basically introduced to help positioning of data in heap frames.

The reduction instructions (`trmsg` and `trobj`) have some extra functionality, now being able to reduce messages or objects in channels that are not local to current

site, that is, channels that were created in another site, and that were either imported explicitly through the `import` instruction or implicitly imported through scope extrusion.

A similar situation involves the instructions `instf` and `instof` that handle instantiations. The `instof` instruction was unfolded, and is now represented by two instructions (`instf` and `instof`) for export and import of remote definitions.

Further details on the implementation of these instructions are given in chapter 4.

3.7 Chapter Overview

In this chapter we presented the DiTyCO model and programming language. We showed how, with only two new constructs, paired with three new semantic rules we obtain a calculus that is able to model several well-known situations of the distributed programming area. We presented some examples written in DiTyCO, illustrating the *remote evaluation* and *code-on-demand* paradigms to show the new distribution and mobile code characteristics. We also outlined the necessary modifications that had to be made to the compiler in order to implement those characteristics. In the next chapter we present the underlining virtual machine that constitutes the run-time system for DiTyCO.

Chapter 4

The Virtual Machine Implementation

In this chapter we describe the architecture and the implementation of the run-time system for the DiTyCO programming language. We begin by giving an outline of the components that constitute the system, and then describe each in detail. Our approach is to begin with the initial TyCO virtual machine (Tvm) and then show what features we have included to make it able to communicate with other Tvm's. This new virtual machine constitutes the DiTyCO virtual machine (DTvm). We conclude the chapter by presenting an example of memory and dynamic structures in use during a typical virtual machine interaction.

4.1 General Description

From an external point of view, the DiTyCO system is characterized by *sites* that coexist in the network and that communicate between them by exchanging objects, messages (asynchronous method calls) and process definitions. To distinguish “language messages” from “TCP messages” hereafter, we will use the term *message* to refer to the language construct that invokes a method in some object, and the term *packet* for the TCP packet that is sent along the network.

Sites are the basic computation unit, and any program is run at some site. Therefore, a site needs to be created for a program to be run. In TyCO, the notion of site

does not exist, neither exists inter-program communication, in DiTyCO, however, site creation is a compulsory step whether or not the program is meant for inter-program communication.

For the use of the system, a site must be created in some specific IP-node, that is, the user must know in what machine (physical machine, eventually connected to a network with other machines) to create the site. After that, communication between the newly created site and other sites takes place independently of their physical location.

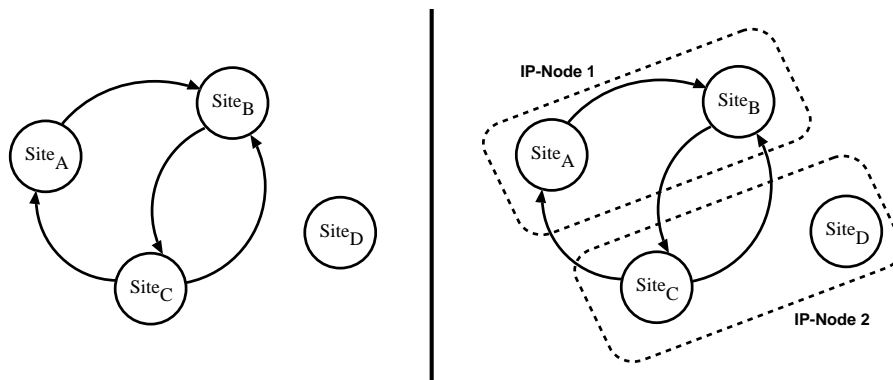


Figure 4.1: Communication between sites in a network

From the internal point of view, our implementation of DiTyCO maps each physical machine connected to the network with a *node*, which constitutes a pool of sites. Therefore, each site lies in its machine node (see figure 4.1). This introduces an extra layer between the site layer and the network layer. Thus, DiTyCO has three layers: **sites**, **nodes** and **networks**. Nodes have an one-to-one correspondence with physical IP machines, 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 machine, opening the possibility of real parallel execution. Finally, the network layer is composed of multiple DiTyCO nodes connected through the underlying IP topology.

Communication between sites is possible through the site's interface which is specified by the programmer by means of *importing* and *exporting* channels and process definitions. A channel or definition that was exported may be accessed in any other site through an import operation. The site's interface is established, in the beginning of execution of the program, by communication between the site and the network. This communication involves packet passing through a *proxy* that exists in every DiTyCO

node. Figure 4.2 depicts this situation: sites are grouped in nodes, and communicate with the network through a node proxy. Arrows pointing from the site to the proxy, and from the proxy to the network are *export* operations; the other arrows are the dual – *import* operations.

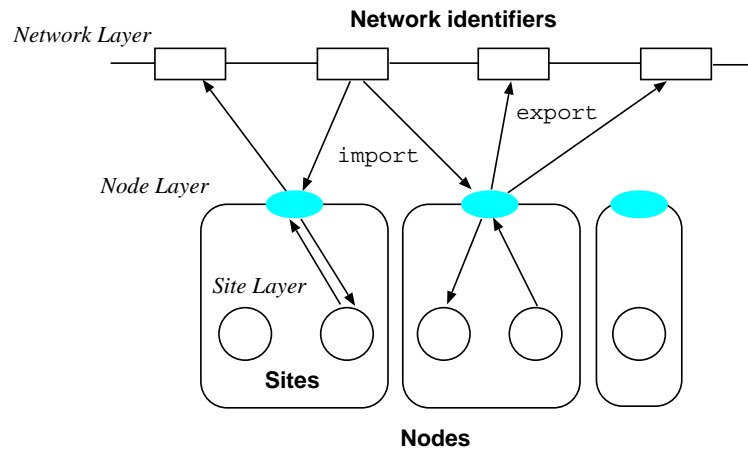


Figure 4.2: Sites building their interface.

Name resolution is done by a network wide service named *Network Name Server* (NNS) much in the same way that *Domain Name Server* (DNS) does. The NNS provides a resolution of names that occur in DiTyCO programs giving the full address for those names, that is, the full address for the IP correspondent machine, plus the respective memory reference.

Communication between different sites needs to go through the three-layers (site, node and network) and then, in the reverse direction until it reaches the target site. The NNS is only useful in the beginning of the execution of programs, when the site interface is being built. After that, communication between sites only involves the NNS when an error condition is reported by the proxy of the target site.

In the following sections we will describe in detail each of the three layers (*site*, *node*, *network*) outlined above. We take a bottom-up approach and begin with the site layer.

4.2 The Site Layer

The site layer is composed by the DTvms which are enhanced Tvms, capable of emulating DiTyCO byte-code and communicating with other DTvms. Each DTvm

is still running on a single thread. The instruction set is augmented, as well as the assembly and the byte-code. We give a description of the memory architecture and focus on the changes operated on the Tvm and its original instruction set.

4.2.1 Memory Architecture

The DiTyCO virtual machine maintains most of the initial data structures present in the original TyCO virtual machine. The new elements introduced are an *exports table*, an handle for the *queue* of messages delivered to the site, a *remote program area* and an *I/O port*. The emulator is also extended to operate on the new items. This architecture is represented in figure 4.3.

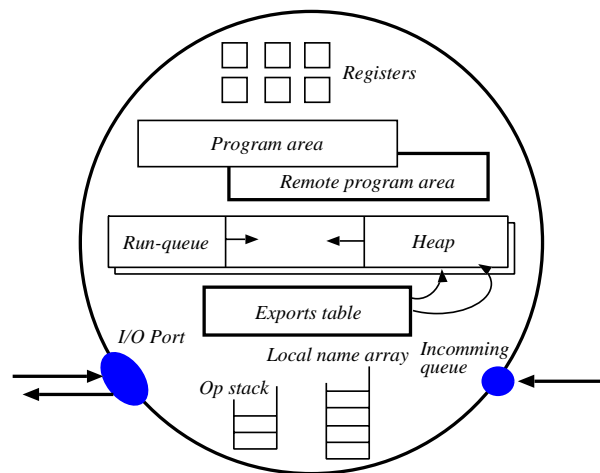


Figure 4.3: The DiTyCO Virtual Machine.

The other data-structures of the machine, namely: the heap, run-queue, local name array, stack, program area and set of registers are the same, and operate much in the same manner as in the Tvm.

4.2.2 The Local Exports Table

The *exports table* is a data structure that, for each site, provides a way to maintain references to local names (channels and definitions) that have been exported. This is important namely in the presence of a garbage collection, which may change the memory reference of those names. It also provides a way of getting the network address

of a non-local names.

References to local names in the heap are set through a pointer to the respective heap frame for that name. For external names, however, we have to store more information: i) the IP address of the machine; ii) an integer specifying the site inside the node, and; iii) the memory position for that name, in the heap of the remote site. All that information can not fit in a normal heap word (32 bits in current implementation). Therefore, for imported names, we store the information through indirect access using the exports table. An imported name is referred in the heap as a tagged heap-word holding the index for the exports table, where the information for that remote name is stored.

Exported names are also placed in the exports table. We can differentiate between imported and exported names by the *IP address* and *Location* of the name, which for local names has a zero on those fields. Exported and imported names come in two flavors: they may be explicit or implicit. When a name is exported (respectively imported) through the `export` (respectively `import`) instruction, it is *explicitly exported* (respectively *explicitly imported*). When a channel is passed as an argument in some message or instantiation, it is *implicitly exported* (respectively *implicitly imported*). In both cases the NNS is not involved. These situations have to be both stored in the exports table for the purpose of keeping the right references when a garbage collection occurs. However, from the point of view of the data stored in the table there is no distinction between implicit and explicit situations. These cases are illustrated in the next table:

<i>index</i>	IP	Loc	Offset	<i>comment</i>
<i>0</i>	196.64.2.35	2	238	<i>imported name</i>
<i>1</i>	0	0	36	<i>exported name</i>
<i>2</i>	0	0	64	<i>implicitly exported name</i>
<i>3</i>	196.64.2.35	2	252	<i>implicitly imported name</i>

Table 4.1: The Exports Table.

There is no distinction between a reference in the exports table to a channel or to a definition. Both have the same type of entry in the table. The only difference is that they are pointing to a different frame in the heap space.

4.2.3 Representing Names, Strings and Values

In DiTyCO the arguments for messages and definitions occupy one word per argument. The word may represent: the value itself, that is, a boolean, an integer or a floating point; a pointer to the string frame for a string; or a name, case in which we have to distinguish between local names and remote names. For local names, either channels or definitions, we place a pointer to the respective heap frame. For remote names (channels or definitions) we use the index for that name in the local exports table. Thus, we have to distinguish between the four cases: local names, remote names, strings and values.

Both local names and strings are referred to, using a pointer and can be distinguished by the descriptor of the frame pointed to. Hence, those two cases may be reduced to a single one. Therefore we only have to distinguish between values, pointers and indexes in the exports table. To accomplish this we use a 2-bit mask as shown in figure 4.4.

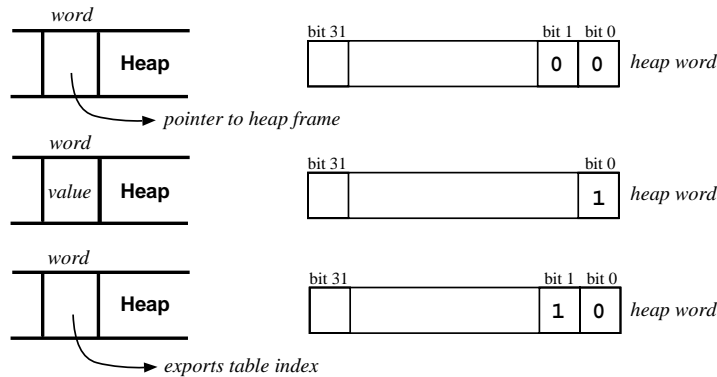


Figure 4.4: DiTyCO arguments

Values are represented with bit 0 set. Pointers and indexes in the exports table have both bit 0 set to 0, and bit 1 set to 1 in the first case, and to 0 in the index case. Recovery of values and indexes from this codification is done by masking the proper number of bits. Pointers can be used directly, taking advantage of the common memory organization, which does not use the last two bits in a 32 bit architecture (addresses align on 4 byte words).

4.2.4 The Incoming Queue

The *incoming queue* is a data structure that holds messages targeted to the site. Those messages may consist of inter-site communication as well as import/export operations. The messages hold information that came from the network and that are being delivered by the node's proxy to the current site. As the proxy is the "producer" for the queue and, the site the "consumer", synchronization is needed, and we use a lock to access the queue. Whenever the DTvm reaches the end of a task, it checks the incoming queue to see if there are any pending messages. In that case they are read and placed in the heap, and tasks (eventually created) are placed in the run-queue for later execution. This happens before checking if there are any other tasks in the run-queue. This way, we prevent unfair situations of execution of tasks, and prevent the site from finishing when there are messages waiting to be processed. However, we do not prevent a site from finishing prior to the en-queueing of a message targeted to the site, and already received by the node's proxy. A site does not contact the proxy to check for new messages before terminating execution because the benefit would be minimal as there is the chance of having messages targeted to the site, that have not yet reached the proxy.

4.2.5 Remote Program Area

The *remote program area* is a place where incoming byte-code is placed. This remote byte-code may be the code for the methods of an external object, or the code for an imported definition. Its structure is slightly different from the *local program area* because it needs management. We use an indexed address space to keep track of byte-code in use. Every block of incoming byte-code is referred through a pointer that is stored in the next available position of the array. Thus, references to the byte-code are done through the index of the array. This situation is described in figure 4.5.

4.2.6 The I/O Port

The user may interact with the site, providing input for the program and receiving output from it using this port. Every site maintains the *I/O port* which is a socket based connection between the thread running the site and the user's shell. This

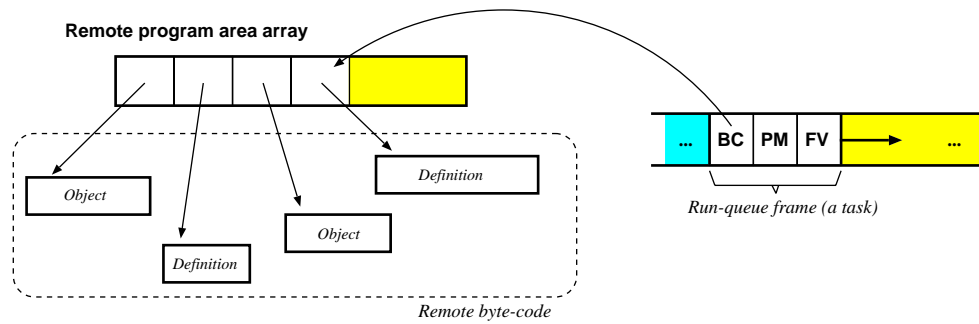


Figure 4.5: The Remote Program Area.

provides an extra degree of flexibility for communicating with sites that are running in other machines.

4.2.7 New instructions

In order to handle the new features of the language, new emulator instructions had to be introduced. The new instructions are:

Import and Export Instructions. As seen in the previous chapter, the language was augmented with two instructions, the `import` and the `export` instructions. Naturally, these two also have their counterparts in the machine instruction set. A new instruction to reserve heap space for imported names was also added.

`nimports n` reserves space in the beginning of the heap to create references to the exports table where information regarding `n` imported names is placed.

```
nimports(n) = { FV = HP; HP -= n; }
```

`export n, varname` makes the name `varname` publicly available by storing it into the NNS. The parameter `n` is the index in the local name array (register) `R[n]` which keeps the binding for the exported local name. This instruction is used both for exported channels as well as for exported definitions.

```
export(k, name) {
  p = create_channel();
  point_to_channel(k, p);
  export_to_NNS(p, name);
}
```

import sitename, fv_i , varname is the dual operation, which queries the NNS for the network address of the name `varname` located at site `sitename`. The return information is placed in the local exports table, whose position is stored in free variable index fv_i .

```
import(site,i,name) {
  import_from_NNS(site,name,&name_addr);
  idx = insert_in_exportsT(&name_addr);
  p = point_to_exports(idx);
  set_free_var(p,i)
}
```

Definitions. Definitions in DiTyCO are implemented in a slightly different way. In TyCO, there was an optimization which placed directly the definition's arguments in the run-queue (optimizing time and space). However, in DiTyCO this procedure can not be made without knowing first if the instantiation is local or remote, as it may be the case that the arguments have to be placed on a remote run-queue. Hence, our approach is to place the arguments first in a heap frame, and then check if the instantiation is to occur locally or in a remote site. If the instantiation is local, then it proceeds as usual by creating a task in the local run-queue; if it is a remote instantiation, then this frame with the arguments is sent to the site where the definition was created.

In DiTyCO definitions may therefore be local or can also be publicly available. When local, the `def` instruction is used jointly with the `newd` instruction. When exported (publicly available), the `def` instruction is used jointly with the `expdef` instruction. The only difference between the instructions `newd` and `expdef` is that the last one contacts the NNS to register the name of the definition.

newd defname, n , r_i creates a heap frame with the correspondent descriptor, a pointer to the byte-code position where the definition `defname` is stored, space for n number of free variables that occur in the definition, and makes the local register index i point to this created frame. Subsequent `put`'s will place the free variables in this frame.

```
newd(name,n,i) {
  p = create_def(name,n);
  point_to_frame(p,i);
}
```

expdef defname, n, r_i does exactly the same as the above instructions, plus the contact with the NNS for storing the full network address for the frame created in the heap.

```
expdef(name, n, i) {
  p = create_def(name, n);
  point_to_frame(p, i);
  export_to_NNS(p, name);
}
```

Heap Frame Filling Instructions. To allow the creation of multiple recursive definitions, we created an instruction that specifies which heap frame (definition) to fill with. Basically, we may have DiTyCO code which resembles the following example:

```
def X()= ... Y[] ...
and Y()= ... Z[] ...
and Z()= ... X[] ...
in ...
```

In order to handle these multiple definitions one needs to first compute the free variables for each of the definitions. We use a **focus** instruction to select the frame where the free variables will be placed.

focus r_i selects a heap frame pointed by R[i] to fill with the respective definition's free variables.

```
focus(i) = { CF = R[i] }
```

Hence, the correspondent generated code for the above example of multiple definitions is:

```
newd X, ..., r0
newd Y, ..., r1
newd Z, ..., r2
focus r0
put ...
...
focus r1
put ...
...
focus r2
put ...
...
```

Instantiation. The instantiation in DiTyCO is implemented using the instructions *instance frame* (`instf`) and *instance of* (`instof`). The first one places the arguments for instantiation in a heap frame, and the second tries to instantiate with those arguments.

`instf n` creates a heap frame with a descriptor and the space for `n` arguments to be placed in order by subsequent `put` instructions.

`instof reg` tries to instantiate the arguments using the definition pointed to by register `reg`. Note that this register may be pointing to a local name (r_i), that is, a local definition, or an imported definition, in which case the register (f_i) must be representing an index in the exports table. In the case of a local definition (`reg` is of the form s_i or f_i , but local) a new task is created in the run-queue with three words: a pointer to the byte-code definition, a pointer to the frame definition (created during `newd` or `expdef`), and a pointer to the arguments (placed in the frame created with `instf`). In the case of a remote definition, the heap frame holding the arguments must be packed and sent to the site where the definition was created for local instantiation.

```
instof(reg) {
  if local(reg)
    instof_local(reg);
  else
    send_frame(reg, pack_inst_frame());
}
```

Other Reduction Instructions. The other instructions that reduce objects and messages also had to be modified to prevent situations in which the reduction is to take place at another site. Therefore, the instructions `trmsg reg` and `trobj reg` had an increase in functionality for cases in which the `reg` argument is a reference to a remote channel. For those cases the message (respectively, the object) is packed, the bindings are translated and it is sent to the site referred by the remote channel for local instantiation.

```
trmsg(reg) {
  if local(reg)
    trmsg_local(reg);
  else
    send_frame(
      reg, pack_msg_frame()
    );
}

trobj(reg) {
  if local(reg)
    trobj_local(reg);
  else
    send_frame(
      reg, pack_obj_frame()
    );
}
```

4.3 The Node Layer

The next layer in our model is the node layer. This layer does not exist in the formal model. It was created, basically, to take advantage of present day computer architectures and, particularly, of machines having several processors. The node layer groups all the sites that are located in the same physical machine. This makes the communication between sites lying in the same node more efficient by taking advantage of shared memory.

4.3.1 Memory Architecture

A DiTyCO node (*DT-node*) is a Unix process that launches several posix-threads in order to establish the DiTyCO system. The node itself launches two threads: the DiTyCO Interface (*DTi*) provides an interface between the user and the DT-node, and the communications daemon (the *proxy*) that consists of several threads to perform inter-site communication. Every site inside a node is launched by the DTi of that node, upon receiving the proper command from the user.

The only restriction to the number of sites in a node is the memory available in the system. Communication between the DiTyCO system threads is done through shared memory.

The user interacts with the node by using a shell – the DTsh – which may communicate with any DT-node in the network. Execution of DiTyCO programs involves four steps: i) registration of a site in the network; ii) creation of the site in a DT-node; iii) loading of the program to that site, and finally; iv) execution of the program in the site. Sites are created by the DTi and communicate with each other via the node’s proxy, no matter whether communication is local to the node or not.

Figure 4.6 illustrates a DT-node with three sites launched by the node’s DTi, as a result of commands issued by three DTsh’s, two lying in the same IP machine and one from an external machine.

In the current implementation, a DT-node uses six threads plus a thread for each site launched. This kind of organization is depicted in figure 4.7 where we show how threads are launched during the start of a DT-node, and the launch of two sites in it.

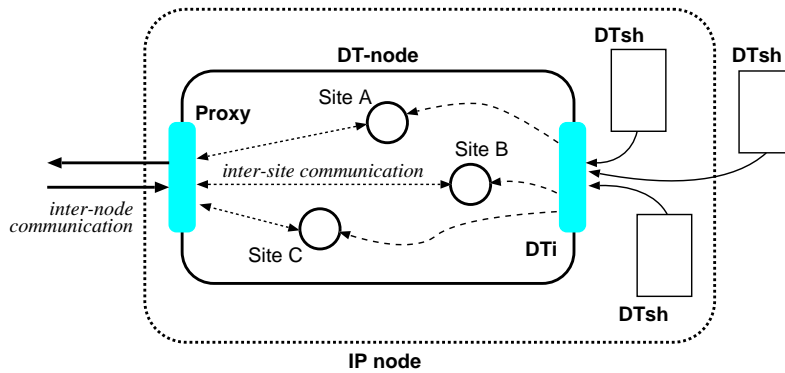


Figure 4.6: A DiTyCO-node in action.

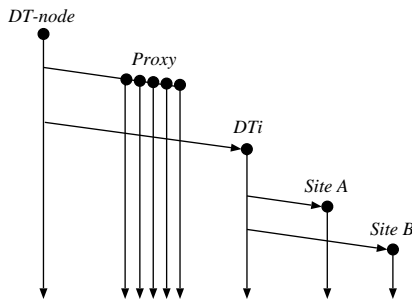


Figure 4.7: The DiTyCO-node Threads

After launching the other threads, thread DT-node is suspended waiting for the proxy and DTi to terminate. Hence, after the creation of the proxy and DTi, the first thread enters a suspension state.

4.3.2 The DiTyCO shell

There are two modes of invoking DiTyCO: from the command line, in order to execute programs which do not receive input from the user, and; a shell which can be used as front-end to the system. This shell is the interface for users to create sites and run DiTyCO programs. It provides the means for the user to launch sites in a node, to provide input to the site and for receiving output from it. The command line is much faster and may be used for batch jobs. However, in order to be correctly executed, the user must provide a fresh site name, otherwise execution will stop with an error message.

The command line format is described below:

```
run -m <machinename> -s <sitename> -f <filename> [h=x,o=y,r=z]
'h' stands for heapsize
'o' stands for operand stack size
'r' stands for local name array size
x,y,z should be specified in bytes
```

The basic menu shell commands are:

1. `site <sitename> <machinename>`
To register the name `sitename` in a node lying at IP node named `machinename`.
2. `login <machinename>`
To login the user in that machine's node.
3. `load <filename>`
To load the byte-code file `filename` on the newly created site.
4. `run [<heapsize> <Op stack size> <local name>]`
To run the program, eventually specifying the size for the heap, the operand stack, and the local name array.
5. `quit`
To quit the DTsh.

The `site` command registers a *sitename* in the network. The user has to specify a site name and the machine name where that site is supposed to be created. The *machine name* is the one that is resolved by the DNS. The site name, on the other hand, is resolved by NNS and has no warranty to be unique at that moment. Thus, it is possible that the `site` command returns an error stating that the name is already in use.

The `login` command creates a communication link between the DTsh and the DT-node of the particular machine specified by the command argument.

The `load` command allocates the space for a new DTvm for running a site; it checks the file size; allocates memory space in the newly created site, and loads the program file in the program area of the newly created site.

The `run` command is responsible for creating the last data structures for running a site. Those are the heap (which includes the `runqueue`); the operand stack, and; the local name array.

Finally, the `quit` command deletes all memory space occupied by the site in the node, contacts the NNS to retract the corresponding sitename, and quits the DTsh. Note that, however, it is possible to load another program, to be executed in the same site after the first one is terminated.

4.3.3 The DiTyCO interface

This thread is responsible for communication between the DiTyCO system and the user, and for the launch of a site in a node. Up to the creation of a site, communication between the user and the node is related only to memory allocation and to registration of information, both in the NNS and in the DT-node. From that point forward, all communication is done between the site and the user (through the DTsh).

The DTi fills in the proper fields of the data-structure that is used, as shared memory space, for allocating users and sites. This is an array that holds several information as the *sitename*, the *socket* and *port* used for communication, and a pointer to the structure holding a site (see figure 4.8 to illustrate this).

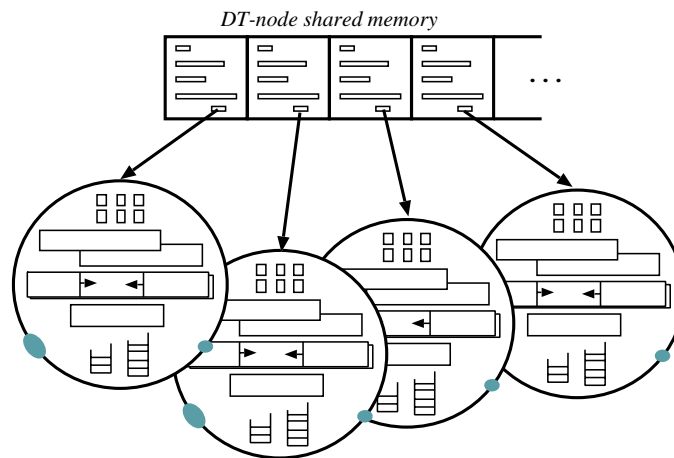


Figure 4.8: DT-node Shared Memory

After creation of the site, and before program execution, a socket for handling I/O from the site is created and its address is passed to the user's shell, closing the initial DTi socket with that user. From that moment, all communication (apart from special shell commands) is passed directly to the site, without involvement of the node interface.

DTi is executed only by one thread. As, it typically deals with I/O from users, there

is a lot of client switching, because the communications socket of DTi is never closed on a specific user. To reduce this burden all user connections are deviated to another socket and all may be attended without delay.

4.3.4 The Proxy

The *proxy* is the module responsible for handling inter-site communication, for the execution of instructions that access the *NNS* and for migration of resources. Internally to the node, the proxy handles special data-structures called *Boxes*. These structures are used by sites to communicate with their proxy, or by the proxy to communicate with local sites. Externally to the node, the proxy handles network TCP data-segments called *packets*. Most of the time the proxy receives a resource targeted to some site. Its job is to receive the packet, to assemble it place it in the incoming queue of the respective site.

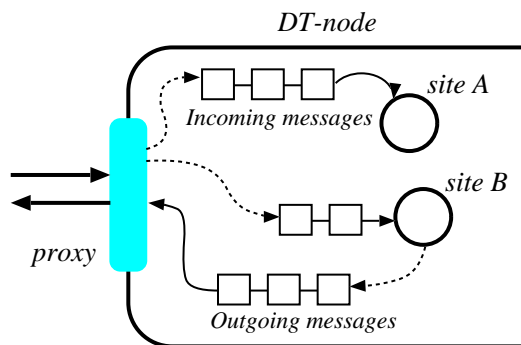


Figure 4.9: The DiTyCO proxy

When a site sends a message to another site, it has first to put the message in the local proxy queue for delivery. Next, it signals the proxy. Then, the proxy reads all messages in its queue and sends them one-by-one. When a message reaches the other node, it is read by the proxy of that node and placed in the incoming queue of the target site.

The proxy is constituted by a set of threads launched when the DiTyCO-node starts execution. One thread operates the proxy's *incoming queue* where messages sent by the sites lying in the node, are placed. The remainder of the threads handle incoming communication to that node. Currently, the proxy uses four threads for receiving

packets.

The job of receiving a TCP connection takes more time than using shared memory. Moreover, the assembling of all packets received, makes the process of receiving messages from the network, more time demanding than communication through shared memory. To favor efficiency in the proxy, we use several threads and a strategy of awakening threads as they are needed to handle remote communications. The strategy is based on a lock around the `accept` command of the socket [80]. The C code looks like this:

```
while(true) {
    // make sure only one thread is accepting connections
    lock_mutex(&lock);
    connection = accept_remote_connections();
    unlock_mutex(&lock);

    // read and deliver the message to DTvm
    receive_message(connection);
    close(connection);
}
```

Inside a node, communication occurs between sites and the proxy in the form of *internal messages*. That exchange of information is done through a special data structure called a *Box*. It includes fields to indicate which kind of request is demanded, information about the location of its sender, size of information carried and, a pointer to a heap frame, translated and packed. The box data structure also has some other fields used to communicate with the proxy, and to send information to the NNS.

The box data structure is as follows:

```
typedef struct Box {
    struct Box *prev;           // list of boxes
    struct Box *next;

    int code;                  // indicates the type of message
    int location;              // location of the site in the node
    int index;

    char varname [MAXNAME]; // to fill in export and import cases
    char sitename [MAXNAME];

    long size;                 // size of data
    void *data;                // data to transfer (usually heap frame)

    int port_address;          // socket parameters for I/O
    struct in_addr ip;
    char foo[4];               // just to pad for memory alignment
}BOX;
```

Each box represent either a message sent to a remote site in which case it is placed on the proxy's outgoing queue, or a remote message received by the proxy to be placed in the local site's incoming queue. These messages are *complete* in the sense that no message is scattered among several boxes – one single box is capable of holding entirely any message.

Externally, things are quite different as messages must all be of the same size. The proxy uses the TCP/IP protocol for packet passing. The two usual function calls for dealing with sockets are `read` and `write`, which need the *size of the message* as a parameter.

Therefore, the `read` function must know in advance how long the message is. We chose to use messages all of the same size, and use more than one message if the content does not fit entirely in a single message. This kind of external message is described below by the following data structure:

```
typedef struct {
    int code;           // type of message
    long size;         // data size present in buffer
    long sizeall;      // total size of fragmented pieces
    int fragmented;    // check for another packet
    char buf[PACKET_SIZE]; // packet buffer
}PACKET;
```

All external communication between nodes is done by sending this kind of packet. It is part of the communication's protocol between DiTyCO nodes to completely receive `sizeof(PACKET)` bytes of data for every packet read. Whether one of those packets is sufficient to enclose the information sent by a site or not, it is a matter to a different procedure that is executed before writing the packet to the socket – the *fragmentation*. The operation is performed by the proxy as a mean to standardize the size of messages transmitted by each node. Fragmentation is discussed in greater detail in section 4.4.4. For communication with NNS there are other type of packets – one for sending requests to the NNS, and another to receive the respective answers. Packets sent from sites to the NNS are described below:

```
typedef struct {
    int code;           // code for the message
    struct in_addr ip; // IP address buffer
    int location;      // site location inside the node
    int index;         // exports table index
}
```

```

    char sitename[MAXNAME];
    char varname [MAXNAME];
} NNS_Incoming;

```

The above described packet is usually used to encapsulate queries (imports), asserts (exports) or retracts to the NNS. A retract occurs when a site finishes execution and its sitename and respective exported names need to be removed from the network. In the case of a query to the NNS, a reply packet is sent to the site, whose structure is described below:

```

typedef struct {
    int code;           // reply code message
    int result;        // result code
    char error[LINESIZE]; // error message
    int location;      // site location inside the node
    int index;         // exports table index
    struct in_addr ip; // IP address buffer
    int loc_origin;    // original location
} NNS_Outgoing;

```

These two data structures are used to communicate with the *NNS* in the form of TCP packets with no fragmentation. Actually, every type of information to be passed, and received from the NNS, fits on these data structures. Thus, fragmentation does not occur in any communication involving the NNS.

4.3.5 Internal Queues

Internal messages holding the information placed in boxes are enqueued either in the *outgoing queue*, owned by the proxy, or in an *incoming queue* owned by the respective site.

The proxy places new messages for the sites in their respective incoming queues, to be read and processed, and sites place messages in the outgoing queue, to be read by the proxy and, eventually, transmitted to another proxy, or to the NNS. Therefore, the proxy is the *producer* for the incoming queue and the *consumer* for the outgoing queue, while a site is the *producer* of the outgoing queue, and the *consumer* of the incoming queue.

Access to those queues is protected from a *race-condition* by using locks and monitors in each queue. These data structures are capable of providing exclusive access to the

queue, as well as, maintaining the number of messages, store them in order of arrival, and to signal the owner upon message arrival.

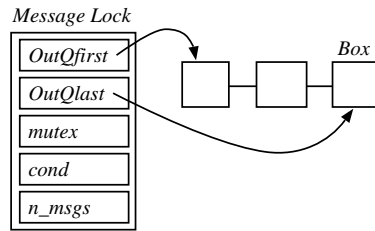


Figure 4.10: Internal Message Queue

The node's outgoing queue is created together with the node, as it is part of node's shared memory space. The incoming queue is part of every DTvm and is created during site creation.

4.4 The Network Layer

This layer provides the means for names and definitions to be made public and accessible by any other site that knows about them. Explicitly exported names and sites are registered at the Network Name Server (NNS). In the network layer, channels and definitions are identified through a network address implementing the σ translation of the base calculus (cf. page 61). All arguments and free variables of messages, objects and definitions, must be submitted to this translation before entering the network. After reaching the other node they are translated again reflecting the new location. The implementation of this translation function is described in section 4.4.2.

4.4.1 The Network Name Server

The NNS is a service available in the network to resolve names of channels, definitions or sites. It is implemented as a process that receives requests for making names available in the network, for giving the full address of a name and, for retracting names from the public domain. We list the NNS main activities:

- **Register a sitename:** this is requested upon creation of a site. It is the `site` command of the DTsh that starts this operation. However, at that moment only

the sitename information can be asserted. It is necessary to wait until the launch of the site to complete the assertion operation. Therefore, the site information is stored in a *standby state*, waiting for a acknowledgement message from the site to complete the registering procedure.

- **Assert a channel or definition name:** the execution of the `export` instruction of a channel or definition sends an export message to NNS, requesting the name to be made publicly available. That single message is sufficient for storing the network address for the name.
- **Retract a sitename** when a site no longer exists its reference must be retracted from the network by deleting it from the NNS. This happens when the DTvm running the site terminates – usually by finishing the program.
- **Retract a channel or definition name.** This command removes channels and process definitions from the NNS tables when a program finishes.
- **Resolve a name.** Returns the network address of that name, usually as a reply to a query as the result of an import operation.

The NNS maintains the information about *sitenames*, and *varnames* (either for channels or definitions). This information is composed of the location of the node (an IP address), an identifier for the site inside the node, and an index for the name at the site's export table. To reduce redundancy and improve coherency, we created two tables to store the above information: one for sites, and another for exported names. Those tables are used jointly to provide replies to queries about addresses of channels and definitions, and to track the location of sites.

In the site's table, the tuples include the lexeme that identifies the site in the source program (the key attribute), a site identifier (an integer) to locate the site inside the node, and the IP address where that node is located:

SiteTable: SiteName \mapsto SiteId \times IPAddress

The key for the table of exported names is compound and it uses the lexeme for the identifier plus the *sitename* of the site it belongs to. Besides the key, each tuple also contains an index to the exports table that corresponds to a heap position in the respective DTvm.

ldTable: SiteName \times ldName \mapsto Heapld

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

(ldTable(server,appletserver), SiteTable(server))

Frequently, the process of registering a site name at the NNS is not possible because the *name* may be already in use. This happens much in the same way as domain names for the DNS on the Internet. The usual procedure is to deny the registration of that name, informing the user that the name is being used. This is also what happens in the DiTyCO System. Whenever a chosen site name is being used, the result of the `site` shell-command is an error, and the user is asked to provide another site name.

In the current 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.

4.4.2 Translation of Names

Before being sent to another site, messages, objects and definitions must be translated, in accordance to the function described in section 3.3.2. This translation must reflect the network address of channels and definitions, which no longer can be referred locally by a memory pointer or an entry in the local exports table. Instead, they must have an independent memory address constituted by the index (the entry in the exports table pointing to a memory address on that site), an identifier (referring to the particular site in that node), and the node's IP address. This information can not fit in a single word of 32 bits long. Therefore, we use two words for each *translated name*. The index and the localization are stored in the first word, and the IP address, in the second word, taking 64 bits to store the full address as illustrated in figure 4.11.

The translation procedure is applied to free variables within resources, and not to code. In a message frame, the translation function, only has to be applied to the arguments

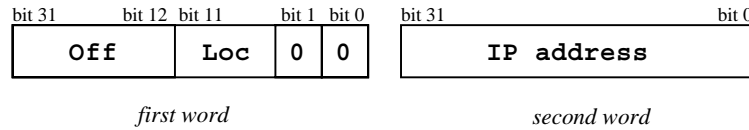


Figure 4.11: A Network Address Representation.

part of the frame. In a similar way, in an object frame, only the free variables part of the object is subject to the translation function. The definition frame is formed of free variables that appear in the definition, thus, the whole frame is translated.

During the translation procedure, if the function finds a local channel (a pointer to a channel in the heap), that channel is *implicitly exported*. This makes that channel appear as an entry in the site's local export table. This is necessary because, that channel may be used in the remote site.

4.4.3 Packing and Unpacking for Communication

Information to be sent to another site is packed by the sender site. The box structure is created and filled according to the type of information that is to be sent. The filed `msg` is set to a newly created buffer of the proper size to hold the message, object or definition to be sent. The computation of the size, translation and packing, is a one-step procedure for definitions and objects and, a two-step procedure for messages. The reason is that it is necessary to go along through all arguments and check if they are strings, in which case their size must be added to the total size, before everything is really packed. Strings are passed by value and thus needs to be packed in the message as well.

String frames are included at the end of the message following the order they appear in the arguments. The pointer `first`, to provide the means to enqueue resources in a channel, is no longer a pointer (a zero is stored in its place), but still maintains its relative position on the frame, to standardize the format. The pointer to the label is also translated into an offset to the place where the method name is stored. In figure 4.12 we have a message frame for a method invocation having 4 arguments (a local name, an integer, a string and a remote name). Those arguments are translated as follows: the local name into a network address; the integer is maintained; the pointer to the string frame into an offset to the place where the string was appended, and;

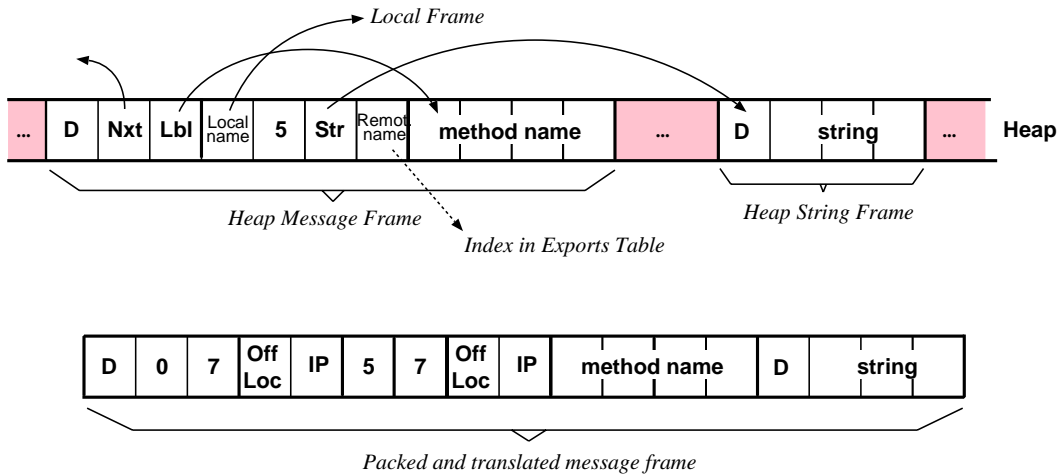


Figure 4.12: A message translated and packed

the remote name is stored in the form of a network address (occupying two words). Note the two sevens representing the number of words until the method name, and the number of words until the string.

Objects are treated in a similar way, but there are no strings to deal with. Therefore, we must translate only free vars, pack together the byte-code for the object in the end of the message, and transform the next pointer, and the pointer to the byte-code, into the proper offsets.

The method table of the object is built from the compiler to use only offsets to the place where each method is stored. Besides that, each object is compiled in an “nested” approach, so that all code belonging to a method is stored in the same nesting level. This reduces the complexity of the search and packing of the object and its method table. Figure 4.13 shows an example of the translation of an object frame

The final case, the definition, is the simplest of all because it consists of just one frame with the instance arguments. Therefore, we only translate these arguments. There are no pointers that must be converted to offsets, no strings and no byte-code.

At the end of the translation and packing procedure, the proxy receives in the outgoing queue, a box with several fields filled, and eventually, in the case of a message, object or definition, a buffer that resembles part of the heap, constituted by the *translated and packed heap frame*. If this message is to be sent to another node then, after the socket writing, this memory space is immediately reclaimed, if not, this box is just placed in

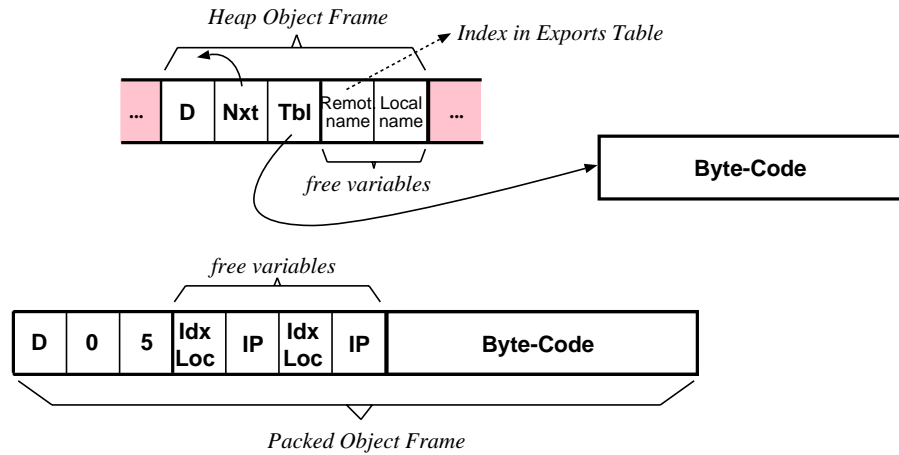


Figure 4.13: An object translated and packed

the respective target site incoming queue, taking advantage of shared memory.

4.4.4 Fragmenting and Assembling Messages

Reading and writing *BSD Sockets* is a procedure that takes three non-empty arguments: the socket identifier, a buffer to place the data to be read or written and, the number of bytes to be processed. Therefore, in a read operation, we must know in advance how many bytes are to be read. This condition forces us to create a *message-passing protocol* in which, messages transferred must have a pre-defined size, such that, the other end knows how many bytes it has to read.

As discussed in section 4.3.4 (page 90), we created a special data for transference of messages between different nodes called *packets*. This type of structure holds a buffer in which code or dynamic heap structures are passed to another site. We discuss in this section how, messages that are longer than this buffer, are handled.

The `packet` data structure has a static buffer. It is in that buffer that the structure `BOX`, plus the heap data, will try to fit. Whenever this does not happen, then, more than one packet is needed to be sent. To illustrate this suppose we are going to send to another site an object, which has two free names in its body, as illustrated in figure 4.14:

This object frame must be packed in a `BOX` structure and placed in the local proxy's outgoing queue. The data encapsulated in the box structure includes the heap frame

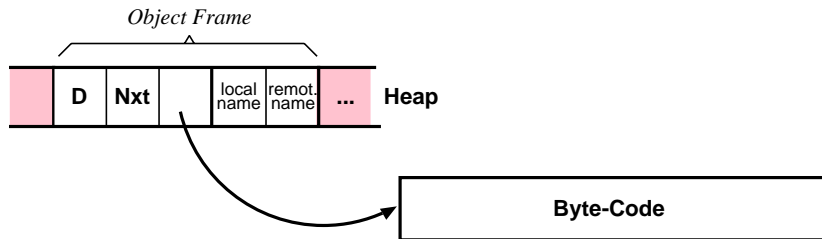


Figure 4.14: An object to be sent to another site.

for the object, as well as, the byte-code for the object method's definition, as illustrated in figure 4.15. During packing, translation of names must be performed. The local and remote free names are translated into a network reference in the form of a triplet (IP address, localization, offset). Both local and remote names must be converted into a network address. Note, also that the frame pointer to the byte code is converted into an offset (five words) to the corresponding place in the current buffer.

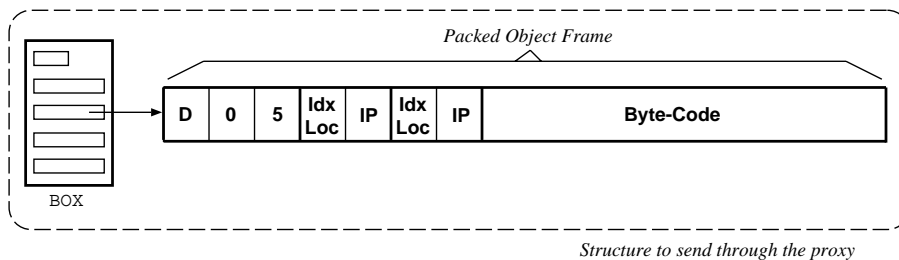


Figure 4.15: Box holding an object.

The codification of the triplet (IP address, localization, offset), is made in 64 bits. We place the IP address in one word, and share a second word between the localization (10 bits) and the offset (22 bits). Therefore, in a packet, network references occupy the double of the space.

The proxy, after receiving the box, checks the size of the buffer to see if it fits in a single packet. Let us assume that it is too large to be held in a single packet buffer. Therefore, fragmentation will be needed and several packets will have to be transmitted. The box header structures and part of the of the box's buffer are placed in the first packet; the remaining of the buffer is distributed by other packets, and transmission is done in the order of the packets. This situation is depicted in figure 4.16:

Therefore, the proxy will transmit to the other proxy only messages (packets) of a fixed size. Each packet has a field that corresponds to the fragment state. If the current packet is not the last, then that field will hold a 1. If the message is the last

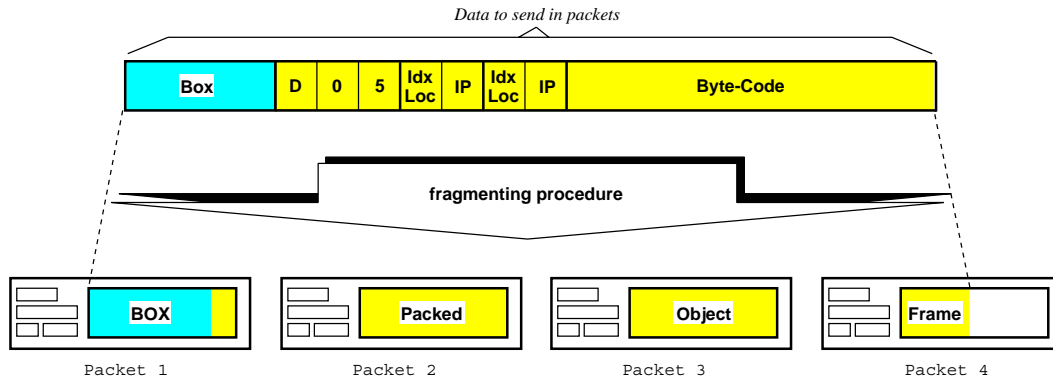


Figure 4.16: The fragmentation procedure of an object in a Box.

one, then the fragment field will hold a 0 (corresponding to *no fragmentation*). Each packet also has the size of the data occupying the current message's buffer, as well as, the total size of the transmitted message. This last field provides a means to allocate a buffer in the other end, right from the first message for receiving and assembling the whole information transmitted.

As the message is being received, it is also being assembled by the remote proxy in a place that will mimic the sender's box buffer. After all packets have been received, the remainder of the newly created box structure are filled and the message is placed on the incoming queue of the target site.

4.5 An Example

In this section we present an example of the construction of frames in the heap and in the run-queue at the DTvm during execution of a very simple example program. However, the program features the most frequent constructions in a program: creation of channels, sending remotely messages and reduction.

Site A	Site B
<pre> export a a ? {ping(x) = x ! pong[]} ... </pre>	<pre> import a from A new b a ! ping[b] b ? {pong() = io ! put["Round Trip"]} </pre>

Execution at site A begins with an export instruction, for which a channel frame is created in the heap; after that, an object frame is placed in the heap as well. This heap configuration is illustrated in figure 4.17.

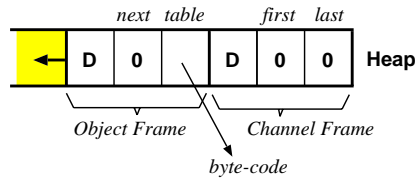


Figure 4.17: A channel and an object are created at **Site A**

Note that, the object frame has no space for free variables as they do not exist in the body of the object in this example. The program at site A now continues normal processing. Meanwhile, at site B, channel `a` is being imported and the full network address for that name is stored in a heap word (reserved by the instruction `nimports`). A new channel (`b`) is locally created, and a message frame is also created in the heap (despite it will not reduce in this site, but sent to another). This situation is illustrated in figure 4.18.

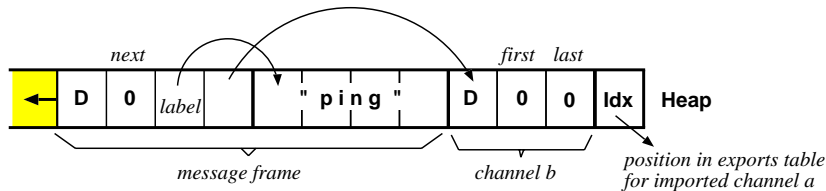


Figure 4.18: An imported channel, a local channel and a message at **Site B**.

Next, an object is enqueued in channel `b` at site B. This object, also, has no free variables and is placed as a frame in the next available positions in the heap as depicted in figure 4.19:

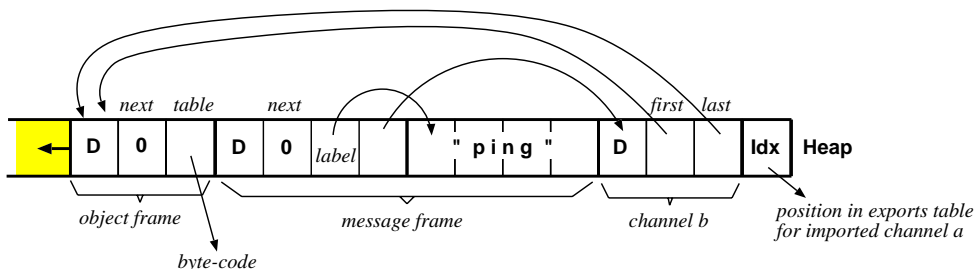


Figure 4.19: Enqueueing an object at the local channel in **Site B**.

We do not lose generality by choosing this particular order of execution (creating the message before the object). Hence, supposing that the DTvm will try to reduce the message created at site B, and finds out that this message is meant to be sent to another site (to site A). Then, the DTvm at site B creates a special buffer, packs the message, sends it to the local proxy, which in turn sends the message to site A or to another proxy (depending on site A being local or lying in another node). The buffer thus created is of the form illustrated in figure 4.20.

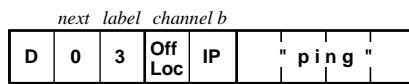


Figure 4.20: Message frame packed and translated.

Upon checking its incoming queue, site A, finds out that there is a message to be read. When processing this message, it re-translates it and checks what to do with it - in this case, it is to be sent to local channel a, which has already an object there, and therefore a reduction may happen. In the figure 4.21 we show the frame created in the run-queue for this reduction.

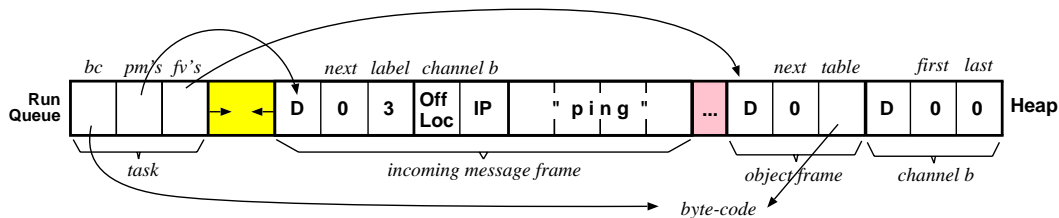


Figure 4.21: Creation of a reduction frame in the run-queue at **Site A**.

At this point, in site A, the reduction occurs; a frame is created in the run-queue, and eventually that new task will be picked for execution. Then, execution will continue from the position in the byte-code pointed by the first word of the frame. That position is the first instruction of the method selected by the incoming message of the existing object. The code will then form a new message frame in the heap invoking method pong.

The new created message is to be sent to a channel used as an argument to the previous method invocation that, after instantiation is channel b from site B. Therefore, this message needs to be translated, packed and sent to site B. After being received by site B, the emulator finds that the message is to be placed at local channel b. However,

in that channel there is already an object, and therefore a reduction will occur. We show now the run-queue configuration at site B, after this reduction stage.

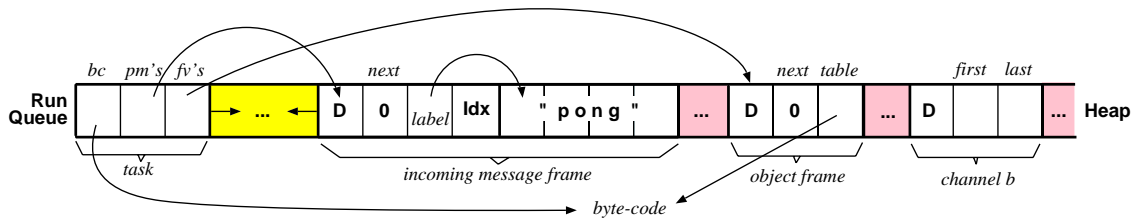


Figure 4.22: Creation of a reduction frame in the run-queue at **Site B**.

When this run-queue frame is picked for execution, the instruction to be executed is the first instruction of the method (`pong`) invoked by the incoming message. This method makes use of a built-in object to provide basic I/O. In this case, it will output the message “Round Trip”. Then, the program ends because no more tasks are found in the run-queue.

4.6 Chapter Overview

In this chapter we presented the details of the DiTyCO architecture and runtime engine. We began by describing the changes made to the original TyCO engine at the site layer; then, we presented the fundamental issues at the node layer, showing the interaction between the proxy and the site; finally, at the network layer, network addresses, sending and receiving packets and the NNS were also addressed. In the end, we illustrate the main ideas of this chapter, presenting an example to show what happens in the DiTyCO data structures, when two sites communicate.

In the next chapter we will focus the attention on the aspects concerning the mobility of resources in DiTyCO.

Chapter 5

Mobility

The DiTyCO language and run-time system supports the concept of mobility. This means the ability to dynamically move resources between network locations. Resource mobility may be broadly classified into two categories: *weak mobility*, meaning that the resources are blocks of code, and; *strong mobility*, meaning that the resources are complete computations [34].

In DiTyCO both types of mobility are present [30, 53]. Weak mobility is induced by the lexical scope of channels and definitions. On the other hand strong mobility is introduced in the language through a new construct that allows a site to move to another IP address any time during its execution. In this chapter we describe an implementation of weak and strong mobility in DiTyCO.

5.1 Introduction to Mobility

Mobility within the context of a distributed system means the transference of resources, code or computations, between two or more execution units. Nowadays, computers are commonly connected to networks and, exchange of data between different execution units implies, in most cases, data exchange between different computers. Thus, the word “mobility” is generally used in a distributed environment; and not in the particular case of several execution units sharing the same physical machine.

In recent years programmers no longer restrict themselves to the transference of static

data, but also create imaginative ways of passing executable code from one execution unit to another. The term “mobile code” is recent and there still is a lack of a commonly accepted and sound definition for it. Even nowadays, the expression “mobile code” is used in the literature with significantly different meanings [7, 17, 34, 74, 84].

The way data is passed between execution units (sites in the DiTyCO model) may be classified under specific methodologies which create software architectures with similar characteristics (as referred in [6]). Software design paradigms for code mobility evolved from the traditional *client-server* approach into more evolved concepts involving migration of resources, code and even entire executing programs. Three main design paradigms exploiting code mobility are identified by Carzaniga et al. [22]: *remote evaluation*, *code-on-demand* and *mobile agent*.

The *client-server* paradigm (CS) is a well-known and widely used approach to build applications that interact over a network. In this paradigm an execution unit offering a set of services is placed at a site (the server). Resources and the know-how needed to accomplish these services are also hosted at the same physical machine (node in the DiTyCO model). The client site located at another node interacts with the server site to request a service. The request is handled by the server using its know-how and its resources. In the next interaction between the two sites, the server delivers back the result to the client.

In the *remote evaluation* paradigm (RE) a site A has the know-how necessary to perform a service, but it lacks the resources required, which happen to be located at site B , in a remote node. Therefore, A sends the know-how (code) to site B which, in turn, executes the code using the resources available locally and delivers the results back to A .

In the *code-on-demand* paradigm (COD), a site A is already able to access the resources it needs, which are co-located within its node. However, it has no information about how to manipulate such resources. Thus, A interacts with another site, B , by requesting the know-how. The second interaction takes place when B delivers the know-how to A (objects in the DiTyCO model), that can, subsequently, use it.

In the *mobile agent* paradigm (MA), the know-how is owned by a site A , which is initially hosted by node N_A , but some of the required resources are located at another node N_B . Hence, A migrates to N_B carrying the know-how and possibly some

intermediate results. After it has moved to node N_B , A resumes its execution using the resources available there.

The MA paradigm differs from other mobile code paradigms since the associated interactions involve the mobility of a running site. Hence, while in RE and in COD the focus is on the transfer of static code between running sites, in MA a whole site is moved to a remote node, along with its state, code and some resources. The paradigms CS, RE and COD can be coded with weak mobility mechanisms, while MA is only directly coded with strong mobility.

5.2 Mobility in DiTyCO

DiTyCO features both code migration (weak mobility) and migration of a whole site (strong mobility). Migration of resources is the transfer of messages, objects or instances from a site to another. Those sites may reside in the same node or lie in difference nodes (that is, in different IP machines). Migration of a site is the ability of a site to suspend execution in a node, move to another node, and resume execution thereof.

DiTyCO is an extension of TyCO, adding distribution and mobility features to the inherent concurrency nature of its base calculus and language. Indeed, just by adding the capacity of referring to channels and definitions created in remote locations (i.e. another node), mobility of resources emerges naturally from the lexical scope, which in this case is augmented from the single machine where the program is being run, to the whole network. Thus, we created the appropriate reduction rules according to this approach.

Mobile code in DiTyCO is characterized as *network aware* because initially the programmer builds the DiTyCO program interface specifying communication points either at local or at remote locations. However, thereafter, all communication is independent of the location of resources.

Several taxonomies have been proposed [34] to classify systems with code mobility. Among the parameters used on these, we have weak and strong mobility – both present in the DiTyCO language. However, in the calculus, only weak mobility is considered. Strong mobility is introduced through a new construct, which has no counterpart in

the calculus.

In DiTyCO we have primitives that implement mobility of messages and objects, that import definitions, and remotely instantiate them. It is possible to create typical *client-server* applications, to recreate a *remote evaluation* situation, or even to make use of the *code-on-demand* paradigm. However, these basic operations do not allow, programs to be created in DiTyCO according to the *mobile agent's* paradigm. It is not possible to model the movement of a program from one location to another only by using weak mobility primitives, as you must access site run-time state to be able to do that.

The mobile agent paradigm requires that the above programming functionality be augmented to allow computations to move near to the resources in order to diminish the bandwidth requirements. Typically, communication in the standard client-server model takes the biggest time slice of the execution time. This is due to the delay of the network and to the available bandwidth. By allowing the server to move near the client, or vice-versa, it should be possible to overcome this communication problem as proposed by Baldi and Picco [12]. As a mean to experiment this new paradigm, we decided to include in DiTyCO a strong migration primitive.

Strong migration can be characterized as either being *reactive* to some command external to the program being executed, in which case it is known as *objective migration*; or *proactive* when it is triggered by the program itself, known as *subjective migration*. In DiTyCO, site migration is triggered by the execution of a `go` instruction, and therefore it is subjective.

Weak migration may be supported either in a *synchronous* or *asynchronous* way, depending on whether the program transferring the resource, blocks (suspends) or not, until the code is executed (received). In DiTyCO resources are moved according to the asynchronous nature of the base calculus. Conversely, its strong mobility feature provides synchronous migration, as sites are suspended until they reach the target node and their execution is resumed.

Weak mobility may also be characterized according to the direction of the code transfer. A site (execution unit) can either *fetch* the code to be dynamically linked and/or executed, or *ship* such code to another computational environment to be executed remotely. In DiTyCO, we followed the code *shipping* approach for the reasons

described below.

5.3 Shipping versus Fetching

In DiTyCO messages and objects are placed in some channel that may be defined in a remote site. This triggers the movement of those messages and objects to that remote site in order to reduce them locally; with definitions the situation is similar: the instantiation of a definition that is defined in a remote site induces the movement of that instantiation parameters to the location where the definition is defined. Therefore, in all reduction situations we have *code shipping*. Contrary to DiTyCO, in many other systems the code is *fetched*. This is particularly evident for definitions or classes in Java, where the client downloads (*fetches*) the code from the site where they are defined, whenever it needs.

In this section we present aspects that are tied to our decision of taking the shipping approach instead of the fetch approach to definitions.

5.3.1 Characteristics of *ship* and *fetch*

We compare and synthesize the characteristics of each approach with respect to the instantiation of remote definitions and assess their impact on the DTvm. We start with the *fetch* (or *download*) case:

- it is necessary to search for the byte-code and pack it together with the respective heap frame before sending to the client;
- in case of a definition, two messages are necessary before reduction occurs: one from the client to the server to request the definition; and the answer from the server with the correspondent code;
- if the definition's code includes an instantiation of another definition, then we recursively have to fetch it before the instantiation may proceed;
- from the programmers point of view, the way the resources migrate is different for definitions, from that one for messages and objects;

- free variables in definitions are typically references to channels or to other definitions, therefore, this approach ultimately increases the number of packets transmitted from one site to another, as the referred resources will most likely need to be moved as well.

Many of the above characteristics may be presented in the *ship* (or *upload*) approach using their counterparts. We skip some of those possibilities and only give the most relevant ones.

- translation only occurs for instantiation arguments (into network identifiers)
- only the arguments frame needs to be sent (no code is moved)
- the byte-code stored in the server stays in the server and there is no propagation, or copy of code
- there are no security problems of running malicious code as the code is run in the server where it was created
- there is an increased coherency in migration of resources, as messages, objects and instances, all move to the place where the channel or definition, is located

5.3.2 Implementation Issues

In what regards the implementation requirements the upload approach is simpler to implement compared to the download approach, mainly for the following two reasons: first, it is a simpler communication pattern as there is only one message to deal with and there is no byte-code to move and second, it allows for the upload of a definition using only objects, as we will show.

The situation we want to model is the creation of a definition in one site (**Server**), and the import and use of that remote definition in a second site (**Client**):

Server	Client
export def X (\tilde{x}) = P in ...	import X from Server ... X[\tilde{v}] ...

In this example, site **Client** imports the definition of X from site **Server**, to be used thereafter with some value \tilde{v} . This command triggers the *upload* of the instance to site **Server** where it is instantiated locally.

Now, let us rewrite this example avoiding the use of external definitions. We have to do something that recreates the same behavior of the previous example with the *upload* of the definition. That is, the definition has to be instantiated, in the site **Server**, with arguments passed by the site **Client**:

Server	Client
export channel def X (\tilde{x}) = P in def Y (a) = { a?{ (\tilde{x}) = X[\tilde{x}] Y[a] } } in Y[channel]	import channel from Server ... channel ! [\tilde{v}] ...

As we can see, we use another definition in site **Server** that maintains an object with a single method that can be used to instantiate the local definition with the arguments coming from another site (the site **Client** in our case). Basically, we add to site **Server** another definition that acts as an *helper* to recreate the upload of definitions. For each definition X_i exported, there must be a method that will instantiate it with arguments coming from another site. All those methods may be grouped under a single definition in an object.

This mechanism can be automatically implemented by the compiler. The code generated for the site, which has an exported definition, has to include an *helper* definition,

instantiated on an exported channel (say, `exported-definitions`).

On the **Client** side, instead of producing the code $X[\tilde{v}]$ for an imported definition, it only needs to produce the code `exported-definitions![\tilde{v}]`.

In this discussion we intended to show that it is possible to model the upload of definitions using only objects. Actually, a definition may be seen as a persistent object, and an instance as a message invoking a method on the persistent object. This feature allows us to derive a compiler that reformulates the code in order to provide a definition of mobility without an implementation of the specific case of uploading definitions. Note, however, that there is not a full substitution of definitions by objects, as definitions can be polymorphic, but objects can not.

5.4 Weak Mobility

Weak mobility refers to the movement of stateless computational components in a network. In the sequel we describe with more detail each aspect of the system's architecture involved in the movement of these resources, and we show what actually happens when an object, message or definition has to move between sites in order to obey the lexical scoping discipline.

5.4.1 The Applet Server Example

To better describe the algorithm developed to support weak mobility in DiTyCO we shall use an idealized “Applet Server” example.

We are interested in showing what happens when the system is moving resources between sites. In the example, the **Server** is located at some site, and has a list of applets that can be downloaded to **Client** sites upon request. This example illustrates the mobility of messages from the **Client** to the **Server** (requests for applets), and the mobility of objects from the **Server** to the **Client** (the requested applet).

The AppletServer example can be written in DiTyCO like this:

Server	Client
<pre> export appletserver def AppletServer (self) = self ? { applet₁(p) = p?(x)= P₁ AppletServer[self] ... applet_k(p) = p?(x)= P_k AppletServer[self] } in AppletServer[appletserver] </pre>	<pre> import appletserver from Server in new y appletserver!applet_k[y] y![v] </pre>

The **Server** holds a definition of an object whose methods are the *applets* available for downloading. This definition is instantiated in a local channel (`appletserver`) exported to the network.

From the **Client**'s point of view, what is needed is to import the public channel `appletserver` to connect with the **Server**; a message to invoke the proper applet and, the local channel `y` (a handle) where the applet can be downloaded and be later executed.

In this example, the **Client** imports a public channel `appletserver` to connect with the **Server**. Using this channel the **Client** sends a message to the **Server**, invoking a method to be executed by the **Server** and passing along a reference to a local channel. Once that method is executed, an object (the requested applet) is placed in the channel passed by the **Client**. This process triggers the movement of the object to the **Client** through lexical scope.

Throughout the example there are three reductions to execute the code of the **Client**'s program. These involve the exchange of two resources between the **Client** and the **Server**. Below, we list each of those reductions:

1. A method invocation on the **Server**. That particular method is the one with the applet required by the **Client**.

2. The transfer of the applet from the **Server** to the **Client**. This is accomplished when the object $p?(x) = P$ is placed in channel y at the **Client**'s site.
3. The local execution of the applet, at the **Client**, using the received object from the **Server**, through a local message/object reduction.

5.4.2 The Migration Mechanism

This section describes the steps required to move a resource between the client and the server site as illustrated in the *Applet Server* example. The DiTyCO architecture includes a proxy, in each node, responsible for the inter-site communication. Every resource, when moved from a site to another site, has to pass through that communication proxy, as depicted in figure 5.1.

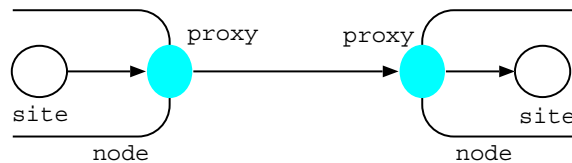


Figure 5.1: Site communication.

The entities involved in this process are: the sending site, the local proxy, the remote proxy (if the transaction is with a remote site), and the target site. Hence, the migration mechanism used in DiTyCO has four steps:

1. In the sending site:
 - (a) create a data structure for intra-node communication;
 - (b) pack and translate the resource;
 - (c) enqueue the message in the proxy's outgoing list.
2. In the sending Proxy:
 - (a) create a data structure for inter-node communication;
 - (b) send the message, fragmenting it in packets if necessary.
3. In the receiving Proxy:

- (a) receive the message, assemble packets as needed;
 - (b) put the message at the site's incoming queue.
4. In the receiving site:
- (a) upon reaching a `ret` instruction, read the queue of incoming resources;
 - (b) process every message in the queue and reduce if possible.

We describe these four fundamental steps in more detail in the following section.

5.5 Implementing Weak Mobility

In DiTyCO weak mobility is expressed by migration of resources, that is, migration of messages, objects and instances. When a resource migrates from one site to another, names (references to channels or to definitions) are translated in order to reflect the new location, while preserving the bindings to the site where they were created. The translation is done on these references sent in copies of heap frames.

Resources are passed in the network in the form of TCP packets. Inside a node, resources travel from sites to the local proxy (and vice-versa) in the form of special data structures, taking advantage of shared memory and use of threads.

5.5.1 The Machine View of the Applet Server

We return to the Applet Server example to explain the procedures involved in the movement of a resource from one site to another. We use the assembly code (and not the byte-code only because the former has a human-readable form) for the example, as it unfolds many aspects of the program execution that otherwise would be hard to follow in the DiTyCO code. Figure 5.2 illustrates the top level of the generated assembly code for the Applet Server DiTyCO example.

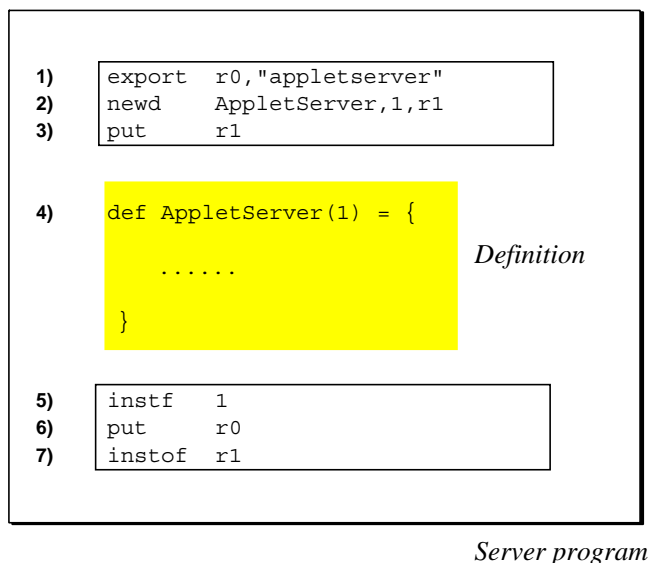


Figure 5.2: Assembly code for the Applet Server example.

5.5.2 Initial Setup Phase

The first instruction – `export` – plays a double role: it creates a channel in the local heap, and makes it public by exporting it into the *Network Name Server* (NNS). The `export` is a synchronous instruction: the virtual machine waits for the acknowledgement of the NNS before continuing. The execution of the `export` involves placing a message in the nodes’ proxy outgoing queue, sending a message to the NNS, receiving the acknowledgement from the NNS, the placement of this message in the sites’ incoming queue and its later reception by the DTvm.

The `newd` instruction (line 2 of the code), prepares a frame in the heap to store a pointer to the definition’s byte-code and the free names that occur in the body of the definition `AppletServer`. It also sets register `r1` to point to this newly created frame. This situation is illustrated in figure 5.3. The new frame is filled with the free names that occur in the definition (plus itself). This is done using the `put` instructions (line 3).

The next instruction to be processed is the `def` instruction. It makes the emulator jump over a series of instructions directly to the `instf` (on line 5). This happens because, the body of a definition is only used when it is instantiated.

The `instf` instruction allocates an heap frame for storing the arguments used for the

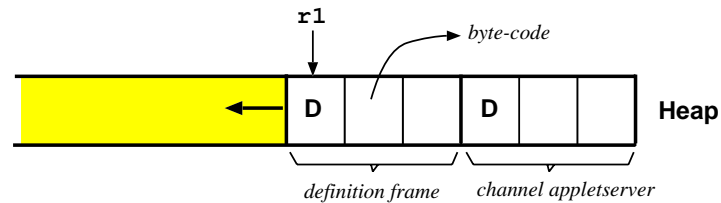


Figure 5.3: Heap configuration at site **Server** after `export` and `newd`.

instantiation of the definition. Those arguments are placed in the frame with `put` instructions.

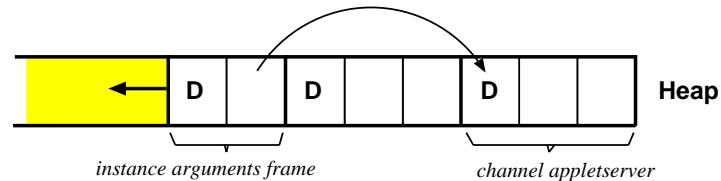


Figure 5.4: Heap configuration in site **Server** after `instf`.

Next, the emulator executes the `instof` instruction (on line 7) to create an instance of the definition. This instruction uses its single argument to check where the code for the definition is located. It may be local (for a local definition) or lying in another site (for a remote definition). Depending on whether the location of the definition is local or remote, different procedures are used: if it is local, then an instantiation is done by creating another DiTyCO task in the run-queue; if it is remote, then the frame storing the arguments of the instantiation is packed and sent to the remote site, where the definition code is located, to be instantiated there.

In this example, the argument of the `instof` instruction is local, therefore a new task is placed in the run-queue, which upon execution will trigger the creation of an object in channel `appletserver`, as described by the assembly code illustrated in figure 5.5.

The first instruction of the definition `AppletServer` is to create an object (`objf`) whose methods are the applets to be downloaded by the client. This object is placed at channel `appletserver` which has already been created in the local heap. The `put` instructions appearing next correspond to the filling of free variables for the object's body.

The final instruction, `trobj`, tries to reduce the object with some message in this case at channel `appletserver`. Therefore, the last memory configuration is the one illustrated

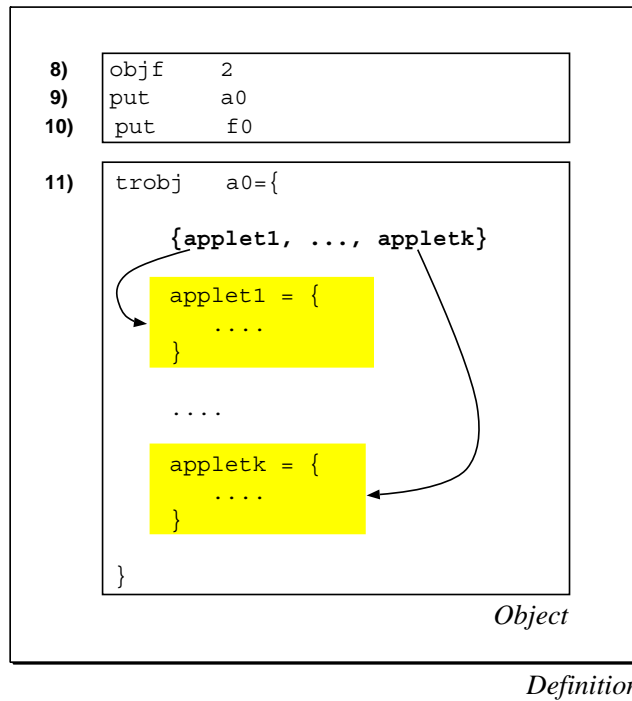


Figure 5.5: Assembly code for the definition.

in the top half of figure 5.6. Note that a task is placed in the run-queue occupying three words, which correspond to a pointer to the byte-code holding the instructions for the definition, the parameters (stored in the instance frame), and the free variables (stored in the definition frame). During the execution of the definition an object is created in the heap, and placed in channel `appletserver`, as illustrated in the bottom half in figure 5.6.

Up to now, the DTvm has not done much yet. Basically, it made available the location where to store the definition `AppletServer` and instantiated it there. On the client side the machine view of the DiTyCO code is:

```

main = {
  nimports 1
  import "Server", f0, "appletserver"
  newc r0 %y, the handle for incoming code
  msgf 1, "appletk"
  put r0
  trmsg f0
  msgf 0, "val"
  trmsg r0
}

```

The client's first instruction (`nimports`) reserves space in the heap to store information

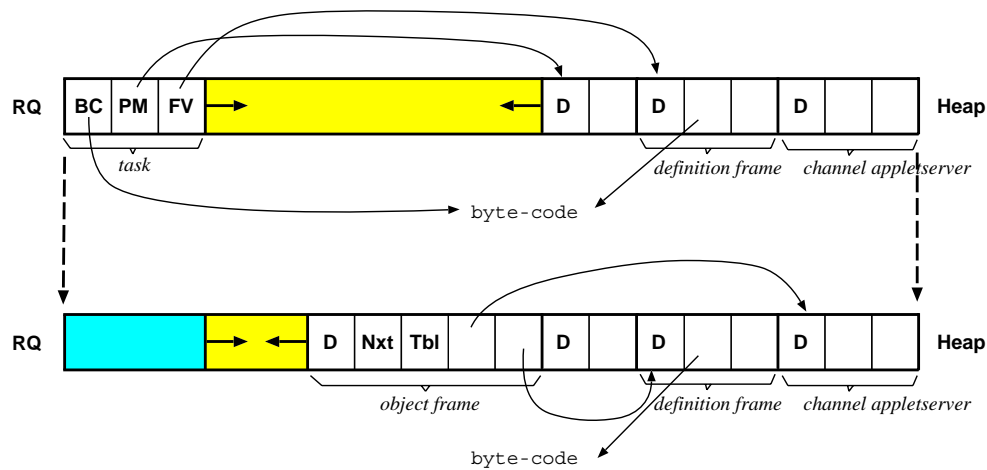


Figure 5.6: Heap configuration in site **Server** after instantiation and objf.

related to the single imported name. Next, comes the `import` which queries the NNS for the name `appletserver` lying in a site **Server**. The returning information is placed in the local exports table, indexed by the value `Idx` stored in the heap position, pointed by `f0`. Then, a new channel is created (known as `y` to the programmer, and as `r0` by the DTvm) to receive the requested applet. A reference to this channel is sent as an argument in the message requesting the applet from the server. The evolution of the heap configuration for the client is illustrated in figure 5.7:

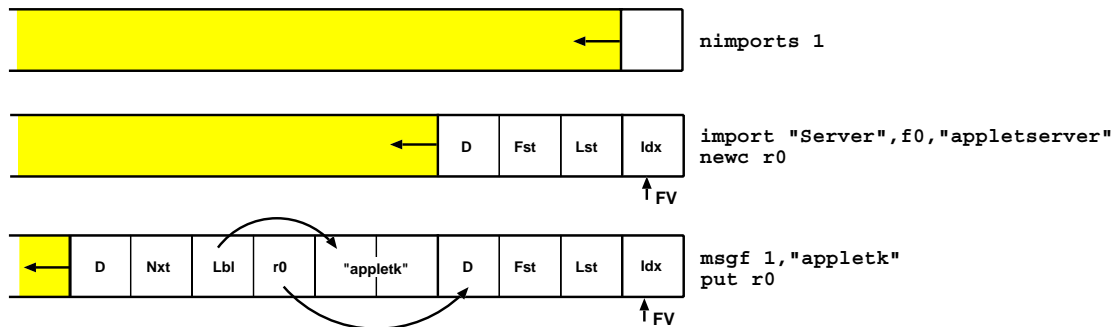


Figure 5.7: Initial Heap configuration stages at site **Client**.

The `trmsg` instruction tries to reduce the message in a channel pointed by a free variable `f0`. However, this channel is not local, as it is defined in site **Server**. This is checked by inspecting the heap word, which is tagged, indicating a position in the exports table, and therefore, representing a network address. Whenever a target channel for a resource is represented by a network address, the resource has to be sent to another site. This process involves three steps:

1. passing the resource from the original site to the node' local proxy;
2. sending the resource from the local proxy to the remote node's proxy;
3. passing the received resource from the remote proxy to the target site.

These steps will be explained in detail in the next sections.

In the remainder of the code, one other message is created to trigger the execution of the received applet. This is achieved by sending a message to the location where the applet will eventually be downloaded (channel y).

5.5.3 Local Site to Local Proxy

At the local site the first step involves packing the message in the Box data structure used for internal node communication (that is, for passing data to the proxy and vice-versa). The packing and translation procedure takes place simultaneously.

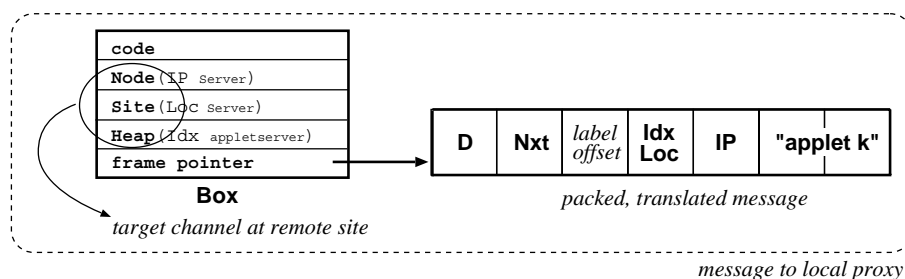


Figure 5.8: Message packed and translated in Box.

The translation process converts names that are sent in the resource frame into network addresses. Local names are expanded into a network address (IP, Location and Heap index) while names already in network address form are kept as they are. During this step, some names are *implicitly exported*. These are names that were not exported in the beginning of the program but; since they are being passed to another site they become remote references (scope extrusion) that can be used in the remote site. Thus, their implicit export provides the way to keep track of them as well.

Remember that the exports table ensures the correct binding between names and their position in the local heap. When a garbage collection is triggered by the lack of heap memory, it may be possible that some channels and definitions stored in the heap be

displaced to another part of the memory. Therefore, the indirection level offered by the exports table provides the correct bindings between network references and their physical memory location.

Back to the Applet Server example, the channel referred by `r0` (in the client's assembly code listed on page 116, and illustrated in figure 5.7) is translated into a network address, and appears in figure 5.8 occupying two words: one word to hold the exports table index (`idx`) for that position, and the location code (`Loc`) of the site in the node; and another word to hold the IP address (`IP`) of the node.

The translation algorithm used for messages and objects is summarized in the following pseudo-code:

```
// Compute space needed to store message frame
n_words = 0 // number of words
for f in Message_Arguments {
  case f of {
    value : n_words += 1 // simple value
    net_addr: n_words += 2 // network address
    channel : n_words += 2 // pointer to channel
    default : // pointer to string
      n_words += stringlength(f) + 1 // strlen plus offset word
  }
}

b = allocate_buffer(n_words)
end = buffer_end(b, n_words) // strings are placed in the end

// Place arguments, translate and convert
for f in Message_Arguments {
  case f of {
    value : copy_word(f,b) // simple value
    net_addr: copy_net_addr(f,b) // network address
    channel : // pointer to channel
      idx = insert_ExportsTable(f) // insert in exports table
      copy_net_addr(inExportsTable(idx), b)
    default : copy_string(f,b,end) // pointer to string
  }
}
set_method_offset(b, f) // offset to methodname
```

This is a two step algorithm on the message frame stored in the heap: the first step is used to compute the space needed to store the message arguments together with possible strings passed as arguments that must be placed in the end of the newly created buffer to store the message frame. The second pass is used to place and

translate the message arguments.

This algorithm is also used for the case of sending instance arguments to another site. The structure is quite similar. The only difference concerns the non-existence of a method label, which is substituted by a reference to the byte-code. The case of an object is, however, different and is summarized below:

```
// Compute the space needed
frame_size = sizeofframe(current_frame)
n_fvars    = numberofFVs(current_frame)
bc_size    = sizeofbc(current_frame)

// each free var will occupy two words
n_words    = frame_size + bc_size + n_fvars
b          = allocate_buffer(n_words)
end        = buffer_end(b, n_words)

// Copy object frame and byte-code
copy_frame(current_frame, b)
copy_bytecode(current_frame, b, end)
set_methodTbl_offset(current_frame, b, end)

// Translate free variables
for f in Object_Free_Vars {
  case f of {
    net_addr:  copy_net_addr(f, b)           // network address
    channel :                               // pointer to a channel
      idx = insert_ExportsTable(f)         // insert in exports table
      copy_net_addr(inExportsTable(idx), b)
  }
}
```

The frame is copied entirely to the buffer and then the free variables of the object are translated. The byte-code corresponding to the implementation of the object methods is placed at the end of the buffer, and the pointer to the its method table is set as an offset from this new position. The part of the frame that holds the arguments has to be translated and inserted in the local exports table.

Once packed and translated, the message is placed in the proxy's outgoing queue. The structure `Box` is enqueued in a linked list of outgoing messages owned by the local proxy. The access to the list is mutually exclusive.

5.5.4 Local Proxy to Remote Proxy

In DiTyCO the proxy manages a set of threads to perform intra, and inter-node communication. There are multiple threads to accept external messages, to read its incoming queue and to send each message placed in the queue.

Migration of resources involves two steps in case of communication between sites belonging to the same node; and, three steps in case of communication between sites lying in different nodes. In this section we describe the general case of a three step migration, which includes the phase in which two proxy's lying in different nodes communicate.

The next list describes the actions involved in the process starting from reading a message placed in the proxy's incoming queue, up to the reception of this message in the remote proxy.

1. Read and process the queue of incoming messages:

All messages placed in this queue are read and processed in turn. The proxy uses a single thread for this job.

2. Pass each message to an external communications thread:

Using a round-robin algorithm each thread is assigned a message which is to be transmitted to the remote proxy until there are no more messages to be sent.

3. Fragment the message in packets:

Each communication thread checks the size of the message to send. If the sum of the size of all fields in the Box, plus the buffer with the frame (and eventually the byte code, for the case of objects) is greater than a pre-defined threshold, then this message must be fragmented into several packets. The strategy is to fill up the packed buffer and to associate with each packet a flag (signaling a continuation packet or not). Note that inter-node communication in DiTyCO is accomplished using TCP and therefore we have reliable, in order, packet switching.

4. Transmit each packet to the other proxy:

Make connection with the target node, through its proxy, and send all the packets that correspond to the message to be transmitted.

In the case of the `AppletServer` example, the instruction `trmsg f0`, at site `Client` (cf. page 116), will trigger resource mobility by sending the message to site `Server`. As the message migration process is asynchronous, the DiTyCO virtual machine continues its execution, preparing another message: `msgf 0,"val"`, which will try to reduce in the next instruction using `trmsg r0`. This last reduction can only happen when there is an object in the local channel pointed by `r0`. Such an object is only placed after a reduction at the site `server` and is triggered by the reception of the `Server's` remote message.

For the moment let us now focus on the process that takes place in site **Server** just prior to the reception of the client's message. The situation is as follows: there is a channel named `appletserver` in which it is placed an object containing the applets available for downloading.

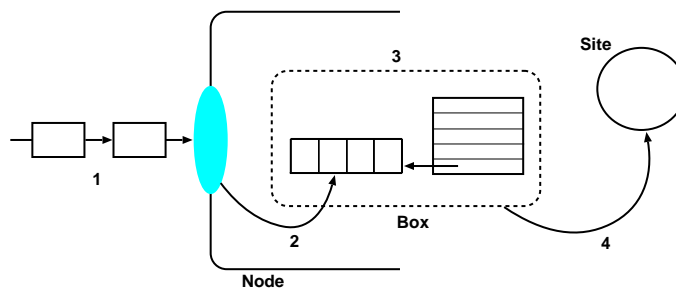


Figure 5.9: Message reception, reassembling and enqueueing at site **Server**.

With the help of figure 5.9 we describe the process that takes place from the reception of all packets sent by the client, up to the enqueueing of the message at site `Server`:

1. the packets arrive at the node where the `Server` is and are received by the local proxy;
2. this proxy reassembles the original message (in the case that the size exceeds the maximum of the DiTyCO common messages protocol). The reassembling of the message involves transposing data from the inter-node data structure to the `Box` structure, used for intra-node communication;
3. the heap frame packed and translated that came inside the received message is copied into a shared memory buffer which is linked to the newly created `Box`;
4. the `Box` data structure is placed in the respective site's incoming queue.

This completes the involvement of the proxy at the target node. From this point on, the process continues exclusively at the target site (in this case, site **Server**).

5.5.5 At the Server Site

Placement of messages in the site's incoming queue does not affect the internal execution of DiTyCO tasks. Every task placed in the runqueue is executed, in normal order, despite the pending message. When one of those tasks ends, that is, when a return instruction (`ret`) is reached, then the incoming queue is inspected for newly arrived messages. Thus, in this case, and after the return instruction, the incoming messages queue is read to process the message. Note that messages, objects and instantiations are processed in slightly different ways.

Now, re-translation takes place, and references lexically bound to the current site are converted from the network format to the local format; *offsets* are converted into pointers, and; the frame is placed in the local heap (in the case of an object, the byte-code is placed in the *remote code* area). Note in figure 5.10 the message argument was retranslated from a network address (`Idx:Loc:IP`) into an exports table index (`Idx`).

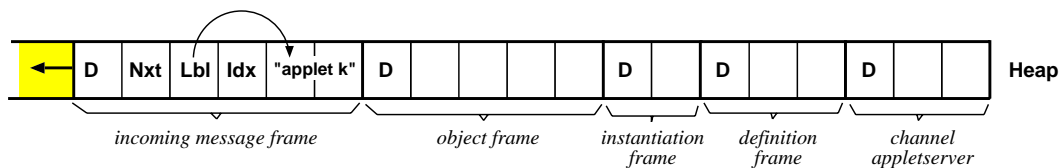


Figure 5.10: Message unpacked and re-translated at site **Server**.

Depending on the state of the target channel where the resource is directed to, two things may occur: either the message reduces with an object or is enqueued in the channel. In our example, communication will take place in channel `appletserver`, and a new task will be placed in the run-queue.

This communication occurred between the message sent by the client, and the object already placed in the channel `appletserver`. The reduction will trigger invocation of the method `appletk` as stated in the program code. The result of this reduction is the execution of the byte-code for the method:

$$\text{applet}_k(p) = p?(x) = P_k \mid \text{AppletServer}[\text{self}]$$

Which corresponds to the assembly code represented in figure 5.11.

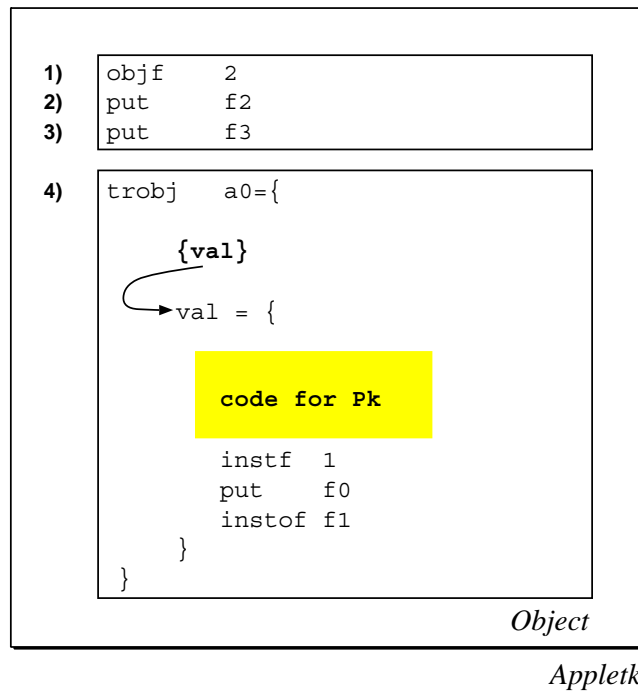


Figure 5.11: Assembly code for the Applet Server example.

This code starts by creating an object frame which will try to reduce at the channel used as an argument to the method invocation `trobj a0`. The argument is a reference to a remote channel, which is the `y` channel created at site **Client**, sent in the message, translated to a network reference. Therefore, reduction at this channel triggers the deployment of the object back to the client's site.

Hence, the execution of the `trobj` at the **Server** makes the object be packed and translated before being sent to channel `y` at site **Client**. This translation involves the use of the **Server** local exports table. The byte-code is placed as it is in the packet. Note that, changes or conversions never occur in the byte-code, but only inside the heap frames that are placed in packets to be sent to remote sites.

To finish the method invocation, the definition `AppletServer` is re-instantiated in the same channel (passed as an argument – `self`), and the object stays in channel `applet-server` waiting for another request for an applet.

5.5.6 Back at the Client

Meanwhile at the client, the program continues executing (recall that, method invocation is asynchronous in DiTyCO). The following code describes the steps taken in the remainder of the program before the applet arrives.

```

main = {
    ...
    msgf      0, "val"
    trmsg     r0          // y
}

```

A message frame is created: `msgf 0, "val"` which tries to reduce at the local channel `y`, represented in the assembly code by `r0` (note that frames representing messages with the method "val" do not use space for the label name). The message is enqueued in the channel until an object (holding the applet) arrives for reduction.

The reception by the proxy node of site **Client** is similar to the one described for the reception of a packet at site **Server**. Then, the (assembled) packet is placed at the incoming queue at site **Client**. Upon reading this incoming queue, the DTvm processes the object in a slightly different manner from the message case. The object frame is, equally placed in the heap, however, with an object, there is byte-code which must be placed in a special area in order to be executed as illustrated in figure 5.12.

Once these operations are completed, the object may reduce with the enqueued message.

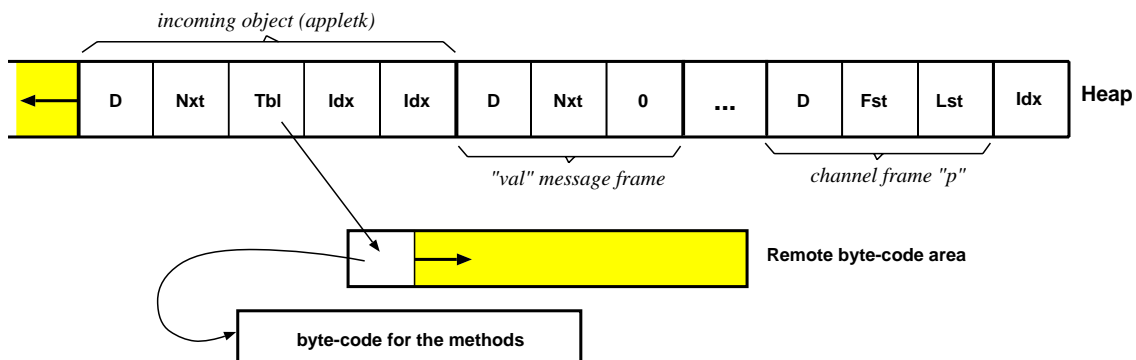


Figure 5.12: Heap at site **Client** after receiving the applet.

The memory area dedicated to external code is created with a specific size which may change according to the memory needs for incoming code (cf. figure 5.12). The external code area uses basic memory management, which recovers space when the global garbage collector is run in order to reclaim heap space. Objects received from other sites are placed in the local heap, but the correspondent byte-code is placed separately from the local program byte-code. Therefore, when those “remote” objects disappear from the local heap during a garbage collection, the corresponding byte-code is also freed.

To finish our example, the object reduces locally at channel y with the local message invoking the desired applet, which now is executed locally at the client.

This completes the description of DiTyCO weak mobility. In the next sections we present DiTyCO’s strong mobility and how it is implemented.

5.6 Strong Mobility

The term *strong mobility* denotes the movement of a running computation (site in DiTyCO) from one place to another in a network (node in DiTyCO). Using *weak mobility* to model *strong mobility* constrains the programmer to explicitly manage the execution state, degrading program readability and ease of debugging. Fuggeta, Picco and Vigna [34] derived formulas that take into account the size of the transferred code and showed that implementing the mobile agent paradigm using weak mobility lead to a bigger code, thus reducing the benefits potentially achievable.

In DiTyCO, strong mobility 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 construct **go**. This construct is not present in the base calculus and was added as an experimental feature to the language and run-time system. Its syntax is as follows:

```
go [machinename | IP address]
```

This instruction triggers the movement of the current site to another IP node. Upon arrival the execution is resumed. Remember that the site names are unique in the

whole network, therefore we cannot have two sites with the same name even in different nodes.

5.7 Implementing Strong Mobility

To introduce strong mobility in DiTyCO we had to take some extra implementation decisions. The most relevant ones are summarised below and will be discussed in more detail in the following sections.

1. All memory blocks must be connected in some way. That is, there must be no single block of memory in use that cannot be reached through the heap, run-queue or from the registers;
2. all memory pointers must be converted into offsets (because memory references are not portable). The main difficulty with this new representation is, to distinguish between primitive values and pointers (expressed as offsets);
3. all messages placed in the site's incoming queue must be processed before the site moves;
4. a mechanism to route resources to the "moving target" needs to be defined and implemented;
5. a protocol for site migration must be derived in order to cope with a site-sized packet transference.

5.7.1 Packing a site for migration

All DiTyCO data structures used in a site must be serialized, that is, sent through the socket connection to the other proxy lying at the target node. Recalling the structure of a DiTyCO site, illustrated in figure 5.13, we describe the data-structures that are packed and the procedure used to do it.

The heap and run-queue share the same array of machine words (32 bits) which grow in opposite directions. This array is packed as it is. The only transformation

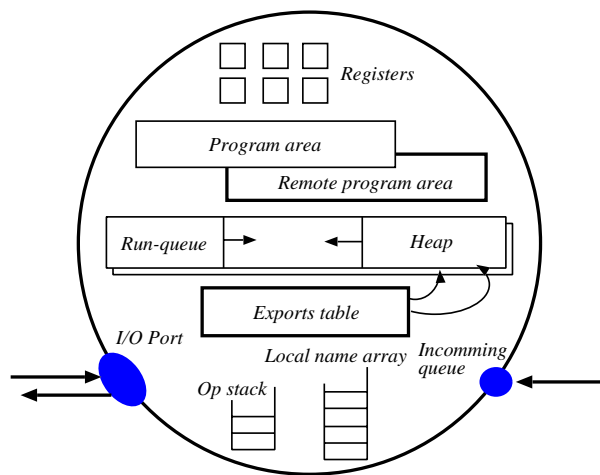


Figure 5.13: The DiTyCO Virtual Machine.

to be done is the translation of memory pointers into offsets for the heap and for the run-queue.

The local name array must be translated into heap offsets.

The local program byte code is packed as it is, because all program jumps are based on code offsets.

The machine registers are packed in an array following the order specified below and performing translation operations to convert memory pointers into offsets.

The operand stack also requires translation of pointers into offsets.

The exports table only has IP addresses, site ordering numbers, and offsets to heap positions, therefore it is packed as it is.

The remote byte-code is packed by keeping an index structure of code blocks followed by the code blocks placed sequentially. The indices are now offsets also.

All the above data structures are packed into a single big buffer holding the state information for the site so that, after migration, the buffer can be unpacked and the site restored in another DiTyCO node. This is possible because we have access to the full state of the DTvm at run-time.

5.7.2 The Site Migration Mechanism

Site migration is a time-consuming operation. It involves packing all data-structures concerning the execution of a site into a buffer and translating every memory pointer into an offset within that buffer area. These operations are performed by a single thread external to the site. Translation of memory into offsets can be entirely avoided if one adopts the offsets representation by default and adjust the DTvm to use them instead. We have taken this approach and are adjusting the implementation accordingly.

Typically a site is run with a (configurable) heap of 256K words, including the run-queue area. The registers and local name array do not occupy more than 1K word altogether. The operand stack is 32K words and the local exports table initial size is 8K words. As the program byte-code, typically, does not add much more memory, we have, around 300K words as the total memory space to be transferred. The general packets used for resource mobility are adequate for the transference of messages, objects and definition arguments, however, not for a transference of this size. Thus, we use a different packet for this particular operation. Moreover, the communication protocol is slightly different.

The data structure used for packing and migrating a site is described below:

```
typedef struct {
    int code;                // code for packet
    char sitename[MAXNAME]; // size of data
    long size;              // size of data
    int data[SITESIZE];     // buffer for packed site

    int port_address;       // socket parameters for I/O
    struct in_addr ip;

    int bc_size;            // program size
    int remote_bc_size;    // remote byte-code size
    int stack_size;        // operand stack size
    int local_exports_size; // local exports table size
}SitePacket;
```

In DiTyCO we have two threads to deal with site migration in order to allow the concurrent delivery and reception of sites. One thread is used to receive sites. The other thread is used to send sites to another node. Whenever a site reaches a **go** instruction it places a “request to migrate” in a proxy queue shared by the sending thread and all sites in the node. Access to the queue is mutually exclusive.

These threads communicate on a special (and well known) port which is different from the one used for transference of resources. This separation between ports for weak migration and ports for strong migration makes it possible to have a different communication protocol according to the type of migration.

When transferring an entire site from one node to another, every memory area is placed in a buffer in a defined sequence. The sequence starts with the heap, then the local program area (**Byte-Code**), the remote code area (**RemoteBC**) and last the exports table (**Exp Tbl**). This ensures that the pointers from the heap, or from the run-queue, to these memory areas are also transformed into offsets as illustrated in figure 5.14.

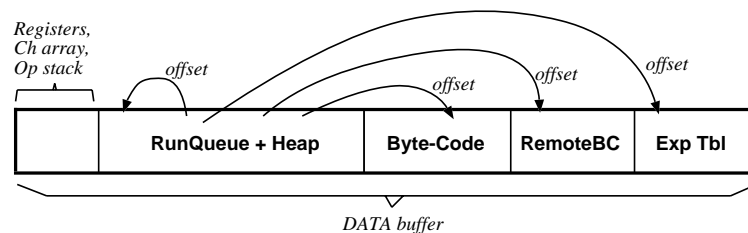


Figure 5.14: Different memory areas in a packed site.

5.7.3 Resource Delivery

At the network level, strong mobility requires the use of the NNS to keep track of sites as they move through the network and to redirect any mobile resources targeted to these moving sites.

The main difficulty here is how to deliver packets sent to sites that are no longer located at their original node. This issue can be expressed in more detail by presenting the problematic situations:

- a proxy receives a packet for a site that no longer runs in the node;
- delivering a resource to a site that is packing up or moving;
- re-routing the resource to the new site location.

These situations have to be confronted with the readiness of the site. The site may be available to receive resources; or it may be packing for migration, or even may

have moved to another node. In the last two cases, the site is inaccessible. It is the responsibility of the proxy to deliver resources to sites in the node. Thus, to assist the proxy in taking appropriate actions for the above situations, we assign different states to a site during its lifetime within the node.

The possible states for a site are:

- *Available*: an executing site that is ready to accept incoming resources;
- *Migrating*: a site that may be packing or already moving to another node;
- *Unknown*: a site that has been confirmed to be executing in another node and its new location is already known by the NNS.

Figure 5.15 illustrates the transition diagram for the site state within a node. A site changes its state from *available* to *migrating* whenever it executes a `go` instruction. We call this a **prepare-to-migrate** action. The node's proxy updates the site state from *migrating* into *unknown* upon the double confirmation that the migrating site has resumed execution in another node and its new location is known by the network. We call this a **close-local-site** action.



Figure 5.15: Transition diagram for the site state within a node.

The two actions **prepare-to-migrate** and **close-local-site** encompass the following steps:

prepare-to-migrate:

1. lock the *site-state*;
2. read all resources from the incoming-queue;
3. change *site-state* into *migrating*;
4. unlock the *site-state*.

close-local-site:

1. send message to NNS requesting update of site location;
2. wait acknowledgement from the NNS;
3. change *site-state* into *unknown*;
4. return resources in the waiting-room to senders with an error message

The **prepare-to-migrate** action requires mutual exclusion because the access to the incoming-queue must be synchronized between the site and the proxy. The **close-local-site** is triggered by the acknowledgement received from the proxy of the node to where the site migrated. Its execution does not require any further synchronization.

Having the above described state information for a site allows a proxy to take appropriate decisions whenever it receives incoming resources for a site. These decisions are described by the **proxy-incoming-messages** procedure:

proxy-incoming-messages:

1. lock the *site-state*;
2. if *site-state* is *available*
 - 2.1 deliver resource to site;
3. unlock the *site-state*;
4. if *site-state* is *migrating*
 - 4.1 enqueue arrived resource in the *waiting-room* of that site;
5. if *site-state* is *unknown*
 - 5.1 return resource to sender with an error message;

Note that synchronization is only necessary when the resource can be delivered to the site, case in which the proxy must place it in the site's incoming-queue.

The *waiting room* is implemented as a queue of assembled packets in shared memory. When the state of the site changes to *unknown* this queue is consumed by the proxy to return the resources and corresponding *site not found* error messages to each of the senders.

We now illustrate the steps taken in a situation of routing a resource to a migrating site and the vital importance of the NNS to keep track of sites and to provide resource mobility under a dynamically changing network topology. In this example we have a site **A** which migrates from node **Y** to node **Z**. During this process site **B** sends a resource to **A**. In figure 5.16 we present how the DiTyCO resource delivery mechanism works.

Step 1 illustrates the moment at which site **A** moves to another node; in step 2 the proxy updates the NNS information for the new location of the site. Suppose now that another site **B** tries to move a resource to the old location (step 3). This shows

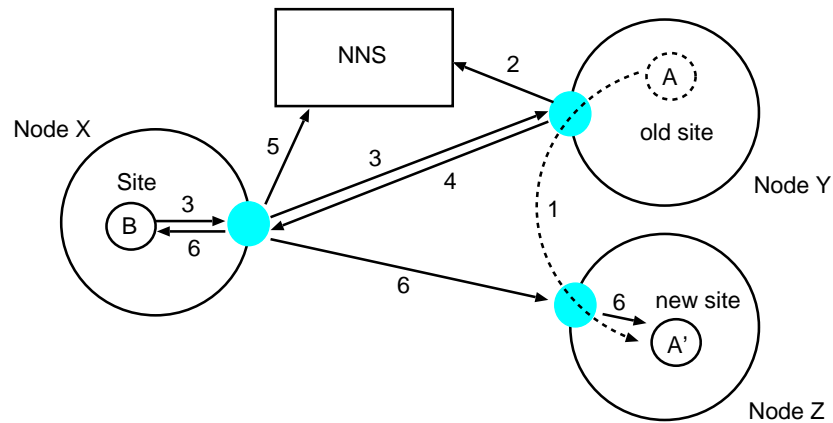


Figure 5.16: Resource Delivery Mechanism.

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 (step 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 (step 5) and resends the message to the new location (step 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 updated dynamically in the network wide service provided by the NNS.

This scheme is based on the need to solve two race conditions: the first one occurs when site B queries the NNS for the new location of A but, it has not yet been updated. The second situation, is to prevent the loss of a resource being placed in a site's incoming queue, when: it is packing; it is moving to the another node, or; it has not yet restarted execution.

Thus, in this example, steps 1 and 2 are unfolded the following way:

1. site A communicates to the local proxy the intention to move to node Z, updates the site status to *migrating* and starts packing itself;
2. site A migrates to node Z and is unpacked there;
3. site A acknowledges the proxy at Y the migration success;
4. the proxy at Y updates the NNS regarding the new location for site A;

example. It shows the migration of site **A**, the attempts of site **B** to send resources to site **A** while it is migrating, and finally the successful rerouting of these resources to site **A** once it is running again in the node.

5.8 Chapter Overview

In this chapter we described the support for mobility in DiTyCO. First, we focussed on the weak mobility, present in the base calculus, which allows migration of messages, objects and instances induced by the lexical scope. We, then, described in detail the support of strong mobility in DiTyCO and how it is accomplished by adding a special `go` construct that explicitly marks the point in the code that triggers the migration of the executing site. We also discussed issues that had to be taken into account to implement strong mobility and derived a protocol for tracking sites and resource deliverability as they move through the network. The strong mobility feature is currently being used in the implementation of a higher-level language, MOB, (encoded in DiTyCO) that is being implemented as part of an ongoing team research work [69].

Chapter 6

Related Work

In this chapter we present a comparative analysis of various models and languages that possess the notion of distribution and some sort of code mobility. The goal is to relate these other proposals with DiTyCO as a way to assess some decisions taken during the system creation, like the existence of a base calculus, which kind of mobility, and how migration is processed, abstractions of the language and main entities. We believe that finding similarities in our system with other independent approaches gives us a higher confidence in our work.

6.1 Basic Issues

For the creation of a programming language whose implementation is *provably* correct, one has to construct it from a solid semantic specification, in such a way that, the resulting implementation is guaranteed to satisfy the correctness requirements. Usually this is done by repeatedly applying correctness-preserving transformations to the operational semantics, until the resulting specifications have the form of abstract machines [43].

Turner [85] presented an abstract machine to implement the π -calculus which was successively refined until a C-based machine was reached. In a similar way, Lopes [56] starting from the TyCO process calculus obtained several abstract machines, providing for each one, the translation to the previous one, and proving soundness of that translation. Other similar studies have been made by Sewell [78] to relate the behavior

of the semantics of Pict and that of its C-program implementation.

This process confronts language designers (theorists) and implementors. Traditionally, implementations are built “by hand” and, while very seldom proved correct, they satisfy well defined needs for very specific contexts. These systems are very usable and ready to be programmed as they, usually, include many functionalities through a set of instructions to address real-world situations. Moreover, a survey of those (usually commercial) systems provides insight for the difficulties inherent to implementation of distribution and code mobility.

Java [57] has been playing a leading role in the field of programming languages used to program distributed applications, that exploit some form of mobility. Java provides libraries with such useful abstractions as: *sockets*, *downloadable classes*, *multi-threading* and *synchronization* as primitive notions to language. Therefore, concurrent and distributed programs can be effectively programmed in Java [48]. Many mobile agent programming environments have been built on top of Java: Mole [81], Concordia [95], MOA [66] and Voyager [36], to name a few.

However, many systems, which consist of Java libraries supporting the programming of agent migration have some limitations due to the library itself: in Java, presently, it is only possible to “serialize” restricted types of objects, in particular it is not possible to serialize threads, and therefore, it is not possible to capture the state of a thread at a given moment of its execution. Hence, these systems based on Java do not allow the implementation of the strong mobility in its full generality. Although it has this limitation, Java is arguably the most useful programming language for distributed applications exploiting forms of weak mobility.

6.2 Survey on the State of the Art

This section concentrates on models and languages that possess code mobility. They all have support for some sort of weak and/or strong mobility, and apart from the commercial systems, all of them are based in some form of the asynchronous π -calculus.

We survey other process calculi and programming languages that, like DiTyCO, have support for concurrency, distribution and code mobility. Our methodology to survey these systems involves the analysis of the following five main issues:

- is there a base calculus supporting the system?
- how does it deal with the notion of distribution?
- does it have the notion of mobility? Of which type and entities?
- which are the fundamental abstractions of the system/language?
- is there an implementation of the system?

We relate each system with DiTyCO, presenting their main differences and similarities. This analysis is not exhaustive and many proposals are left out, as we only consider systems that do possess the notions of distribution and/or of code mobility.

6.2.1 The $D\pi$ calculus

The Distributed π -calculus [44] is designed to be “a minor extension” of the π -calculus that introduces the notions of distribution and mobility of processes. In this calculus, locations form a physically flat space. The only communication allowed is local. In $D\pi$ -calculus, all processes execute at a specific named location but can move between locations.

For a reduction to take place it is necessary that the respective redexes be at the same location, otherwise it is necessary that, first a process migrate to the site and then reduce locally. Non-local channels are transmitted using *structured values* which are of the form:

$$a@s$$

meaning a channel named a , located at site s (just like the network notation in DiTyCO). Migration in $D\pi$ is expressed using the primitive:

$$\text{goto } s.Q$$

where Q is a process and s is the target site. Process Q may only start executing after reaching site s . Thus, migration is a blocking operation. This system is probably one of the most similar to DiTyCO.

6.2.2 The π_{1l} -calculus

The π_{1l} -calculus [8, 9] is a form of the asynchronous π -calculus that takes into account the possibility of failures of nodes in a network, together with the notion of code mobility. The calculus has a distinctive feature from the π -calculus, which ensures (statically, using a the type system) the unicity of receivers, in the sense that channels transmitted in messages cannot thereafter be used to create receivers.

The network topology is flat in the sense that there is no nesting of localities. There are only two layers: the process layer, and the locality layer (this structure of two layers is also present in the DiTyCO calculus). A locality may be in one of two states: either *running* or *stopped*.

When it is in a stopped state, it cannot emit messages, nor accept migrating processes. However, it can still receive messages, though it has no observable consequence. A locality can be stopped, but, after being stopped there is no means to restart it (as it cannot receive processes).

The migrating entities in this calculus are processes. In a migration, the environment is taken into account: the process migrates together with the processes that have been substituted for the recursion variables that occurred in itself. Thus, the state of the process is also moved.

Migration is expressed by relocating a running process to another locality, using the instruction:

$$\text{spawn}(s, P)$$

The semantics of this instruction is: move process P to locality s , provided, s is a running locality. During migration, the process P cannot emit or receive messages. A mechanism for tracking moving receivers is mentioned in [9].

6.2.3 The Distributed Join-calculus

This calculus [33, 50] is a distributed form of the Join-calculus (also known as DJoin). It includes features to deal with partial failure and is based on the reflexive chemical abstract machine [32].

The basic entities are messages, definitions, def-processes (processes containing a definition), patterns and solutions (multisets of def-processes).

Localities in DJoin are organized in a tree structure which is constructed as programs are being run. Local communication is asynchronous and channels and locality names may be transmitted in messages. Several processes may receive on the same name, provided they share the same locality. Remote communication is transparent in the sense that names are globally unique, thus there is no difference between local and remote communication.

When a locality migrates, all its sub-localities also migrate. The tree structure of localities in DJoin is created as follows: a migrating locality becomes a direct sub-locality of the locality designed to be the target of the migration.

A locality can either be “live” or “dead”. The failure of a locality entails the failure of all its sub-localities. A failed locality cannot migrate nor emit any message. However, messages, and even other localities, may migrate into a failed locality. Failure of localities may be detected from any other live locality.

Migration is expressed by the instruction:

$$\text{go}(s, P)$$

Where s is the target locality and P is the continuation. The `go` instruction is a subjective form of migration and it is an asynchronous operation like in DiTyCO. Migrating localities maintain their links with the context, however, as they are just names, the binding process is completely transparent.

The *Join Calculus Language* (JCL) and the *Join Objective-CAML* (JoCaml) are the two main implementations of the Join-calculus with localities (DJoin) [31, 50]. Both support weak and strong mobility, however, JCL was designed from the start with distribution in mind, and JoCaml [23] is an extension of the OCaml (Objective Caml) programming language, with primitives for distributed programming.

A location may represent the notion of a physical site (in the case of an empty location), of mobile code (when it contains channel definitions), or a mobile agent (when it contains threads).

Communication is transparent for the user/programmer as there is no distinction in the code, between local and remote channels. Channels are first class entities, which

can be sent as message arguments and extend their scope over the network.

JCL and JoCaml support three types of code mobility: i) remote evaluation; ii) code on demand; iii) mobile agent. The last one is achieved through strong and subjective mobility, that is, the decision to migrate comes from a program instruction - `go` - as in DiTyCO. To deal with migration, message forwarders are left in the initial site to preserve communication links.

A synchronous channel is created like this:

```
let def name(args) = P(args)
```

Through the above expression, a channel `name` is created and `P` is executed whenever a message is received. Agents are created through the code:

```
let loc agent do P
```

This command creates an agent named `agent` executing `P`. As channels are unidirectional agents can only send messages.

The JCL abstract machine interprets byte-code and is fairly similar to DiTyCO with a heap for storing definitions, locations and environments, and also sharing the space with a run-queue. It is register based and uses a stack containing the stack of a thread.

Threads are active computations that occur in locations. A scheduler ensures time-sharing between runnable threads in the abstract machine run-queue. Threads may be in one of three possible states: i) running; ii) runnable; iii) blocked.

Channels act as mailboxes where asynchronous messages meet. Local communication is achieved by function calls and remote communication by TCP/IP and Unix sockets.

JCL does not provide a mechanism for garbage collection, whereas JoCaml has it allowing to collect cycles between different run-times. The naming service is centralized in JCL and possibly distributed in JoCaml. The naming service is responsible for the creation of fresh names in physical sites and for a mechanism to share the names between different sites. It also allows registration of names, such that any object of the language can be referred to.

6.2.4 The Seal-calculus

The Seal-calculus [91, 92] was conceived to incorporate the fundamental properties of Internet programs. Its main design principles are the absence of global state, explicit localities, restricted connectivity, dynamic reconfiguration and access control.

The calculus entities are processes (sequence of communication and migration capacities) and resources (memory, peripherals, and even services). However, the only resources explicitly manipulated by the calculus are channels and domains.

A *seal* is a named locality, containing a number of running processes. There may be several seals with the same name. Seals are structured hierarchically, and the topology may change as a consequence of the mobility of seals.

During communication it is possible to have multiple receivers on the same channel name. Local naming is explicit in the calculus through the use of the \star symbol. Local communication (between two processes sharing the same seal) is synchronous.

Remote communication is only possible between immediately related seals, that is, between parent and child seals. For remote communication to take place, the calculus requires a further authorization called a “portal”, which is a linear access permission to a specific channel. The owner of the channel may grant this permission upon request, to its parent or children seals. As in local communication, remote communication is also synchronous.

In Seal, migration is expressed as communication by sending a seal name on a channel. This, has the effect of moving the seal to the location where the communication is to take place. The migration is objective, as it is triggered by an instruction outside the migrating seal. It is also asynchronous because the migrating seal executes independently from the migration order.

The way of expressing communication between a seal and a parent seal, is to decorate the channel (located at either of the two seals) name at the parent with the symbol \uparrow :

$$n[\bar{x}^\uparrow(\vec{z}).P] \mid x^\star(\vec{y}).Q \rightarrow n[P] \mid Q\{\vec{z}/\vec{y}\}$$

The calculus implements a set of protection mechanisms that allow different security policies: it has control of names by the use of lexical scoping; it controls the use of resources through the “portals”; it has control of migration and communication

through the hierarchical model of the calculus (it is only possible to step up or down one level at a time, and for remote communication, permissions granted by “portals” are required).

The Seal-calculus satisfies the “perfect-firewall” property, $(\nu x)x[P] \simeq 0$, because if a seal’s name is not known from any other seal, there cannot be any portal open for that name, therefore, it cannot exist any communication on a channel in that seal.

6.2.5 X-Klaim and Klava

X-Klaim derives from Klaim (Kernel Language for Agents Interaction and Mobility) [25, 26], a language based on the Linda tuple space model. In Klaim the tuple space is extended into a multiple distributed tuple space. X-Klaim extends Klaim with variable declarations, enriched operations, assignments, conditionals and sequential and iterative process compositions.

Each node on the network corresponds to a single tuple space, which can be accessed through its locality. Processes can execute operations over distributed tuple spaces.

In X-Klaim there is static and dynamic scoping according to the communication primitive used. As with DiTyCO, the name resolution is initially done by a net server to provide a direct connection which is used thereafter without interference from the net server.

The Klaim network topology is flat, despite being able to dynamically evolve by the creation of new nodes (domains). However, some extensions have been proposed where nodes are structured into a hierarchy [15].

The framework for X-Klaim consists of a Java packet (the Klava), which provides the run-time system for X-Klaim and a compiler that translates the X-Klaim programs into Java programs which in turn use the package Klava. Klava only supports weak mobility of agents, however, X-Klaim allows strong mobility in the same line as DiTyCO, by using the primitive `go@s` which makes an agent migrate to site s . Classes needed by a process are automatically collected and delivered together with the process.

In Klaim, processes run concurrently, both at the same node or at different nodes. During program execution operations on the tuple spaces and nodes can be done

through the constructors:

- $\text{in}(t)@s$ which, evaluates tuple t and looks for a matching tuple t' in the tuple space located at s .
- $\text{read}(t)@s$ differs from $\text{in}(t)@s$ because the selected tuple is not removed from the tuple space.
- $\text{out}(t)@s$ adds a tuple resulting from the evaluation of t to the tuple space located at s .
- $\text{eval}(P)@s$ spawns process P for execution at s .
- $\text{newloc}(s)$ creates a new node in the net and binds it to s .

X-Klaim programs are compiled into Java programs by the X-Klaim compiler, which in turn are processed by the Klava package [14] in order to be run by the Java interpreter.

6.2.6 Safe Mobile Ambients

Safe Mobile Ambients (SA) [42, 49] was derived from the Mobile Ambients [20, 21, 37] calculus by removing interferences that were produced when performing different movement operations concurrently.

Ambients may nest forming a hierarchical structure. Processes that share the same ambient can interact with each other by exchanging messages. When they belong to different ambients, communication is only possible if an ambient moves inside another and is *opened*.

Movement of ambients require the handshaking between the migrating ambient and the exited or entered ambient. The three primitives for mobility used in Ambients (in , out and $\overline{\text{open}}$), which are form of subjective migration, require in SA, an objective authorization by handshaking with the primitives $\overline{\text{in}}$, $\overline{\text{out}}$ and open , respectively. The open primitive in SA is no longer an objective operation, as it now requires the agreement of the ambient to be opened.

The safe ambients abstract machine is written in Java. The system's architecture (implementation) is similar to DiTyCO being based on a three layer system. In safe

mobile ambients there is the *agent layer* in which agents communicate between them by exchanging messages. Agents local to a host are gathered in the same *node layer*. The node layer is implemented using the Java RMI and is responsible for inter-node communication. Each node corresponds to a physical machine and may contain several *agents*. The network layer provides a name registry resolution for RMI communications to take place.

In the SA Abstract Machine, an agent starts and ends its life on the same location. When an agent is destroyed (by opening it), its contents migrate to another host.

As in TyCO, safe mobile ambients has a type system which prevents execution errors.

6.2.7 Nomadic Pict

The Nomadic Pict [79] system extends the Pict language [71] (which is based on the π -calculus, and does not support the concepts of mobility and distribution) with primitives for migrating agents in a distributed environment. The main abstractions are: channels, processes, agents and sites.

In Nomadic Pict agents need to be at the same site to communicate between them. There is no remote communication in Nomadic Pict – all communication must be local. However, there is a protocol which makes use of migration to implement transparent remote communication. Inside an agent, processes communicate using named channels. Local communication occurs with a blocking input construct, and names (channels, agents and sites) may be passed during the communication.

Nomadic Pict has a global naming space, such that there is a mechanism for creating fresh names in the all network.

The agent is the unit of migration in Nomadic Pict and are created like this:

$$\text{agent } a = P \text{ in } Q$$

The above instruction creates an agent named a with body P , running concurrently with Q in the same site.

In Nomadic Pict (strong) migration occurs by the execution of an instruction of the form:

$$\text{migrate to } s \rightarrow P$$

Which moves the agent executing this instruction to site s . Process P is also added to this agent's code. Thus, it is a subjective kind of migration.

Migration is also an asynchronous operation with respect to processes executing in the migrating agent. The reason for the continuation (P) is to be able to synchronize the behavior of the agent with its movements in the network. Messages and receivers that are part of the agent also move during migration.

6.2.8 Obliq

Obliq [19] is an object-based programming language featuring mobile code. The basic entities in Obliq's implementation are: sites, memory locations, values and threads. The communication is assumed to be reliable and transient failures are not distinct from unreachable nodes.

Obliq relies on the notion of network transparency in the sense that the semantics of a program does not depend on the physical location of its components. A network reference in Obliq is a pair of a *site name* and *memory location* at that site. A *name server* is used to register objects, to provide network references to objects and to access remote objects. References are either local (memory locations), or global (network references) and cannot be manipulated at the programming level. A *site* is considered a place where threads are run. Sites do not nest, hence, the network topology is flat.

Obliq has various forms of migration, though none is formalized. Migration is not explicitly a programming construct in the language, but, there are several types of migration associated with primitive language operations. For example, it is possible to model "code on demand", "code shipping" and creation of "surrogates" to allow for object migration. One of the main features of Obliq is that references are never broken upon migration, because, if their value is immobile (as with objects or memory locations) it is transformed into network references. Memory locations and objects are transformed into network references prior to migration.

In Obliq, an object can be exported to the name server using the command:

```
net_export(obj, NameServer, siteObj)
```

Where `obj` is the registration name for the object, `siteObj` is the object and, `NameServer`

is a string containing the IP address of the machine running the name server.

The dual operation is:

```
let siteObj = net_import(obj, NameServer)
```

After this instruction object operations can be applied to `siteObj` as if it were a local object, and the name server is no longer used for communication. The object can also be made available to a third site by transmitting its reference through a communication channel.

Obliq is one of the first implemented programming languages which supports mobile code. It is object-based as it does not have the notions of class nor of inheritance. In model and philosophy Obliq is close to DiTyCO.

6.2.9 Agent TCL

Agent TCL [39, 40, 47] is an extension of the TCL scripting language to enable distribution and agent migration. Recently, Agent TCL is a part of a larger project known as D'Agents [4, 41].

The concept of domain is represented by the *agent server*, which is a process running in each site, that executes and migrates agents.

Agent TCL includes a number of primitives that allow creation and migration of agents. Those agents are represented as TCL scripts. Both agents, and agent servers have a unique name in the network.

There is no distinction between local and remote communication. Agents have a single mechanism to communicate with other agents. Using such mechanism we may differentiate two kinds of inter-agent communication: *message passing* and *rendezvous*. The former is a blocking communication (with timeout), monadic and point-to-point. The later has semantics similar to message passing, however, it establishes a direct connection and synchronization with the other agent.

Migration in Agent TCL is presented in two ways: as *code shipping* (creation of a new agent in a remote agent server) specifying the agent script code to be executed; and, *strong migration* by executing the continuation of the current agent in another agent server.

As in DiTyCO, strong migration is blocking in the sense that execution of the agent will resume directly at the point where it was interrupted for migration to take place. In Agent TCL it is also possible agent cloning in which, the server will give the remotely created clone a new identity.

In Agent TCL, the migration of agents can be specified using three different commands: `agent_submit` which creates a new agent at a specified agent server; `agent_jump` which triggers migration of the current agent towards a specified agent server; `agent_fork` which creates a copy of the current agent at the specified agent server. Each agent server is identified by the IP address of the underlying site. An agent name is constituted by the agent server running the agent, plus an unique private name. Therefore, this situation mimics the network address form in DiTyCO (which has one more field to hold the heap position). There is no mechanism in the language to keep track of agents, nor forwarding of messages. Messages sent to an agent that has migrated are lost.

In the language there is the notion of “trust”. A server will only accept an incoming migrating agent if it comes from a trusted site. Agent TCL aims to protect network sites against malicious agents and to protect agents against one another. It is the agent server that checks an incoming agent, and authenticates it before passing the agent to the appropriate interpreter. Authentication is based on PGP. The access control policy is contained in access control lists managed by the agent server.

6.2.10 Telescript

Telescript [93] is an object-oriented language with an emphasis on security and strong mobility. It was, probably, the first commercial agent language.

Places (or sites) are locations that are occupied by agents. It is possible to have several agents in the same location. Agents are processes which can move from one place to another by encoding their state and transmitting it to another place where execution is resumed. Telescript has a portable language named *Low Telescript* which constitutes the code that actually is transferred between execution units. Agents move by using the `go` instruction, which implements a subjective migration mechanism. Another instruction - `send`, implements remote cloning.

In the language there is the notion of *ownership* of a resource. In a migration this notion of ownership is used to determine the set of resources that must be carried with the migrating execution unit. Bindings to migrated resources owned by other execution units in the source site are removed.

The Telescript system, developed by General Magic, includes the Telescript language, the Telescript run-time engine which interprets the language, the Telescript protocol set, which deals primarily with the encoding and decoding of agents, and a set of tools to support the development of Telescript applications.

6.3 Chapter Overview

Many systems are based on a process calculus, remarkably on the π -calculus. The notion of distribution constitutes, in many of the described systems, which are based on process calculi, an extension to another prior calculus that did not have that notion. Examples are: Pict as the base for Nomadic Pict; Join evolved into DJoin and DiTyCO, which emerged from TyCO.

Distribution in these systems is characterized according to two different topologies. The network of locations may be presented as flat space (Nomadic Pict, DiTyCO, $D\pi$, π_{II} , Obliq and Klaim), or as a hierarchical structure (DJoin, Seal, Safe Mobile Ambients, Jcl and JoCaml).

When the network is organized in a tree of locations, migration of one of these locations triggers, in some cases, the migration of the whole sub-tree (as in DJoin) and in others, just the single location (Seal and Ambients).

Some systems have a strong emphasis on security concerns (its the case of Ambients and Seal). In these, “membranes” protect computations, communications are protected by the use of permissions granted by “portals” and special communication schemes based in parent-child relationship of nodes of the network tree are devised in order to enforce security. Others like Nomadic Pict, DJoin, DiTyCO and Obliq do not address these issues.

Strong mobility is usually implemented using a subjective form of migration in most existing systems. It is basically implemented as an operation which suspends the

computation, and resumes it upon arrival to another location, which is also the case in DiTyCO.

With respect to the system's architecture most of the implementations have some form of a name server to provide a global name resolution as it is the case of Obliq, Nomadic Pict, X-Klaim and DiTyCO. The two-level architecture is shared by systems like Agent TCL and Nomadic Pict while the DiTyCO and Ambients implementation is based on a three-layer architecture. The use of byte-code, or some form of portable code and state, is also very common in these systems as Jcl, JoCaml, Ambients and DiTyCO.

Chapter 7

Conclusions

7.1 Overview of the Research

This thesis builds on TyCO calculus, its programming language and the run-time system. Our utmost goal was to add distribution and mobility to the programming language, without sacrificing the base calculus, the language and the virtual machine.

The goal was reached. In its current state DiTyCO is a system capable of executing distributed programs with code mobility. The language has evolved minimally, with the addition of just three new constructs. The new DiTyCO virtual machine, maintains the basic architecture and functioning principles of its predecessor. The base calculus was extended, in our perspective, in the most logical direction, and following the nowadays trend on adding distribution and mobility to process calculi.

The base TyCO calculus was revised to include the notion of *site* – a place where computations occur. The TyCO’s movement of resources between channels (belonging to the same site), was substituted by a movement of resources in a *network of sites* obeying a lexical scope. We derived the necessary calculus rules, such that, computations could proceed in different sites, and reduction rules evolved in a way that messages and objects are no longer restricted to a local channel for reduction (although, reduction is always local). Furthermore, definitions can also be imported from the site in which they were created and used locally. These new features gave DiTyCO a sort of code mobility known as *weak mobility*, characterized by the transference, in a distributed system, of pieces of code that can be executed remotely.

The DiTyCO programming language grew from the TyCO programming language by the addition of the pair `export`, `import` which permit the creation of bindings to external resources. Using that created interface, DiTyCO programs can proceed exchanging messages, objects and definitions, and dynamically changing its interface. Nevertheless, the language maintains the TyCO, object-based, concurrent style, with the functional flavor.

The original TyCO virtual machine (Tvm) was modified as some kernel instructions had to be re-implemented and new ones introduced. New data-structures were also added, as the *incoming queue* of arrived messages, the *I/O* port to communicate with the user for input and output, the *remote code area* to host remote byte-code, and the *exports table* to keep track of names in the network. Routines for translation and packing of resources were also included in the DiTyCO virtual machine (DTvm).

The DiTyCO software system was built on a three-layer architecture composed of *sites*, *nodes* and the *network*. The DiTyCO virtual machine (DTvm) is run (as a separate thread) in a *node*, where other sites may also be executing. The node layer maps one-to-one with physical machines connected in a network. There exists only one node per physical IP, but many sites per node. Communication between sites is made through the node's proxy, which we developed as being a multi-threaded communications module. There are special data-structures (named *boxes* for inter-node communication, which exchange data between sites and the proxy through shared memory), and *packets* for communication between different proxies. A process of fragmenting and reassembling messages was created to keep all packets that transmit resources with the same size, hence simplifying the DiTyCO transmission protocol and overcoming the need to know in advance the size of a packet, as imposed by the TCP protocol.

A *network name server* (NNS) was introduced, as a service, to associate remote names to network locations. This service maintains tables of sites and of *names* (channels and definitions). It provides the mechanism to translate simple names into located names. The network of sites is a flat space as sites do not nest, neither are in a hierarchical structure.

DiTyCO programs are compiled, as in TyCO, through a two-level compiler. In the first pass. the compiler generates an intermediate code (a kind of an *assembly*) from the DiTyCO program; in the second pass, it generates the *byte-code*, which is interpreted

by the DTvm.

DiTyCO's weak mobility features allowed the use of the most common programming paradigms in distributed computations: *client-server*, *remote evaluation* and *code-on-demand*. We presented some examples not only to illustrate DiTyCO's programming style in respect to coding programming applications, but also to show that nowadays programming paradigms for distributed applications could easily be coded in our language. The *mobile agent* programming paradigm is quite recent, and despite being difficult to maintain calculus properties and non-interference of environments in a migrating site [72], we decided to add an instruction that allows DiTyCO programs to be coded in this new paradigm.

The `go` instruction introduces *strong mobility* in DiTyCO in the form of an atomic operation that involves suspending execution and packing the whole site, moving it into another node, unpacking the migrated site and resuming its execution thereafter. The devised packing procedure serializes all the site's data structures into a big buffer where memory references become simple offsets. A new protocol for the transference of sites was created in order to cope with the new dimension of inter-node packets, and only for transaction of sites.

To deal with a network of migrating sites we devised an infrastructure to cope with these migrations and routing resources to the migrating sites. The mechanism is based on: the NNS to get new locations, on a *multi-state table* of sites, and on a node's *waiting room* where packets are queued until they can be delivered, or returned with proper error conditions.

With the above mentioned characteristics DiTyCO is a system with builtin notions of concurrency, distribution and profiting of both weak and strong mobility capabilities.

7.2 Summary of Contributions

The main contribution of this thesis is the design and implementation of DiTyCO, a distributed, concurrent and object-based programming language with code mobility. We believe that the major contributions of this work are:

- The DiTyCO programming language.

- The DiTyCO runtime architecture and its implementation:
 - the DTvm (the virtual machine where sites are run);
 - the Node architecture;
 - the Network Name Server;
 - a protocol for interchanging packets of resources in the network.
- An implementation of strong mobility:
 - the support for strong mobility;
 - a protocol for interchanging sites in the network;
 - a solution for race-conditions during site migration;
 - a reliable routing of resources between migrating sites;

7.3 Future Work

Further work on the DiTyCO language will focus on the DiTyCO-calculus, on fine tuning of the DTvm and on the supporting technologies of the run-time system. We describe each of them below:

- create a distributed type-checking mechanism to prevent run-time errors due to types mismatch and also to provide a basic security in interactions between remote sites;
- enhance the calculus to provide semantics for strong mobility. There is recent work done in this area [72];
- fault-tolerance mechanisms and termination detection need to be introduced in the system. For such, a failure semantics must be devised or adapt from the vast volume of work that has been done in this subject;
- security mechanisms, other than the basic type-checking, should also be considered in DiTyCO. Among these are access control, and secure communications with encryption.

We are aware that DiTyCO needs to be extensively tested. Despite, the basic research direction to provide functionality, efficiency is the natural dual property that must also be maintained. Therefore, we want to:

- evolve the NNS into a hierarchical distributed name server, in the same line as DNS, to avoid the bottleneck of a single machine in the network;
- even though the DiTyCO architecture and the proxy in particular were developed with efficiency concerns, we feel that it is important to assess the system's performance according to site execution when compared to non-distributed execution; also assess the efficiency of resource transference and the slice of the whole execution time during site migration;
- verify if DiTyCO scales well to support a large number of exported channels, executing and migrating sites, hosted by a significant number of nodes. And, ultimately, to assess the behavior of the system in *real-world* applications;

Finally, the implementation of a higher-level agent-based programming language based on DiTyCO, with constructs for abstract data-types and common communication and migration patterns is already being developed within another project.

7.4 Final Remarks

This work began with the idea of creating a language and run-time system, based on a process calculus, such that programs could be checked according to the calculus rules, and proved to be correct. The language part (except for the `go` instruction) is done. The run-time system needs further checking to prove that the semantics of DiTyCO are the same of the current implementation and, therefore, to prove that the DTvm is indeed correct. Despite these concerns, we are aware that a dissertation is never finished, instead, it is abandoned...

Bibliography

- [1] Java(TM) 2 Platform, Standard Edition, v1.2.2 API Specification.
- [2] The COM Specification.
<http://www.microsoft.com/oledev/olecom/title.htm>, 1995.
- [3] The Common Object Request Broker: Architecture and Specification, Revision 2.0. <http://www.omg.org/corba/corbiiop.htm>, July 1995.
- [4] The D'Agents Project. Center for Mobile Computing, Dartmouth College, 2002.
<http://agent.cs.dartmouth.edu/>.
- [5] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [6] G. Abowd, R. Allen, and D. Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, December 1993.
- [7] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of SIGPLAN'96 Conference on Programming Language Design and Implementation*, 1996.
- [8] R. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Proceedings of the 2nd International Conference on Coordination Languages and Models (COORDINATION'97)*, volume 1282 of *LNCS*, pages 374–391. Springer-Verlag, 1997. Extended version appeared in INRIA Research Report 3109.
- [9] R. Amadio. On Modeling Mobility. *Theoretical Computer Science*, 240:147–176, 2000.

- [10] R. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Journal of the Theoretical Computer Science*, 195(2):291–324, March 1998. Also appeared in *Proceedings of CONCUR'96 (August, 1996)*, Springer LNCS 1119.
- [11] Y. Artsy, H. Chang, and R. Finkel. Interprocess Communication in Charlotte. *IEEE Software*, 4(1):22–28, January 1987.
- [12] M. Baldi and G. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In R. Kemmerer, editor, *Proceedings of the 20th International Conference on Software Engineering*, pages 146–155. IEEE Computer Society Press, 1998.
- [13] A. Barak, S. Gunday, and R. Wheeler. The MOSIX distributed operating system: Load balancing for the UNIX. volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
- [14] L. Bettini, R. de Nicola, and R. Pugliese. KLAVA: A Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, John Wiley & Sons 2002.
- [15] L. Bettini, M. Loreti, and R. Pugliese. An Infrastructure Language for Open Nets. In *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC'02), Special Track on Coordination Models, Languages and Applications*, pages 373–377. ACM Press, 2002.
- [16] A. Birrel, G. Nelson, S. Owicki, and E. Wobber. Network objects. *Software-Practice and Experience*, 24(S4):87–130, December 1995.
- [17] M. Boreale, R. De Nicola, M. Lacoste, and V. Vaconcelos. Analysis of distribution structures: state of the art. MIKADO Global Computing Project, RR/WP3/1, July 2002. Deliverable D3.1.1.
- [18] G. Boudol. Asynchrony and the π -calculus (note). Technical Report 1702, INRIA Sophia-Antipolis, May 1992.
- [19] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995. A preliminary version appeared in Proceedings of the 22nd ACM Symposium on Principles of Programming Languages, San Francisco, California.

- [20] L. Cardelli and A. Gordon. Mobile Ambients. In M. Nivat, editor, *Proceedings of Foundation of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.
- [21] L. Cardelli and A. Gordon. Types for mobile ambients. In *26th Annual Symposium on Principles of Programming Languages (POPL)*, pages 79–92, San Antonio, TX, 1999. ACM.
- [22] A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In R. Taylor, editor, *Proceedings of the 19th Conference on Software Engineering (ICSE'97)*, pages 22–32. ACM Press, 1997.
- [23] S. Conchon and F. Le Fessant. JoCaml: Mobile Agents for Objective-Caml. In *Symposium on Agent Systems and Applications, Mobile Agents (ASA/MA'99)*, pages 22–29. IEEE Computer Society, 1999.
- [24] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, Reading, MA, USA, second edition, 1994.
- [25] R. de Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. In Catalin Roman and Ghezzi, editors, *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing)*, volume 24, pages 315–330, 1998.
- [26] R. de Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240:215–254, Elsevier Science, 2000.
- [27] A. Chien et al. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. March 1997.
- [28] N. Boden et al. Myrinet: A Gigabit per second Local Area Network. *IEEE-Micro*, 15(1):29–36, February 1995.
- [29] A. Figueira, L. Lopes, F. Silva, and V. Vasconcelos. DiTyCO: Concorrência, Distribuição e Mobilidade de Código. In *Actas do Encontro Português de Computação Móvel (EPCM'99)*, pages 25–35, Tomar, Portugal, 1999.
- [30] A. Figueira, H. Paulino, L. Lopes, and F. Silva. Distributed Typed Concurrent Objects: a Programming Language for Distributed Computations with Mobile

- Resources. *J.UCS*, 9(8):745–760, 2003. Also appeared in Proceedings of the SBLP’03.
- [31] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Palaiseau, November 1998.
- [32] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join Calculus. In *Conference record of the 23rd ACM Symposium on Principles of Programming Languages (POPL’96)*, pages 372–385, January 1996.
- [33] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A Calculus of Mobile Agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR’96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, August 1996.
- [34] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transaction on Software Engineering*, 24(5):342–361, May 1998.
- [35] S. Garland and N. Lynch. The ioa language and toolset: Support for designing, analysing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1998.
- [36] G. Glass. Object Space Voyager Core Package Technical Overview, 1999. White Paper.
- [37] A. Gordon and L. Cardelli. Equational properties of mobile ambients. In W. Thomas, editor, *Proceedings of Foundation of Software Science and Computation Structures (FoSSaCS)*, volume 1578 of *LNCS*, pages 212–226, Berlin, April 1999. Springer-Verlag.
- [38] J. Gosling, B Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.
- [39] R. Gray. Agent Tcl: A Flexible and Secure Mobile Agent System. In M. Diekhans and M. Roseman, editors, *Proceedings Fourth Annual Tcl/Tk Workshop (TCL’96)*, pages 9–23, 1996.
- [40] R. Gray. *Agent Tcl: A Flexible and Secure Mobile Agent System*. PhD thesis, Dartmouth College, 1997.

- [41] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a Multiple Language Mobile Agent System. In *Proceedings Mobile Agents and Security Workshop*, volume 1419 of *LNCS*, 1998.
- [42] X. Guan, Y. Yang, and J. You. Making ambients more robust. In *Proceedings International Conference on Software: Theory and Practice*, pages 377–384, Beijing, China, 2000.
- [43] John Hannan and Dale Miller. From operational semantics to abstract machines. *Journal of Mathematical Structures in Computer Science*, 2(4):415–459, 1992.
- [44] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. In Uwe Nestmann and Benjamin Pierce, editors, *Proceedings of HLCL98*, volume 16.3, pages 3–17. Electronic Notes in Computer Science, Elsevier, 1998.
- [45] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 133–147. Springer-Verlag, 1991.
- [46] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
- [47] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent Tcl: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, 4(1):58–67, 1997.
- [48] Doug Lea. *Concurrent Programming in Java: Design principles and Patterns*. Addison-Wesley, 2nd edition, 1999.
- [49] F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proceedings POPL 2000*, pages 352–364. ACM Press, 2000.
- [50] J.-J. Levy. Some Results in the Join-Calculus. In Martin Abadi and Takayaso Ito, editors, *Proceedings of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 233–249. Springer, 1997.
- [51] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, USA, 1997.

- [52] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference*, pages 283–290, San Francisco, California, 1992.
- [53] L. Lopes, A. Figueira, F. Silva, and V. Vasconcelos. A Concurrent Programming Environment with Support for Distributed Computations and Code Mobility. In *Proceedings CLUSTER'00*, pages 297–306. IEEE Press, 2000.
- [54] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: Implementing Mobile Objects in the Realm of Process Calculi. In *Proceedings of the 5th Mobile Object Systems Workshop (MOS'99)*, Lisbon, Portugal. part of the ECOOP'99 Conference.
- [55] Luís Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, Faculty of Sciences, University of Porto, October 1999.
- [56] Luís Lopes, Vasco Vasconcelos, and Fernando Silva. Fine grained multithreading with process calculi. *IEEE Transactions on Computers*, 50(9):229–233, August 2001.
- [57] Sun Microsystems. The Java Language: An Overview. Technical report, Sun Microsystems, 1994.
- [58] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [59] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (part I). Technical Report ECS-LFCS-89-85, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1989.
- [60] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (part II). Technical Report ECS-LFCS-89-86, Laboratory for Foundations of Computer Science, University of Edinburgh, June 1989.
- [61] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.
- [62] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. 1990.

- [63] Robin Milner. The polyadic π -calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.
- [64] Robin Milner. *Communicating and Mobile Systems: The π -calculus*. Cambridge University Press, May 1999.
- [65] D. Milojevic, F. Douglass, and R. Wheeler, editors. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, Reading, MA, USA, 1999.
- [66] Dejan S. Milojevic, William LaForge, and Deepika Chauhan. Mobile Objects and Agents (MOA). In *Proceedings of the 4th Conference on Object Oriented Technologies and Systems (COOTS'98)*, pages 179–194. USENIX Association, 1998.
- [67] S. Mullender and G. Rossum. Amoeba: A distributed operating system for the 1990s. In A. Ananda and B. Srinivasan, editors, *Distributed Computing Systems: Concepts and Structures*, pages 201–212, Los Alamos, CA, 1992. IEEE Press.
- [68] J. Ousterhout, A. Cherenon, F. Douglass, M. Nelson, and B. Welch. The Sprite Network Operating System. Technical Report UCB/CSD/87/359, Computer Science Division (EECS), University of California, Berkeley, June 1987.
- [69] H. Paulino, P. Marques, L. Lopes, V. Vasconcelos, and F. Silva. A Multi-threaded Asynchronous Language. In *Parallel Computing Technologies - 7th International Conference, PaCT'03*, volume 2763 of *LNCS*, pages 316–323, August 2003.
- [70] B. Pierce. Foundational calculi for programming languages. In Allen Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press and ACM, 1997.
- [71] B. Pierce. Programming in the Pi-Calculus: A Tutorial Introduction to Pict (Pict Version 4.0). Technical report, University of Indiana, March 1997.
- [72] António Ravara, Ana Matos, Vasco Vasconcelos, and Luís Lopes. A Lexically Scoped Distributed π -calculus. Technical Report DI/FCUL TR-02-04, Department of Computer Science, University of Lisbon, April 2002.

- [73] J. Riely and M. Hennessy. Distributed Processes and Location Failures. volume 1256, pages 471–481, 1997.
- [74] R. Rouaix. A web navigator with applets in caml. *Computer Networks and ISDN Systems*, 28(7-11):1365–1371, 1996.
- [75] D. Sangiorgi. A theory of bisimulation for the π -calculus. In *Acta Informatica*, volume 33, pages 69–97. 1996. Earlier version published as Report ECS-LFCS-93-270, University of Edinburgh.
- [76] D. Sangiorgi. An interpretation of typed objects into typed π -calculus. *Information and Computation*, 143(1), 1998.
- [77] D. Sangiorgi. Interpreting functions as π -calculus processes: A tutorial. Technical Report RR-3470, INRIA, August 1998. Revised on February 1999.
- [78] Peter Sewel. On implementatios and semantics of a concurrent programming language. In Antoni Mazurkiewicz and Józef Winkowski, editors, *Proceedings of CONCUR'97*, volume 1243 of *LNCS*, pages 391–405. Springer, 1997.
- [79] P. Sewell, P. Wojciechowski, and B. Pierce. Location-independent communication for mobile agents: a two-level architecture. Technical Report 462, Computer Lab, University of Cambridge, 1998.
- [80] Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 4th edition, November 1994.
- [81] M. Straber, J. Baumann, and F. Hohl. Mole - A Java Based Mobile Agent System. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, 1996.
- [82] A. Tanenbaum, R. Renesse, H. Staveren, G. Sharp, S. Mullender, J. Jansen, and G. Rossum. Experience with Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.
- [83] Andrew Tanenbaum. *Modern Operating Systems*. Prentice-Hall, New Jersey, 1992.
- [84] Tommy Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997. Also Technical Report 1083, University of Rennes, IRISA.

- [85] D. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.
- [86] V. Vasconcelos. Processes, Functions, Datatypes. 5(2):97–110, 1999.
- [87] V. Vasconcelos and R. Bastos. Core-tyco – the language definition. Technical Report TR-98-3, DI/FCUL, March 1998.
- [88] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.
- [89] Vasco Vasconcelos, Luís Lopes, and Fernando Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16 of *Electronic Notes in Theoretical Computer Science*, pages 19–34. Elsevier Science, 1998.
- [90] Vasco T. Vasconcelos. *A Process Calculus Approach to Typed Concurrent Objects*. PhD thesis, Keio University, November 1994.
- [91] J. Vitek and G. Castagna. Towards a Calculus of Secure Mobile Computations. In *Workshop on Internet Programming Languages*, Chicago, Illinois, May 1998. IEEE.
- [92] J. Vitek and G. Castagna. Seal: A Framework for Secure Mobile Computations. In *Proceedings Internet Programming Languages*, volume 1686 of *LNCS*. Springer-Verlag, 1999.
- [93] J. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press / MIT Press, 1996.
- [94] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the java system. *Computing Systems*, 9(4):265–290, 1996.
- [95] D. Wong, N. Paciorek, T. Walsh, and J. DiCeglie. Concordia: An Infrastructure for Collaborating Mobile Agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the 1st International Workshop Mobile Agents 97*, volume 1219 of *LNCS*, pages 86–97. Springer-Verlag, 1997.