MIKADO Global Computing Project IST-2001-32222

Mobile Calculi Based on Domains

Analysis of distribution structures: state of the art

MIKADO Deliverable D3.1.1

Title :	Analysis of distribution structures: state of the art
Editor :	R. De Nicola (U. of Florence)
Authors :	M. Boreale, R. De Nicola, L. Bettini (U. of Florence), M. Lacoste (FTR&D), V. Vasconcelos (U. of Lisbon)
Classification :	Deliverable D3.1.1, Public
Reference :	RR/WP3/1
Version :	1.0
Date :	July 2002

© INRIA, France Telecom R&D, U. of Florence, U. of Sussex, U. of Lisbon

Abstract

The goal of this document is to give an overview of the state of the art in execution structures for global programming languages. The introductory section touches upon the main issues that arise in connection with the design of global applications. Existing paradigms, technologies and languages for global programming are the subject of Part 1. Part 2 analyzes in detail the execution structures of those languages that are central to the project.

Contents

1 Issues in global programming **5**

I Paradigms, Technologies and Languages **7**

2 The distributed objects paradigm **7**

3 The mobile code paradigm **7**

4 Supporting technologies **9**
4.1 Corba 9
4.2 Soap 9
4.3 The Java framework 10
4.4 .Net Framework 11

5 Languages **12**
5.1 Pict 12
5.2 Obliq 12
5.3 Network Objects 12
5.4 Oz 13
5.5 Garf 13
5.6 Language implementations 13

II Analysis of Execution Structures **15**

6 X-Klaim and Klava **17**
6.1 Main References 17
6.2 Execution Structure 17
6.2.1 General Architecture 17
6.2.2 Processor Abstraction 18
6.2.3 Basic Objects 18
6.2.4 Communication Primitives 19
6.2.5 Execution Environment 20
6.2.6 Memory Management 20
6.3 Distribution 20
6.3.1 Naming and Identification 20
6.3.2 Communication Mechanism 20
6.3.3 Fault Tolerance 21
6.4 Mobility 21
6.4.1 Execution unit 21
6.4.2 Type of Mobility 21
6.4.3 Hosting Structure 21
6.4.4 Resource Management 21
6.4.5 Quality of Service 21

6.5 Security . . . 22
6.5.1 Execution Errors . . . 22
6.5.2 Identification and Authentication . . . 22
6.5.3 Access Rights . . . 22
6.5.4 Communication Security . . . 22
6.6 Proof of Correctness . . . 22
6.7 Simple Case Study . . . 22

7 TyCO/DiTyCO 25
7.1 Main References . . . 25
7.2 Execution Structure . . . 26
7.2.1 General Architecture . . . 26
7.2.2 Processor Abstraction . . . 27
7.2.3 Basic Objects . . . 27
7.2.4 Communication Primitives . . . 28
7.2.5 Execution Environment . . . 28
7.2.6 Memory Management . . . 30
7.3 Distribution . . . 30
7.3.1 Naming and Identification . . . 30
7.3.2 Communication Mechanism . . . 31
7.3.3 Fault Tolerance . . . 31
7.4 Mobility . . . 31
7.4.1 Execution unit . . . 31
7.4.2 Type of Mobility . . . 31
7.4.3 Hosting Structure . . . 31
7.4.4 Resource Management . . . 31
7.4.5 Quality of Service . . . 31
7.5 Security . . . 32
7.5.1 Execution Errors . . . 32
7.5.2 Identification and Authentication . . . 32
7.5.3 Access Rights . . . 32
7.5.4 Communication Security . . . 32
7.6 Proof of Correctness . . . 32
7.7 Simple Case Study . . . 32

8 Safe Mobile Ambients 34
8.1 Main References . . . 34
8.2 Execution Structure . . . 35
8.2.1 General Architecture . . . 35
8.2.2 Processor Abstraction . . . 36
8.2.3 Basic Objects . . . 36
8.2.4 Communication Primitives . . . 36
8.2.5 Execution Environment . . . 36
8.2.6 Memory Management . . . 36
8.3 Distribution . . . 37

8.3.1 Naming and Identification . . . 37
8.3.2 Communication Mechanism . . . 37
8.3.3 Fault Tolerance . . . 37
8.4 Mobility . . . 37
8.4.1 Execution unit . . . 37
8.4.2 Type of Mobility . . . 38
8.4.3 Hosting Structure . . . 38
8.4.4 Resource Management . . . 38
8.4.5 Quality of Service . . . 38
8.5 Security . . . 38
8.5.1 Execution Errors . . . 38
8.5.2 Identification and Authentication . . . 38
8.5.3 Access Rights . . . 38
8.5.4 Communication Security . . . 39
8.6 Proof of Correctness . . . 39
8.7 Simple Case Study . . . 39

9 Jcl and JoCaml **40**
9.1 Main References . . . 40
9.2 Execution Structure . . . 41
9.2.1 General Architecture . . . 41
9.2.2 Processor Abstraction . . . 41
9.2.3 Basic Objects . . . 42
9.2.4 Communication Primitives . . . 42
9.2.5 Execution Environment . . . 43
9.2.6 Memory Management . . . 43
9.3 Distribution . . . 43
9.3.1 Naming and Identification . . . 43
9.3.2 Communication Mechanism . . . 44
9.3.3 Fault Tolerance . . . 44
9.4 Mobility . . . 44
9.4.1 Execution Unit and Hosting Structure . . . 44
9.4.2 Type of Mobility . . . 45
9.4.3 Resource Management . . . 45
9.5 Security . . . 45
9.6 Proof of Correctness . . . 45
9.7 Simple Case Study . . . 45

1 Issues in global programming

A global application is typically composed by a number of threads, whose code comes from different and, possibly, untrusted sites in the net, that run concurrently and interact also remotely. For instance, a browser can locally execute applets downloaded from different sites. Of course, traditional distributed applications can be multithreaded too, but here the point is that global applications are network dependent. In what follows, we illustrate the main issues one has to consider when designing and developing applications over a global network.

Mobility A global network supports and exploits different forms of *mobility*. Namely, it supplies mechanisms for connecting devices and services while (or after) they physically move, and mechanisms that permit migration among hosts of units of computation.

Scalability A global network is highly dynamic. Hence, it should be easily scalable in order to manage the addition/removal of hosts, subnetworks and users without requiring hosts and applications to be reconfigured.

Connectivity A net connection may not be available all the time to all hosts: hosts such as, e.g., PDA may be part of the net only for a short period of time. Moreover most connections, in spite of being continuous, may suffer from unpredictable congestion and failures. This causes fluctuations and temporary interruptions of bandwidth. Bandwidth, even in absence of such failures, may not be constant because of network traffic and sharing of the same communication channels and devices. Deciding whether a failure or a long waiting is due to a computational error or to a long delay may be impossible: the failure of a network link is not distinguishable from the failure of a processor or a component on the other side of the link and thus a time out error is often raised in these situations. These problems are quite limited in *Local Area Networks* (*LAN*), because the net is faster and dedicated to a particular group of computers, and response times can be more easily estimated; in a global network, communication problems are amplified.

Heterogeneity A global net is a heterogeneous infrastructure, since it connects hosts with different architectures and different operating systems. Interaction among different hosts is still possible thanks to protocols which establish conventions about the format and the language of the communication. However, exploiting mobility raises new problems because executable code is inherently architecture and system dependent.

Security Global applications rely on network communications thus, apart from being potential victims of computer viruses, can be tampered and eavesdropped in many ways [Che91, Spa89, VJ99], and for this reason they are basically vulnerable and insecure [Ord96]. In general, mobile code and communications may need to cross administrative domains and firewalls, and untrusted sites as well. Techniques such as *cryptography* may solve some problems related to data communications, but, when executable code is exchanged, such techniques may not be sufficient anymore. Much research effort has been devoted on protecting servers from untrusted and potentially malicious mobile code. On the converse, mobile code must

be protected from potentially malicious servers that might try to modify it or other sensible private data [Yee99, FGS96, ST98].

Locality The distribution of resources (at low level) should be as transparent as possible to the end user, who, however, should be aware of the fact that an interaction with a procedure/service on a remote computer requires much more time than an operation on the local computer. Additionally, if the user is aware of where the computation takes place, he can also argue about possible causes of failures or long waiting (probably due to network bandwidth and latency).

Barriers Global structures may not be exploited totally also because of barriers that administrative domains build and that make some direct communications among specific computers impossible. For instance, there can be firewalls that disable the access to a few services of some computers.

The rest of the document is organized as follows. Part I is devoted to describing paradigms, technologies and languages for global programming. In particular, we first survey the main paradigms for global programming, that is distributed objects (Section 2) and mobile code (Section 3). Next, we review industrial technologies for the support to global programming (Section 4). An overview of a few, mostly academic, proposals for distributed programming languages is finally presented (Section 5). In Part II we examine in detail those languages that are central to the project (Sections 6 through 9).

Part I
Paradigms, Technologies and Languages

2 The distributed objects paradigm

Traditional distributed applications exploit an object oriented design technique over an infrastructure which provides a transparent access to *distributed objects* [Car95, BNOW95, Obj98, W3C]: one can call a remote object's method as if it were local, once a remote reference through a central server (often a name server) has been obtained.

Distributed objects are basically *active objects*, i.e. objects whose code may be executed by multiple threads, or, in other words, objects that contain multiple threads of execution. This seems natural, since an application over a global network is inherently *concurrent*, and thus it is composed of many processes (or threads) possibly located on different hosts, interacting with each other. Coordinating multiple distributed processes requires efficient means of synchronization. Indeed, processes that are supposed to be coordinated may run on different hosts and few assumptions about their execution speed can be made. Moreover, synchronizing many distributed threads may require too much time, thus, together with such issues as network latency and partial failures, decreasing the performance of the involved applications.

A commonly accepted technology for distributed objects interaction is to have a suitable infrastructure where the implementation of the objects is kept independent of their interfaces. Then, the mechanisms used to implement a method invocation can significantly differ depending on the location of the object, but are still hidden to the programmer that can use local and remote objects in the same way. Therefore, the communication paradigm may end up being the same as the language paradigm; however, the programmer must be aware of the objects location and of where the computation actually takes place, since the difference between a remote and a local method invocation is estimated in some orders of magnitude. An implementation that ignores such a difference might have serious performance problems especially if it requires a large amount of communication among components that physically reside on different machines.

3 The mobile code paradigm

It has been widely argued that mobility will be an important technology for applications over a global network [HCK94, Kna96, CGPV97, Tho97, LO99]. The main breakthrough is that global applications may exchange active units of behaviour (mobile code), not just passive data. In what follows, we present some basic issues concerning mobile code.

In the literature, the expression *Mobile Code* is used with different meanings; here are some excerpts:

- "The term *mobile code* describes any program that can be shipped unchanged to a heterogeneous collection of processors and executed *with identical semantics* on each processor" [ATLLW96].

- "... *mobile code*, an approach where programs are considered as documents, and should therefore be accessible, transmitted and displayed (i.e. evaluated) as any other document" [Rou96];
- "... *mobile code* is software that travels on a heterogeneous network, crossing protection domains, and is automatically executed upon arrival at the destination" [Tho97].

Mobile code has given rise to new programming paradigms [CGPV97, CPV97] other than the traditional client-server one:

- *Remote Evaluation*: some code is sent for execution to a remote host;
- *Code On-Demand*: some code is downloaded from a remote host to be executed locally;
- *Mobile Agents*: processes can suspend their execution and migrate to new hosts, where they can resume their execution.

Mobile agents are different from remote evaluation and code on-demand in that the code does not need to be sent and retrieved explicitly: the agent migrates autonomously somewhere else and can continue executing there. Mobile agents are quite transparent even for the end user, who, however, cannot ignore that:

- an agent is basically a program, hence, in order to run, it needs a so called *execution environment*, i.e. a server that supplies resources for execution (*runtime support*);
- an agent is *autonomous*: it executes independently of the user who created it (*goal driven*);
- an agent is able to sense the changes in the environment and to act according to these changes (*reactivity*);
- an agent acts according to previous experience (*adaptivity* and *learning*).

An agent is *self contained* in that it possesses all the data it needs to execute and migrate, since this information is typically carried with the agent during migration. Decisions of executing and moving are taken according to information supplied by the user of the agent. However, the agent may also autonomously decide to take different choices due to contextual events such as, e.g., network temporary unavailability or host not responding.

We may think of an agent as a class (of a class-based programming language), though thinking of it as a *wrapper* of a process with its code, execution state and data (a kind of *closure*), would be more appropriate. An agent is an active entity, because it contains a thread that is running (while the agent is executing) or sleeping (while the agent is suspended or moving).

Maybe one of the most interesting features of mobile agents, apart from mobility and autonomy, is the possibility to implement *disconnected operations* [PR98]: an agent may remotely execute even if the user (its owner) is not connected; if this is the case, the agent may decide to "sleep" and then periodically try to reestablish the connection with its owner. Conversely, the user, when reconnected, may try to *retract* the agent back home (i.e. instruct the remote agent to go back its home site).

4 Supporting technologies

In this section we describe four frameworks that can be used for developing infrastructures for either distribute objects or mobile code systems. More specifically we consider CORBA, SOAP, the Java framework and the .Net framework. CORBA is an architecture oriented towards distributed objects. The other three technologies can be used for supporting both distributed objects and mobile code programming. However, while SOAP just provides a lightweight messaging framework, Java and .Net are comprehensive infrastructure.

4.1 Corba

The *Common Object Request Broker Architecture* (*CORBA*) [Obj98] makes remote method invocation independent of the language thus permitting not only connecting heterogeneous computers but also integrating code written in many different mainstream programming languages. More specifically, *CORBA* integrates *RPC* (*Remote Procedure Calls*) with the object oriented paradigm, trying to combine the benefits of the latter (modularity and reuse) with the communication mechanisms of the former.

The primary components of CORBA are: Object services, Common Facilities, Domain Interfaces, Application Interfaces and Object Request Broker (ORB).

Object services, *Common Facilitie* and *Domain Interfaces* provide, respectively, facilities at independent-domain level, end-user application level and application-domain level. Examples of facilities are those that permits clients to find objects based, respectibely, on names (*Naming Service*) and on their properties (*Trading Service*). *Application Interfaces* are interfaces developed specifically for a given application. These interfaces are not standardized.

The *Object Request Broker* (ORB) provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the client.

4.2 Soap

SOAP [W3C] is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible framework for the exchange of messages over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics.

SOAP provides a distributed processing model that assumes a message originates at an initial sender and is sent to an ultimate receiver via zero or more intermediaries. Note that the SOAP distributed processing model can support many Message Exchange Patterns (MEPs) including but not limited to one-way messages, request/response interactions, and peer-to-peer conversations.

SOAP enables exchange of SOAP messages using a variety of underlying protocols. The formal set of rules for carrying a SOAP message within or on top of another protocol (underlying protocol) for the purpose of exchange is called a binding.

SOAP framework easily permits programming *Remote Procedure Call*. Differently from CORBA and RMI, SOAP does not require a specific infrastructure and it is not language-dependent. Moreover, since SOAP it is based on XML, *Object services* can be deployed as *Web services*.

4.3 The Java framework

Java [AGH00] is a programming language that supplies many useful abstractions for programming distributed applications that exploit forms of mobility. In fact, several mobile agent programming environments have been built on top of Java, we just mention *Mole* [SBH96], *Concordia* [WPWD97], *Mobile Objects and Agents* (*MOA*, [MLC98]) and *Voyager* [Gla99].

Java provides class libraries that contain many useful features for programming distributed applications [HSHB99] such as, e.g., *sockets* and *multi-threading*. Moreover, in Java, *synchronization* is a primitive notion (a class with synchronized methods is basically a *monitor*). Hence, coordinating concurrent and distributed processes can effectively be accomplished by exploiting these features [Lea99].

Moreover, Java is *architecture independent*: a Java program is compiled into an intermediate language (the *byte code*), which is executed by a virtual machine (the *Java Virtual Machine*). Thus, a Java program can be executed in any computer endowed with such an interpreter.

Finally, Java provides functionalities for loading dynamically *objects* and *classes* from remote sites. Java *serialization* supplies an easy and customizable way to write the state of an object onto a stream, which, in turn, can be connected to any communication channel, such as a file or a net socket. Java implements *dynamic class loading*, i.e. the code of a class is linked and loaded in memory at run time, when it is needed (e.g. when an object of that class is created). Thus, new pieces of code can be loaded even from the net. This feature is very useful for distributed applications that require such a kind of extensibility. Forms of code mobility can be easily implemented in this way.

To guarantee system security Java implements many mechanisms. For instance, some direct memory and execution state accesses are denied and a strict control on the code, both at compile time (strong type checking) and at execution time (*byte code verifier*), is performed. Moreover, Java allows programmers to customize many security policies [Gon99] such as, e.g., file system access and system calls.

Java RMI Java Remote Method Invocation (RMI) [Sunb] allows programmers to write distributed programs providing a simple and direct model for distributed computation with Java objects. These objects can be new Java objects, or can be simple Java wrappers around an existing API.

Basically, RMI is Java's remote procedure call (RPC) mechanism. RMI has several advantages over traditional RPC systems, because it is part of Java's object oriented approach. Traditional RPC systems are language-neutral, and therefore are essentially least-common-denominator systems: they cannot provide functionality that is not available on all possible target platforms. Conversely, one of the main advantage of RMI is that it permits passing full objects as arguments and return values, not just predefined data types. This means that a programmer can pass complex types as a single argument to a remote method. In existing

RPC systems the client would have to decompose such an object into primitive data types, ship those data types, and the re-create the complex object. RMI lets the client ship objects directly across the wire with no extra code.

Sun ONE Sun Open Network Environment [Suna] (Sun ONE) is the new environment of Sun Microsystem for developing network oriented applications. Sun ONE aims at providing an highly scalable and robust foundation for traditional software applications as well as current web-based applications, while laying the foundation for the distributed computing models such as Web services.

4.4 .Net Framework

The .NET Framework [Mic] is a new computing platform that simplifies application development in the highly distributed environment of the Internet.

Basically the .NET Framework aims at providing a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but remotely downloaded, or executed remotely.

Moreover, .Net Framework aims at providing a code-execution environment that guarantees safe execution of code, including code created by an unknown or semi-trusted third party.

The .NET Framework has two main components: the common language runtime and the .NET Framework class library.

The common language runtime is the foundation of the .NET Framework. The runtime can be thought of as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that ensure security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code.

The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that can be used to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

The .NET Framework can be hosted by unmanaged components that load the common language runtime into their processes and initiate the execution of managed code, thereby creating a software environment that can exploit both managed and unmanaged features. The .NET Framework not only provides several runtime hosts, but also supports the development of third-party runtime hosts.

.Net Remoting Framework .NET Remoting [OH] provides a framework that allows objects to interact with one another across application domains. The framework provides a number of services, including activation and lifetime support, as well as communication channels responsible for transporting messages to and from remote applications. Formatters are used for encoding and decoding the messages before they are transported by the channel.

Applications can use binary encoding where performance is critical, or XML encoding where interoperability with other remoting frameworks is essential. All XML encoding uses the SOAP protocol in transporting messages from one application domain to the other. Remoting was designed with security in mind, and a number of hooks are provided that allow channel sinks to gain access to the messages and serialized stream before the stream is transported over the channel.

5 Languages

We survey some recent, mostly academic, proposals for languages that are meant to support global/distributed programming. We also discuss the issue of programs and implementation correctness with respect to the formal definition of the language. The overview below does not mention four languages that are central to the project (X-Klaim, Mobile Ambients, TyCO and JoCaml), whose detailed analysis is postponed to Part 2.

5.1 Pict

Pict [PT97] is a concurrent functional programming language based on the primitives of π-calculus [MPW92]. Communication takes place through channels and asynchronously. Channels are accessed through names, thus Pict provides mobility of names. An extension of Pict, *Nomadic Pict* [WS00], supplies mobility of code.

Pict is statically type checked; typing is partially implicit thanks to a mechanism of type inference. The name restriction of the π-calculus is a first form of privacy, since a restricted name (channel) can be used only by processes belonging to the scope of the restriction.

The *Spi calculus* [AG98] is an extension of π-calculus with cryptographic primitives that can be used for the analysis of security protocols, in particular for verifying properties such as integrity and confidentiality.

5.2 Obliq

Obliq [Car95] is an interpreted object-based programming language implemented in Modula-3. It enables distributed objects, remote procedure calls, and code mobility. It is location transparent and lexically scoped: when a process moves to a remote site the references to local resources becomes network references, thus the moved code can still access the resources of the home site as if they were local. Apart from this form of mobility of names, Obliq also supports weak mobility of code.

Obliq is not typed checked statically, however at run-time the code is strongly checked. It also provides two forms of protection: *protected objects* can be accessed only partially (e.g. they cannot be cloned) and the lexically scoped nature of the language provides a primitive form of resource access control.

5.3 Network Objects

Network Objects [BNOW95] is a library extension of Modula-3 enabling remote procedure call services. Objects are not mobile, but they can be passed by value or by reference. Objects

are passive entities and they do not possess mechanism for synchronization based on method calls and they are not persistent.

Since Modula-3 is strongly typed, the majority of checking is performed statically. Moreover the extension *Secure Network Objects* [DABW99] extends the library with operations for guaranteeing integrity and non-repudiation of references and objects.

5.4 Oz

Oz [Smo95] is a language based on logic programming through constraints (CLP), thus it is based on *rules* (a formula of first order logic with values and variables). The *store* is the execution environment of threads and lexical scope is respected within threads. Morever it is location transparent.

Both synchronous and asynchronous communications are supported, and they are based on *ports*. Both name and code mobility are supported, and the mobility is implicit (indeed mobility is supported by all the entities of the language).

The virtual machine is based on the *fail-stop* failure model, however specific errors can be treated by the programmer directly. The virtual machines also implements a ticket mechanism for establishing connections among Oz sites. Moreover virtual sites can be created in order to keep track of the consumption of resources for specific computations.

5.5 Garf

Garf [GGM97] is an actor language based on SmallTalk. The virtual machine of SmallTalk allows the execution of concurrent objects and message exchange. Garf consists of a library of classes and relies on a run-time support for the communication among virtual machines.

There is no difference between remote and local communication. Communication is synchronous by default (as in SmallTalk), but means for obtaining asynchronous communications are also supplied.

Garf is essentially resistant to failures, thus a failure can never alterate the functionality of a system. Error handling is left to the programmer.

5.6 Language implementations

The correctness of programming language implementation confronts language designers and implementors. Traditionally, such implementations are built "by hand", and very seldom proved correct. When such an attempt is taken, the resulting implementation have little relationship to the language's semantic specifications and so correctness is difficult to prove. Alternatively, a language implementation is constructed from the semantic specification in such a way that the resulting implementation is guaranteed to satisfy correctness requirements. This approach provides a mechanical process to construct machines whose correctness is guaranteed. While highly desirable, not much work on *provably correct implementations* has been done in the realm of concurrent languages, let alone distributed languages.

Hannan and Miller construct abstract machines for sequential languages by repeatedly applying correctness-preserving transformations to operational semantics, until the resulting specifications have the form of abstract machines [HM92]. The specifications are given by

proof systems axiomatizing evaluation relations; the proofs constructible in these systems provide concrete objects representing the computation steps performed during evaluation.

For concurrent languages, Turner describes a simple abstract machine to implement π-calculus computations [Tur96]. The initial machine is successively refined toward a realistic implementation; a C-based machine is lastly presented. Correctness proof is given only for the first abstract machine. On a similar vein, Lopes, Vasconcelos, and Silva start from the TyCO process calculus and provide a series of (multi-threaded) abstract machines [LVS01]. For each machine, a translation to the previous machine is presented and soundness proved. Sewell studies the relation between the semantics of Pict, a language based on the π-calculus, and its actually implementation considered as the abstract behavior of a C-program [Sew97].

Part II
Analysis of Execution Structures

Below, we list some criteria that we will use for analyzing the four languages that are crucial for the project: X-KLAIM (Section 6), TyCo (Section 7), Safe Mobile Ambients (Section 8) and JOCAML (Section 9). These criteria explain how basic concepts like location abstractions, communication, mobility and security are treated in implementations.

Main references: constains a list of references to papers regarding the language.

Execution Structure: describes the general structure of the implementation.

- **General architecture:** contains a short overview of the language architecture.
- **Processor Abstraction:** describes the kind of concurrency (multithread, singlethread,...) that is implemented/used in the language.
- **Basic Objects:** describes the basic entities of the language.
- **Communication primitives:** identifies the primitives that programs can use to communicate.
- **Execution Environment:** describes the enviroment where programs are executed in.
- **Memory Management:** describes the memory management policies (garbage collection, allocation policies) adopted in the implementation.

Distribution: describes how the language deals with explicit location of distributed sites, and in particular whether the programmer can or has to specify this location.

- **Naming and Identification:** describes the naming/addressing mechanisms used in the language implementation.
- **Communication Mechanism:** describes the communication paradigm adopted by the language.
- **Fault Tolerance:** describes the primitives used to guarantee fault tolerance (accesses to broken-resources).

Mobility: describes the kind of mobility supported by the language.

- **Execution unit:** describes the subject of mobility, the part of the system that is moved.
- **Type of mobility:** characterizes the kind of mobility (physical/logical, strong/weak) that the language provides.
- **Hosting structure:** describes the structure that receives and executes execution units moved in the system.
- **Quality of service:** contains the primitives that programmers can use to guarantee quality of services.

Security: identifies the security mechanisms that are implemented in the language.

Execution errors: contains mechanims for handling errors at runtime.

Identification and authentication: descibes the language's primitives for secrecy.

Access rights: describes the primitives that the language provides for programming accesses to resources whithin programs.

Communication security: identifies primitives and mechanisms that are used in the language implementation for guaranteeing communication security.

Proof of Correctness: discusses results concernin the correctness of the implementation with respect to the language model.

Simple Case Study: contains a small example that clarify the use of the language for programming a simple application.

6 X-KLAIM and KLAVA

X-KLAIM (*eXtended* KLAIM) [BDFP98] is an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple tuple spaces and mobile code. It is based on the kernel language KLAIM (*Kernel Language for Agent Interaction and Mobility*) [DFP98] and is inspired by the coordination language Linda [Gel85].

6.1 Main References

- L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In P. Ciancarini and R. Tolksdorf, editors, *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 110–115. IEEE Computer Society Press, 1998.
- L. Bettini. Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile. Master's thesis, Dip. di Sistemi e Informatica, Univ. di Firenze, April 1998.
- L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java Framework for Distributed and Mobile Applications. *Software Practice and Experieces*, 2002, to appear.
- L. Bettini and R. De Nicola. Translating Strong Mobility into Weak Mobility. In Gian Pietro Picco, editor, *Mobile Agents*, number 2240 in LNCS, pages 182–197. Springer, 2001.
- L. Bettini and R. De Nicola. A Middleware for Secure Distributed Tuple Spaces. Draft, available at `http://music.dsi.unifi.it/papers.html`, 2002.
- Lorenzo Bettini, Rocco De Nicola, GianLuigi Ferrari, and Rosario Pugliese. Mobile Applications in X-KLAIM. In A. Corradi, A. Omicini, and A. Poggi, editors, *Proc. of WOA 2000*, 2000.
- Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. X-KLAIM and KLAVA: Programming Mobile Code. In M. Lenisa and M. Miculan, editors, *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
- L. Bettini, R. De Nicola, and M. Loreti. Software Update via Mobile Agent Based Programming. In *Proc. of ACM SAC 2002, Special Track on Agents, Interactions, Mobility, and Systems*. ACM Press, 2002.
- R. De Nicola and M. Loreti. A Modal Logic for Mobile Agents. *ACM Transaction of Computational Logics*, 2002. to appear.

6.2 Execution Structure

6.2.1 General Architecture

X-KLAIM relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are containers of information items (called *fields*). X-KLAIM is equipped with specific primitives

for inserting (retrieving) data and processes in (from) tuple spaces that are distributed over different hosts.

In Linda there is only one global shared tuple space; KLAIM extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*.

X-KLAIM extends KLAIM with a high level syntax for processes: it provides variable declarations, enriched operations, assignments, conditionals, sequential and iterative process composition. The implementation of X-KLAIM consists of KLAVA, a Java [AGH00] package that provides the run-time system for X-KLAIM operations, and a compiler, which translates X-KLAIM programs into Java programs that use KLAVA. The structure of the KLAIM framework is depicted in Figure 1. X-KLAIM and KLAVA are available on line at `http://music.dsi.unifi.it`. KLAVA is briefly described in [BDFP98] and presented in detail in [Bet98, BDP01].

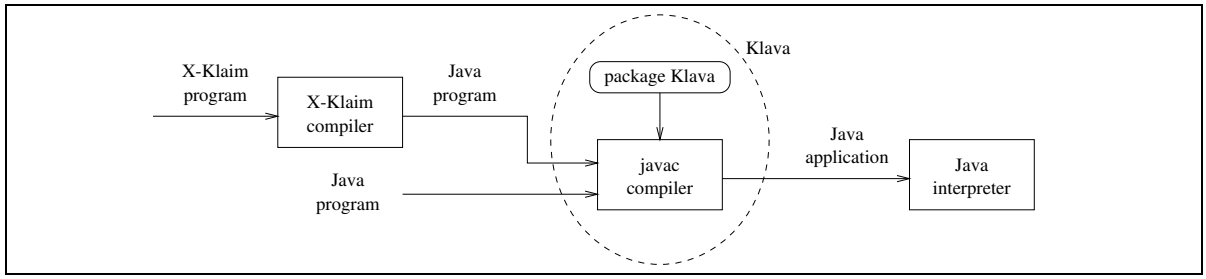

Figure 1: The framework for X-KLAIM.

If X-KLAIM source code is stored in a file called `foo.xklaim`, it can be compiled by means of the X-KLAIM compiler and the result will be the file `foo.java`. This last file can be compiled and executed by means of the standard jdk commands. When the program is executed, the host and the port number of the *Net server*, which is a class in the package KLAVA, must also be specified. This server keeps track of the physical localities of the nodes which are part of the net and must be started before any other node. Thus, X-KLAIM can be used to write the highest layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the KLAIM paradigm.

6.2.2 Processor Abstraction

A KLAIM process executes on a KLAIM node that is part of a KLAIM net. All these objects rely on Java threads.

6.2.3 Basic Objects

KLAIM relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are containers of information items (called *fields*). There are two kinds of fields: *actual fields* (i.e. expressions, processes, localities, constants, identifiers) and *formal fields* (i.e. variables). Syntactically, a formal field is denoted with !*ide*, where *ide* is an identifier. Tuples are anonymous and content-addressable; *pattern-matching* is used to select tuples in a tuple space:

- two tuples match if they have the same number of fields and corresponding fields have matching values or formals;
- formal fields match any value of the same type, but two formals never match, and two actual fields match only if they are identical.

For instance, tuple (`"foo"`, `"bar"`, 100 + 200) matches with (`"foo"`, `"bar"`, !*Val*). After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *Val* (an integer variable) will contain the integer value 300.

In Linda there is only one global shared tuple space; KLAIM extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*.

Processes are the main computational units and can execute operations over distributed tuple spaces.

6.2.4 Communication Primitives

the following operations over tuple spaces and nodes are available:

- **in**$(t)@l$: evaluates tuple t and looks for a matching tuple t' in the tuple space located at l. Whenever a matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available.
- **read**$(t)@l$: differs from **in**$(t)@l$ only because the tuple t' selected by pattern-matching is not removed from the tuple space located at l.
- **out**$(t)@l$: adds the tuple resulting from the evaluation of t to the tuple space located at l.
- **eval**$(P)@l$: spawns process P for execution at node l.
- **newloc**(l): creates a new node in the net and binds its physical locality to l. The node can be considered a "private" node that can be accessed by the other nodes only if the creator communicates the value of variable l, which is the only means to access the fresh node.

During tuple evaluation, expressions are computed and logical localities are translated into physical ones. Evaluating a process implies substituting it with its *closure*, namely the process along with the environment of the node where the evaluation is taking place. Hence, a remarkable difference between **out**$(P)@l$ and **eval**$(P)@l$ is that **out** adds the closure of P to the tuple space located at l, while **eval** sends P, not its closure, for execution at l. This will affect the evaluation of logical localities: when a process needs to translate a logical locality into a physical one, first its own allocation environment is used (if it has one) and then, if the translation fails, the environment of the node where the process runs is used. This means that a process delivered with an **out** will use a *static scoping* strategy for logical localities

while a process remotely spawned with an **eval** will use a *dynamic scoping* strategy. Thus, for instance, in the former case, `self` will refer to the originating site, while, in the latter case, `self` will refer to the current execution site.

X-KLAIM extends KLAIM with a high level syntax for processes: it provides variable declarations, enriched operations, assignments, conditionals, sequential and iterative process composition.

6.2.5 Execution Environment

KLAVA is implemented in Java, and it relies on Java byte-code both for execution and for code exchange.

6.2.6 Memory Management

The basic memory component is the tuple space. No other memory issue is considered either in the model and in the implementation (thus relying on the underlying run-time system, e.g. Java memory management and garbage collection).

6.3 Distribution

6.3.1 Naming and Identification

Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. There are two kinds of localities:

- *Physical localities* are the identifiers through which nodes can be uniquely identified within a net.
- *Logical localities* are symbolic names for nodes. A reserved logical locality, `self`, can be used by processes to refer to their execution node.

Physical localities have an absolute meaning within the net, while logical localities have a relative meaning depending on the node where they are interpreted and can be thought as aliases for network resources. Logical localities are associated to physical localities through *allocation environments*, represented as partial functions. Each node has its own environment that, in particular, associates `self` to the physical locality of the node.

6.3.2 Communication Mechanism

The communication takes place through tuples and tuple spaces. Pattern matching is used as a selection mechanism.

At a lower level, in KLAVA, nodes communicate by means of messages delivered through streams (connected to sockets). By default communications take place indirectly, through the net server. However, direct connections are also permitted: in this case, the first time a node A has to communicate with a node B (that accepts direct connections), the IP address of B is obtained through the net server and a direct connection between A and B is established (all these operations are performed automatically by KLAVA). Then, further messages between

these two nodes will be exchanged directly through this connection. In case of firewalls or network restrictions the access to a remote node may be permitted only through a net server. For instance, an applet can only open a network connection towards the computer it has been downloaded from. If on this computer there is a net server running, the applet is still able to indirectly communicate with all the nodes and, possibly, with applets that are part of that KLAVA net. In this sense, a `Net` object can also be seen as a gateway for nodes, allowing them to communicate even if they belong to different restricted domains.

6.3.3 Fault Tolerance

KLAVA does not provide direct support for saving processes and mobile agents into external memory in order to deal with nodes' shutdown and reboot. Safety can be recovered by explicitly programming serialization of components of a node (tuple space, processes, etc.).

6.4 Mobility

6.4.1 Execution unit

The main execution and migration unit is a KLAVA process. The classes needed by a process are automatically collected and delivered together with the process.

6.4.2 Type of Mobility

KLAVA can only supply *weak mobility* of agents, due to the fact that Java does not permit dynamic inspection of the byte code stack and this makes impossible to save the execution state for later use.

X-KLAIM also supplies *strong mobility* by means of action **go**@l that makes an agent migrate to l and resume its execution at l from the instruction following the migration. This kind of mobility is implemented on top of KLAVA by means of an appropriate precompilation phase [BD01].

6.4.3 Hosting Structure

At a lower level a KLAVA node relies on a customized class loader that receives a process together with its classes and takes care of loading these classes on demand, when needed by the process during its execution.

6.4.4 Resource Management

In the implementaion there is basically no means for resource control. For KLAIM a type system for access control was presented in [DFPV00, DFP99].

6.4.5 Quality of Service

A timeout (expressed in milliseconds) can be specified for **in** and **read**, through the keyword **within**; the operation becomes a boolean expression that can be tested in order to establish if the operation succeeded:

```
if in(!x, !y)@l within 2000 then
  # ... success!
else
  # ... timeout occurred
endif
```

Time-outs can be used when retrieving information for avoiding that processes block due to network latency bandwidth or to absence of matching tuples.

In KLAVA if the specified timeout (expressed in milliseconds) expires before the corresponding operation returns, then a `KlavaTimeOutException` exception will be thrown. Time-outs can thus be handled in a `try...catch` block.

6.5 Security

6.5.1 Execution Errors

Exceptions are used only in KLAVA. In X-KLAIM there is no direct exception mechanism. The package KLAVA relies on some customized exceptions dealing with errors such as a failed translation for a logical locality or the impossibility of establishing a direct connection with another node.

6.5.2 Identification and Authentication

None.

6.5.3 Access Rights

The type system for access control [DFPV00, DFP99] for KLAIM has not been implemented yet.

6.5.4 Communication Security

In [BD02] KLAVA is extended with cryptographic primitives: classical Linda operations are extended for handling encrypted data. Operations are also supplied for encrypting and decrypting tuple contents; This finer granularity allows mobile agents, which are not supposed to carry private keys with them when migrating, to collect encrypted data, while executing on remote sites, and decrypt them safely when back at the home site.

6.6 Proof of Correctness

There is no proof of correctness of the implementation at the moment.

We are working on applying the modal logic for KLAIM [DL02] also to X-KLAIM in order to proove properties of systems programmed in X-KLAIM [BDL02].

6.7 Simple Case Study

In this section we will show how to program a *news gatherer*, that relies on mobile agents for retrieving information on remote sites, using our framework. In particular we will implement this example both in X-KLAIM and in KLAVA. We assume that some data are distributed

over the nodes of a KLAVA net and that each node either contains the information we are searching for, or the locality of the next node to visit in the net. This example is taken from [DFP98], and in KLAIM can be specified[1] as shown in Listing 1.

$$
\begin{array}{l}
\mathit{NewsGatherer}(\mathit{item}, \mathit{retLoc}) = \\
\quad \mathbf{read}(\mathit{item}, !\mathit{itemVal})@\texttt{self}.\mathbf{out}(\mathit{itemVal})@\mathit{retLoc}.\mathbf{nil} \\
\quad + \\
\quad \mathbf{read}(\mathit{item}, !\mathit{nextLoc})@\texttt{self}.\mathbf{eval}(\mathit{NewsGatherer}(\mathit{item}, \mathit{retLoc}))@\mathit{nextLoc}.\mathbf{nil}
\end{array}
$$

Listing 1: The news gatherer specified in KLAIM.

The agent *NewsGatherer* tries to read one of two possible tuples: the first tuple contains information we are searching for and the second one the locality of the next node to visit. In the first case, the agent communicates the result to its owner and terminates, in the other case it simply spawns itself to the next node (Figure 2).

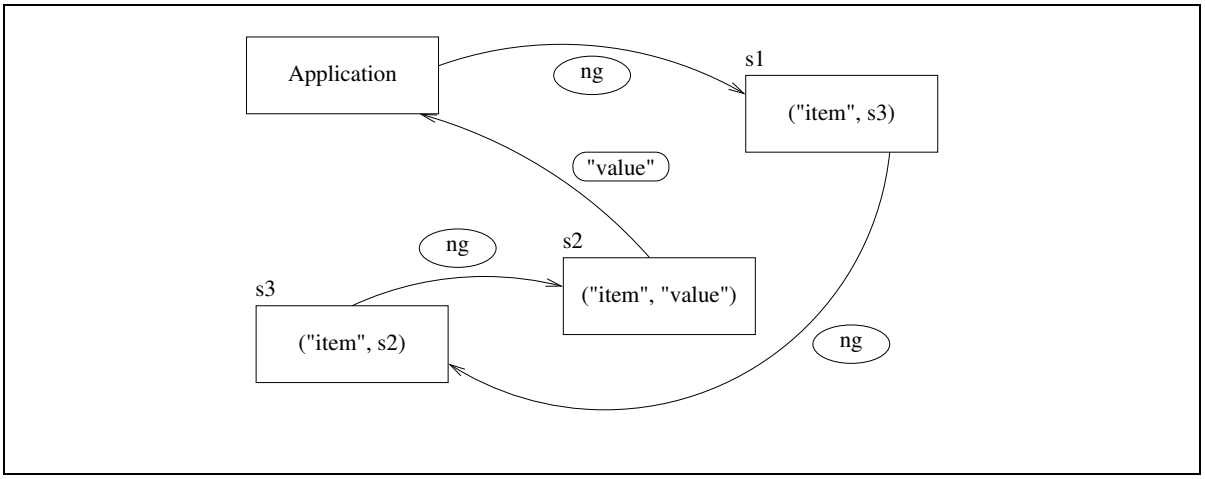

Figure 2: The news gatherer example (ng is the news gatherer agent).

The implementations in X-KLAIM and in KLAVA are reported, respectively, in Listing 2 and 3. If the result of the query is a locality, a new instance of the process `NewsGatherer` is remotely spawned by means of an **eval**. We use a timeout to test the presence of the tuple containing the information: if this is not found within two seconds, we retrieve the locality of the next node to visit. Notice that, also in KLAVA, for spawning a new process on a remote site, only an **eval** invocation is required: the underline system will take care of serializing the process through the network together with code and the code of all the classes it will use and the values of its fields. The two implementations are quite similar, and indeed the code generated by the X-KLAIM compiler is not much different from the one shown in Listing 3.

In X-KLAIM strong mobility can be exploited, thus the same example can be implemented by using the **go**@l operation, as illustrated in Listing 4.

[1] $P_1 + P_2$ spawns both processes P_1 and P_2, but only one of the two will continue its execution.

```
rec NewsGatherer[ item : str, retLoc : loc ]
  declare
    var itemVal : str ;
    var nextLoc : loc
  begin
    if read( item, !itemVal )@self within 2000 then
      out( itemVal )@retLoc
    else
      read( item, !nextLoc )@self ;
      eval( NewsGatherer( item, retLoc ) )@nextLoc
    endif
  end
```

Listing 2: The implementation of the news gatherer in X-KLAIM.

```
class NewsGatherer extends KlavaProcess {
  protected KString itemVal ;
  protected KString item ;
  protected Locality retLoc ;

  public NewsGatherer( KString item, Locality retLoc ) {
    this.item = item ;
    this.retLoc = retLoc ;
  }

  public void execute() throws KlavaException {
    itemVal = new KString();
    try {
      read( item, itemVal, self, 2000 );
      out( itemVal, retLoc );
    } catch (KlavaTimeOutException e) {
      Locality nextLoc = new PhysicalLocality();
      read( item, nextLoc, self );
      eval( new NewsGatherer( item, retLoc ), nextLoc );
    }
  }
}
```

Listing 3: The implementation of the news gatherer in KLAVA.

```
rec NewsGatherer[ item : str, retLoc : loc ]
  declare
    var itemVal : str ;
    var nextLoc : loc ;
    var found : bool
  begin
    found := false;
    while not found do
      if read( item, !itemVal )@self within 2000 then
        out( itemVal )@retLoc ;
        found := true ;
      else
        read( item, !nextLoc )@self ;
        go@nextLoc
      endif
    enddo
  end
```

Listing 4: X-KLAIM implementation exploiting strong mobility.

7 TyCO/DiTyCO

TyCO ("TYped Concurrent Objects") is a programming language based on an namesake asynchronous π-calculus with input-guarded labelled sums (modeling objects) and output selector (modeling asynchronous method invocations). It provides a form of concurrent object-based programming (that is, objects but no inheritance). The language is quite simple: the basic syntax reduces to half-a-dozen constructors, and can be thought of as an assembly language for concurrent, object-based programming. To help in writing common programming patterns, a few derived constructors are available. TyCO is strongly and implicitely typed: a type inference system assigns monomorphic types to variables, and (la ML) polymorphic types to process definitions.

DiTyCO ("Distributed TyCO") is an extension of TyCO that provides language and run-time level support for distributed computations and mobility. At the programming level it grows from TyCO by allowing channels to be located at sites. Sites are the programming unit modules. Two simple constructs define the primary interfaces of sites: export and import. The DiTyCO source code is compiled into byte-code for an extended TyCO virtual machine. Each site is implemented as a thread that runs an extended TyCO byte-code emulator with private stack and heap areas. These are embeded in an underlying infrastructure, invisible for the programmer that supports connectivity between the sites in the physical network. The theoretical framework for DiTyCO has been established in the $LSD\pi$ calculus.

7.1 Main References

- Antonio Ravara, Ana Matos, Vasco T. Vasconcelos, and Luis Lopes, *A Lexically Scoped Distributed π-Calculus.* Technical Report DI/FCUL TR-02-4. Department of Computer Science, University of Lisbon. April 2002.

- Luis Lopes, Ilvaro Figueira, Fernando Silva, Vasco Vasconcelos, *A Concurrent Programming Environment with Support for Distributed Computations and Code Mobility*, in International Conference on Cluster Computing (CLUSTER2000), Chemnitz, pp. 297-306, IEEE Computer Society Press, 2000.

- Luis Lopes, Fernando Silva, Vasco Vasconcelos, *A Virtual Machine for a Process Calculus*, International Conference on Principles and Practice of Declarative Programming (PPDP'99), LNCS 1702, pages 244-260, Springer-Verlag, 1999.

- Vasco Vasconcelos, Luis Lopes, Fernando Silva, *Distribution and Mobility with Lexical Scoping in Process Calculi*, 3rd International Workshop on High-Level Concurrent Languages (HLCL'98), part of CONCUR'98, Nice, France. ENTCS 16(3), Elsevier Science Publishers, 1998.

7.2 Execution Structure

7.2.1 General Architecture

The TyCO Virtual Machine (TyCOVM) runs byte-code files generated by the TyCO compiler. The compiler, incorporating a type-inference system, gathers static information that is used to generate optimized code. The design and implementation of the virtual machine focuses on performance, compactness, and architecture independence with a view to mobile computing.

The DiTyCO implementation has three layers: network, nodes and sites, although only two – network and sites, are logical (fig. 3). Sites form the basic sequential units of computation.

Sites are simply extended TyCOVMs. They are hosted by nodes, implemented as an underlying layer of software that provides communication services (e.g., remote communication). Nodes have a one-to-one correspondence with physical IP nodes and may have an arbitrary number of sites computing concurrently. Finally, the network is composed of multiple DiTyCO nodes connected in a static IP topology. This layer provides services to nodes, mainly name-space resolution.

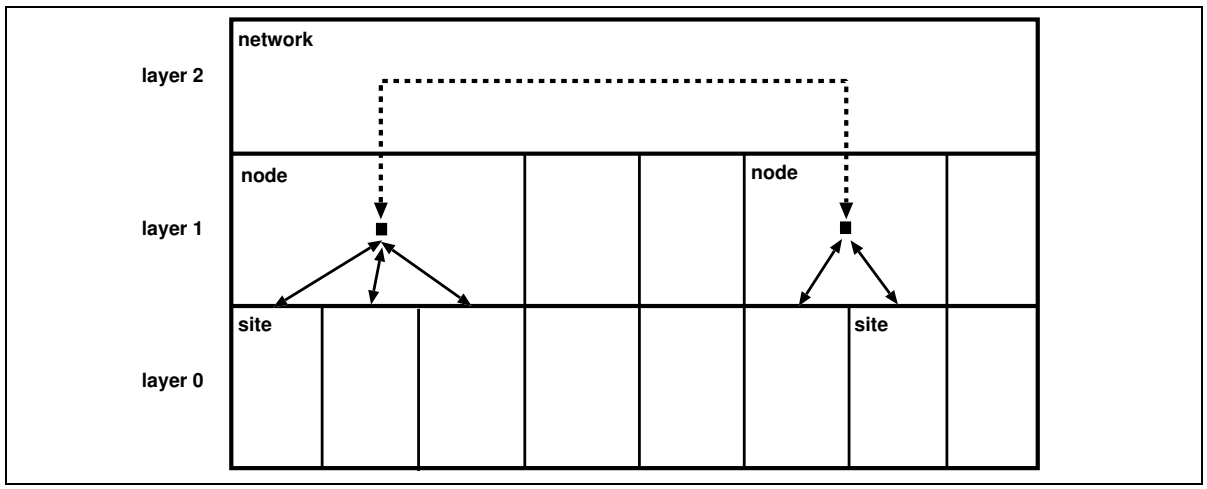

Figure 3: The architecture of DiTyCO

7.2.2 Processor Abstraction

The TyCO Virtual Machine is currently single threaded, although a multithreaded implementation is underway. The memory architecture is divided into four main areas (fig. 4):

- program: keeps the byte-code for the program to be executed. The code is composed of a sequence of instruction blocks and method tables for objects;
- heap: is used to allocate run-time data-structures. Space is allocated in blocks of contiguous machine words called frames. The virtual machine manipulates three basic kinds of frames at runtime: messages, objects, definitions and channels;
- stack: is used to keep local variables and temporary results, namely for primitive datatypes;
- run-queue: is used to keep a set of ready to run instances of process definitions or method calls.

Data is moved between the heap and the stack while running programs. The size of the heap and stack are parameters of the machine. The virtual machine uses a small set of global registers to control the main data-structures, namely the heap and run-queue:

- register hp points to the next available position in the heap;
- registers eq and sq keep the boundaries of the run-queue;
- register pc points to the next instruction to be executed in the byte-code;
- register sp points to the top of the operand stack;
- register fp points to the current environment frame, where the bindings for the local variables are located.

7.2.3 Basic Objects

The virtual machine manipulates four basic entities:

- *channels* are the means via which processes interact;
- *definitions* are template processes that may be instantiated to allow for recursion;
- *objects* are collections of methods, selectable by labels, located at some channel;
- *asynchronous messages* are method invocations for objects, located at some channel.

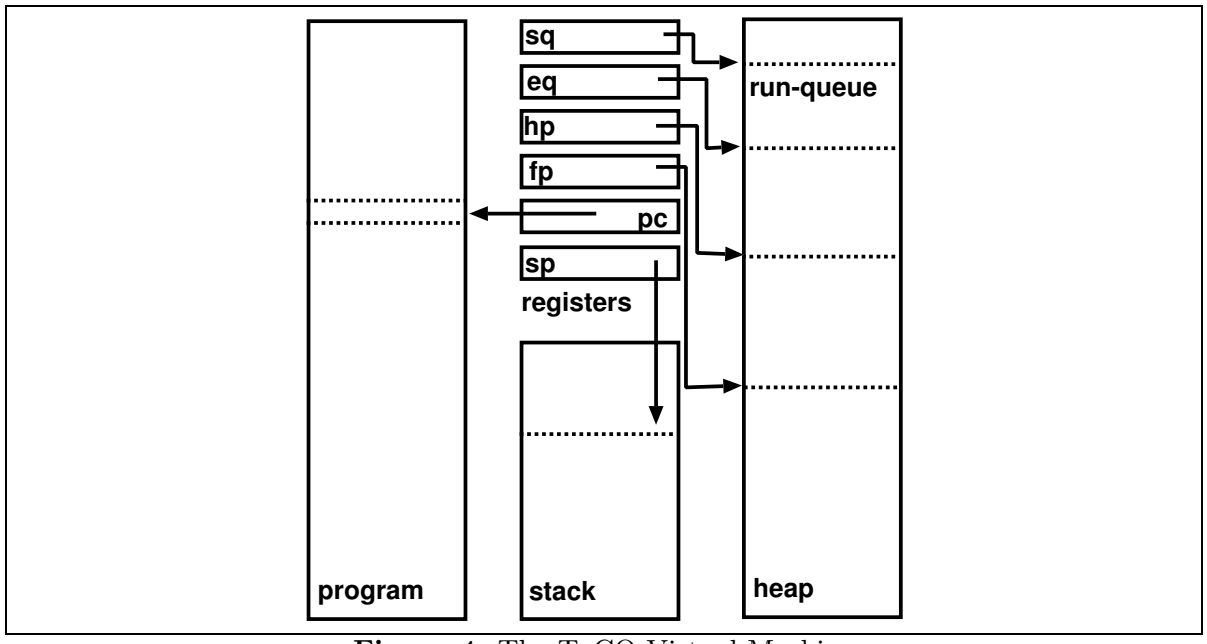

Figure 4: The TyCO Virtual Machine

7.2.4 Communication Primitives

Remote communication between sites does not exist at the programming level. Resources move to the sites according to their lexical scope. Thus all communication in DiTyCO is performed locally, at the site level. This involves the reduction of matching object-message pairs. This process is transparent to the programmer and implemented as byte-code instructions.

Remote communication does exist between sites, mediated by the node layer, to support name resolution and resource migration over the network.

A DiTyCO site is simply a TyCOVM, extended with an interface that allows interaction with the node layer (fig. 5). Such interaction is quite simple involving just the sending and receiving of closures: pieces of byte-code and their associated free variable bindings.

7.2.5 Execution Environment

In a local setting, the heap of the virtual machine holds instances of four main, dynamically allocated, data-structures:

- channels are shared communication queues. Messages and objects are placed in these queues while reduction is not possible;
- definitions are frames that hold a reference to the byte-code for the process definition and the bindings for the free variables in the body of the definition;
- messages are frames that hold the label of a method being invoked plus the bindings for the arguments of the call;
- objects are frames that hold a reference to a method table plus the bindings for the free variables in the body of the object.

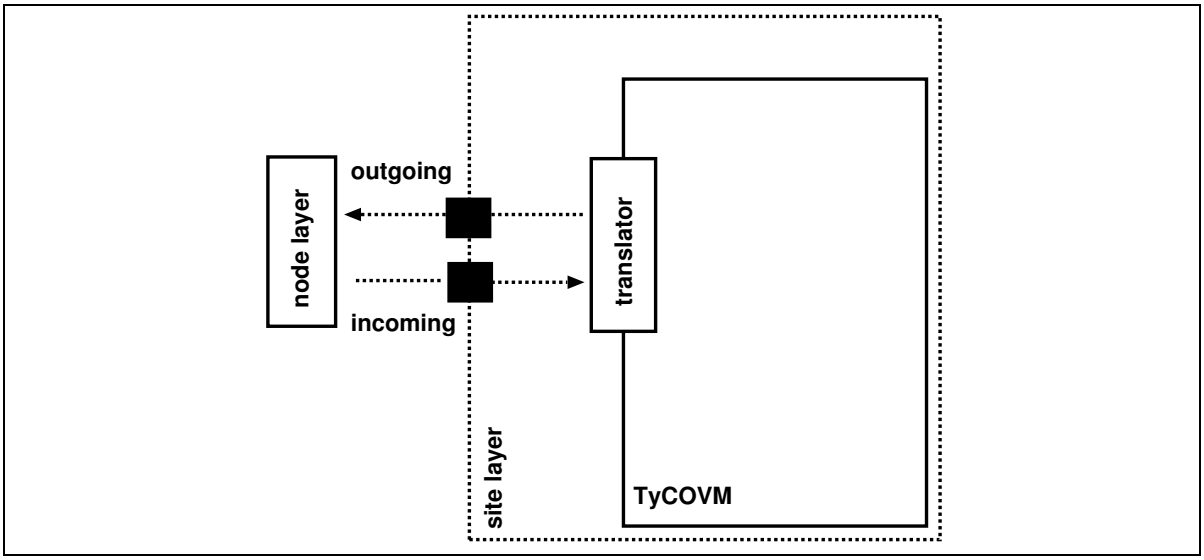

Figure 5: The Extended TyCO Virtual Machine

In a distributed setting with mobility, we add a few more data-structures to the run-time. First, sites are not explicitly programmed in DiTyCO. They are viewed implicitly as program modules with a given external interface that is published to the network. They have no corresponding data-structure but are otherwise understood as individual extended TyCOVMs.

Closures of messages, objects and definitions may be sent over the logical network to interact with the processes running at a remote site. The closures are built by packing the following information:

- an object closure packs the bindings for the free variables of the body of the object, plus the complete byte-code for the object, including method table and the byte-code for each of the methods. This imposes some discipline in the way the DiTyCO byte-code is structured to allow for fast packing;
- a message closure packs the bindings for the arguments and the label for the method to be invoked at the remote site. As messages are asynchronous, no byte-code is moved;
- finally, a definition closure packs the byte-code for the body of the definition, plus the bindings for the free variables. Again, the byte-code for definitions is structured to allow the fast packing of one or maybe several, mutually recursive, definitions.

Since channels are lexically bound, they are represented with two distinct internal formats: *network references* and *local references*, depending on whether they are located at the home site or at a remote site (see bellow). A mechanism for translating between formats is required and is provided by translation tables. There is one such table per site and they map network references into local references.

The translation occurs in two steps, in a migration:

- when a closure *arrives* at a site the bindings in the closure for local channels are transformed from a network format into local heap references, by consulting the table;

- when a closure *leaves* a site, the references to local channels in it are translated into unique network references and added to the translation table of the home site.

The only bindings affected are those associated with the *origin* and *destination* of the migration.

7.2.6 Memory Management

The virtual machine performs two kinds of garbage collections:

- a shallow garbage collection if the available space between the highest memory address and the sq address is enough to continue. In this case a fast shift of the run-queue is all that is required;
- a full garbage collection is performed otherwise.

For the full garbage collection the machine uses a copying algorithm. The garbage collector copies active frames from the current heap to a new new heap. Active frames are those that can still be used in future computations within the program. At the boundaries of instruction blocks, the future run-time behavior of the program depends only on the contents of the run-queue. This is because a given frame is active and may be used in the future only if there is a direct reference to it from the run-queue, or if it is referenced by another active frame. The garbage collector then recursively copies the trees of frames whose roots are the entries in the run-queue.

7.3 Distribution

7.3.1 Naming and Identification

In DiTyCO there are two kinds of names: sites and channels. Channels are lexically scoped, being bound to the site they are created in. Sites identify program components uniquely over the network.

Both sites and site interfaces (exported channels) are registered at a Network Name Service provided by the network layer of the implementation. The service provides the following information:

- the current IP location of a given site. This is updated dynamically;
- the current IP, the site and a unique identifier associated with a given exported channel.

Channels being referenced at a site may be lexically bound to other sites or local. Thus, the extended TyCOVM must work with two kinds of references:

- local references that are simply heap references;
- network references that are triples: (IP, site, identifier), describing a channel in the heap of another site.

Dynamic translation of network references is required whenever a closure arrives at a site. Network references for channels in the current site are replaced by local references. Likewise, when a closure leaves a site, local references are replaced by corresponding network references. All other bindings are kept the same.

7.3.2 Communication Mechanism

Local communication, within a site, is implemented with virtual machine instructions and is transparent to the user. Remote communication does not occur at the programming level and is transparently provided to allow migration between sites. The infra-structure that supports this service is implemented using standard TCP/IP sockets.

7.3.3 Fault Tolerance

Not addressed.

7.4 Mobility

7.4.1 Execution unit

In DiTyCO there are three kinds of program units that may be migrated between sites: asynchronous messages, objects and process definitions. All that is moved is some byte-code and some associated bindings. Some action by the programmer at the destination site is required for the migrating units to run.

7.4.2 Type of Mobility

DiTyCO supports only *weak mobility*, i.e., only byte-code and data migrate. For messages and objects, migration is *subjective* since they migrate in response to the lexical binding of their destination channel (code shipping). For process definitions, however, migration is *objective* as the byte-code is fetched from a site in response to an external request from a client site.

7.4.3 Hosting Structure

Since the migrating units are simply closures, upon their arrival to the destination site they are placed in the heap area of the process running the TyCOVM, where they may be triggered by some program instruction. Thus, at this level, sites (implemented as extended TyCOVMs) constitute the hosting structure.

At lower-level, however, the migrating closures are routed between sites with the help of the underlying node layer (fig. 6). The node layer receives closures from another IP address and immediately adds them to a queue the extended TyCOVM is listenning to. The complementary action is taken by the node layer when a closure is being sent over to a site elsewhere in the network. A closure is extracted from a queue of outgoing messages, shared with the extended TyCOVM. These queues are also used by sites to make requests to the name resolution service provided by the network layer, through the node layer.

7.4.4 Resource Management

Resources are lexically bound to their home site. They are otherwise accessible from anywhere in the network, if their associated channel is known.

7.4.5 Quality of Service

Not addressed.

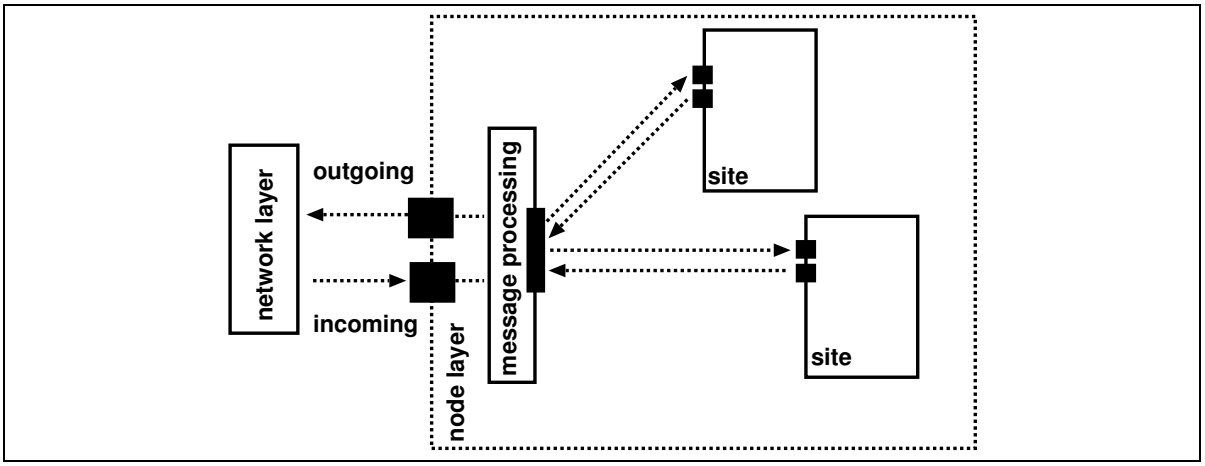

Figure 6: A DiTyCO Node

7.5 Security

7.5.1 Execution Errors

Communication is always local, that is, within sites. Strong, static typing is used to ensure that the types for interacting processes at distinct sites match, thus eliminating run-time protocol errors.

7.5.2 Identification and Authentication

Not addressed.

7.5.3 Access Rights

Not addressed.

7.5.4 Communication Security

Not addressed.

7.6 Proof of Correctness

The TyCO Virtual Machine has been proved sound relative to the underlying process calculus, TyCO. No such result is presently available for the extended virtual machine with respect to the $LSD\pi$ calculus or its straightforward extension to TyCO.

7.7 Simple Case Study

This small case study is inspired in the SETI (Search for Extra-Terrestrial Intelligence) research project. Seti@home was developed by the SETI research team as a way to deal with the vast computational power required to process data obtained by the project's radio-telescopes. Here we model an application that only requires a single command from a remote client to be downloaded from the SETI server site and run *forever* at the client site processing data

chunks from SETI's database. The code for both seti and client sites is given below. Note that, the process **let** $x = a\,!\,l[\tilde{v}]$ **in** P abbreviates **new** r $a\,!\,l[\tilde{v}r] \mid r?(x) = P$, thus hidding the reply-to channel r. It describes a method invocation, implemented via two (asynchronous) remote messages. The particular case where the channel r is used only for synchronization is written more succintely as: $a\,!\,l[\tilde{v}]; P$. The code for the client and the server is as follows:

SETI Site	**Client Site**

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

The code in the client just instantiates a copy of the Install class at the seti server site. Since Install is lexical bound to the seti site, this triggers the migration of a copy of the code for Install to the client site. Free identifiers in the code get translated to expose the lexical binding. This is represented by the application of the map $\sigma_{\mathsf{seti\ client}}$ to the body of the definition. Thus, after the migration we have the following situation at the client:

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

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

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

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

8 Safe Mobile Ambients

SAM is a distributed abstract machine, implementating an ambient-like calculus. The Ambient calculus is a model for mobile distributed computing. An ambient is the unit of movement. Processes within the same ambient may exchange messages; ambients may be nested, so to form a hierarchical structure. The three primitives for movement allow: an ambient to enter another ambient, $n[\,\texttt{in}\,m.P|Q\,]|m[\,R\,] \longrightarrow m[\,n[\,P|Q\,]|R\,]$; an ambient to exit another ambient, $m[\,n[\,\texttt{out}\,m.P|Q\,]|R\,] \longrightarrow n[\,P|Q\,]|m[\,R\,]$; a process to dissolve an ambient boundary thus obtaining access to its content, $\texttt{open}\,n.P|n[\,Q\,] \longrightarrow P|Q$.

The calculus that we actually implement is *typed Safe Ambients* rather than untyped Mobile Ambients. SA is a variant of the original calculus that eliminates certain forms of interference in ambients, ie grave interferences. They are produced when an ambient tries to perform two different movement operations at the same time, as for instance $n[\,\texttt{in}\,h.P|\texttt{out}\,n.Q|R\,]$. The control of mobility is obtained in SA by a modification of the syntax and a type system; the absence of grave interferences brings benefits both to proves and to the implementation.

One of the difficulties of a distributed implementation of an ambient-like language is that each movement operation involves ambients on different hierarchical levels (eg. out operation need to relocate an ambient outside its current parent and place it in its grandparent). One should however bear in mind that, in an ambient-like formalism, entering and exiting ambients is not very useful without opening some ambients. The processes local to an ambient a can interact with the processes local to ambient b only if one of the ambients moves inside the other *and* is opened. For these reasons in our abstract machine we separate between the logical structure of an ambient system and its physical distribution. Exploiting this, the interpretation of the movement associated to the capabilities is *reversed* (wrt Mobile Ambients): the movement of the `open` capability is physical, that is, the location of some processes changes, whereas that of `in` and `out` is only logical, meaning that some hierarchical dependencies among ambients may change, but not their physical displacement: intuitively, `in` and `out` became in SAM acquisition of access rights, while `open` is an exercise of them. We may say that the movement given by `in` and `out` is first-order (some names move, and some pointers are updated), whereas the movement given by `open` is higher order, ie some terms move.

8.1 Main References

- M.Bugliesi and G.Castagna. Secure safe ambients. In *Proc. 28th POPL*. ACM Press, 2001.
- L. Cardelli and A.D. Gordon. Mobile ambients. *Proc. FoSSaCS '98*, LNCS 1378, pages 140–155. Springer Verlag, 1998.
- L.Cardelli and A.D. Gordon. Types for mobile ambients. In *Proc. 26th POPL*, pages 79–92. ACM Press, 1999.
- C.Fournet, J.-J. Lévy, and A.Schmitt. An asynchronous distributed implementation fo mobile ambients. *IFIP TCS2000*, LNCS 1872, pages 348–364. Springer Verlag, 2000.
- F.Levi and D.Sangiorgi. Controlling interference in ambients. In *Proc. 27th POPL*. ACM Press, 2000.

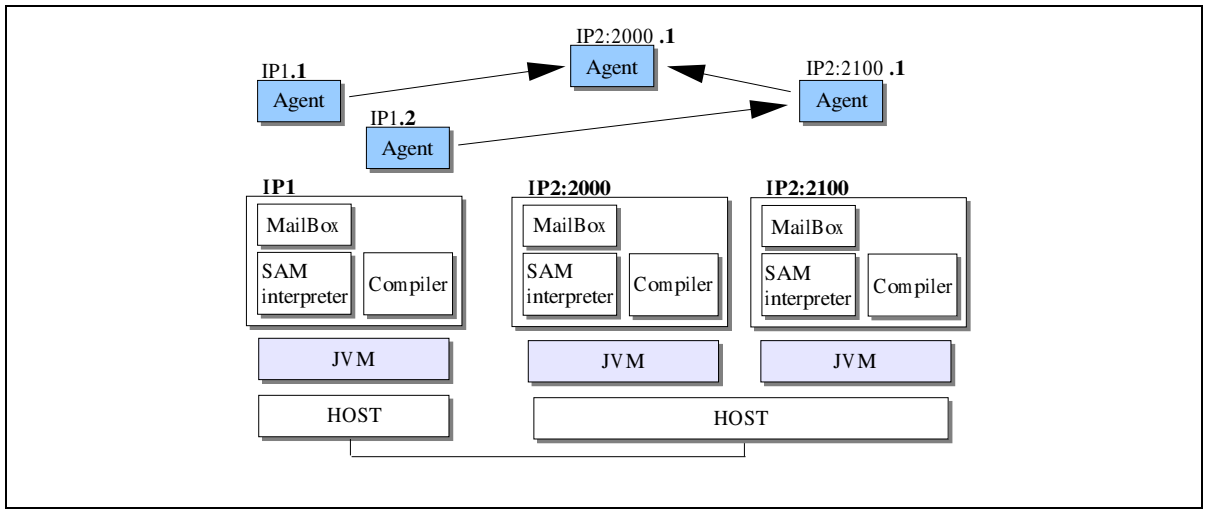

Figure 7: SAM architecture layers

- Andrea Valente, Davide Sangiorgi. A distributed abstract machine for Safe Ambients. CSCS-ICALP 2001, LNCS 2076,pages 408-420, july 2001.

8.2 Execution Structure

8.2.1 General Architecture

SAM stands for Safe ambient Abstract Machine. Our implementation, written in Java following the definition of SAM, allows clustering of agents on the same IP node (ie a physical machine). Therefore the implementation is made of three layers: agents, nodes and the network (Figure 7).

The address k of an agent is composed of the *IP-name* of the node on which its resides, plus a suffix, which is different for each agent in that node: each agent has a unique location name.

Agent are executed by the node (the SAM interpreter) and the processes local to an agent are scheduled using a round-robin policy. Each agent knows its name, its address, its parent's address, and keeps a link to its node. Agents operate by exchanging messages, and from a physical point of view, the messages are of two kinds: local, when both agents reside on the same node, and remote, when two distinct nodes are involved. In the implementation however every message is sent from the sender agent to its local node, as if it the reciever agent was local; its the node that knows (by cheching the IP address of the reciever against its own) whether to keep the message in the local mailbox or forward it to a remote node's one.

Each node is special Java *RMI object*, with its own thread of execution, takes care of inter-node communications. For this, nodes act alternatively as clients (requiring that a message is sent to another computer) and as servers (receiving a message and pushing it into a local mail-box): they form a p2p communitiy.

The node layer is implemented using Java RMI and *serialization*, and the network layer simply provides IP-name registry for RMI communications to take place (using Java *RMIregistry*).

8.2.2 Processor Abstraction

Our abstract machine is implemented as an interpreter running on top of the Java Virtual Machine; this means that from our interpreter the only *hardware* visible is the JVM itself. Threads and resources are accessible via the primitives of the JVM.

8.2.3 Basic Objects

processes, objects, threads, sites, address spaces, patterns...

A distributed system is expressed in SAM by a *net*, ie a parallel composition of *agents* and *messages*, with some names possibly restricted.

An agent can be a *located ambient* or a *forwarder*. Located ambients are the basic unit of SAM, and represent SA ambients running at a specific physical address, and their (internal) local processes; a located ambient becomes a forwarder when opened. About messages, they can be of two forms: *requests* and *completions*. The former are the messages that agents send to their parent to request a movement operation; the latter are the messages that the parent sends back to its children to complete the operation.

Processes inside located ambients is mostly identical to processes in SA. The only additions are: a special prefix $No.P$, which appears in an agent when it has sent a request to its parent but has not received the answer yet; and the *requests*, which represent messages received from the children and not yet served, ie a local queue of pending messages is kept inside every agent.

8.2.4 Communication Primitives

The Safe ambient Abstract Machine works on asynchronous messages, all the three basic operations in the SA calculus (ie in,out and open moves) are simulated by protocols of asynchronous messages.

Local communications can also take place inside an agent.

8.2.5 Execution Environment

Since we followed the patter *interpreter* to code SAM in Java, every term of the original calculus is represented inside the JVM by a tree of objects (one for each subterm of the complete term). Our compiler compiles a typed SA expression (ie a program) into such a tree, then serialize it on a stream; from there the *executable* can be retrieved by the interpreter and executed.

8.2.6 Memory Management

All the terms being trees of Java objects, are subject to the standard memory management (ie the gc) provided by the local JVM.

Inside a SAM node however, open operations can generate structures that should need a distributed gc to be removed. According with our *flat mobility* policy (subambients do not move), an *opened* agent will disappear and be replaced by a *forwarder*, a special kind of agent with the form:

$h : k\,!$

whose intended meaning is that every message arriving at h from any of its children, is sent to k. Agents of this kind don't contain running processes, nor subambients, so they absorb very little memory and computational power, but there is no way, up to now, to remove them from the system once they are created.

8.3 Distribution

8.3.1 Naming and Identification

SAM separates between the logical and the physical distribution of ambients. The logical distribution is given by the tree structure of the ambient syntax.

The physical distribution is given by the association of a location to each running ambient.

In SAM, an ambient named n is represented as a *located ambient* $h{:}\, n[\, P\,]_k$, where h is the location at which the ambient runs, k is the location of the parent ambient, and P collects the processes local to the ambient. While the same name may be assigned to several ambients, a location univocally identifies an agent: it can be thought of as its physical address. Infact the address k is composed of the *IP-name* of the node on which this agent resides, plus a suffix, which is different (progressive) for each agent in that node.

8.3.2 Communication Mechanism

Since agents may run at different physical sites, they communicate with each other by means of *asynchronous* messages.

Actions as `in` and `out` modify the logical distribution, but only `open` can change the physical distribution.

For instance, a SA term $n[\, P_1|P_2|m_1[\, Q_1\,]|m_2[\, Q_2\,]\,]$, where P_1 and P_2 are the local processes of n, and Q_i $(i = 1, 2)$ is a local process of m_i (i.e., m_i has no subambients), becomes in SAM:

$$h{:}\, n[\, P_1|P_2\,]_{\texttt{root}} \parallel k_1{:}\, m_1[\, Q_1\,]_h \parallel k_2{:}\, m_2[\, Q_2\,]_h$$

where h, k_1, k_2 are different location names, `root` is a special name indicating the outermost location, and $\parallel$ is parallel composition of located ambients. Note the above configuration is actually obtained after two creation steps, in which the root ambient spawns off the two ambients located at k_1 and k_2.

8.3.3 Fault Tolerance

This issue is outside the scope of SA, up to now.

8.4 Mobility

8.4.1 Execution unit

A located ambient is the unit of execution and movement, ie what we call *an agent* in SAM. Processes within the same agent may exchange (local) messages; agents keep parent-links to model the hierarchical structure of the original SA ambients.

8.4.2 Type of Mobility

In SAM we keep the strong mobility approach common to all ambient-based calculi.

8.4.3 Hosting Structure

In SAM an agent starts and ends its life on the same physical location. Only opening an agent (ie destroying it) its contents can migrate to another host. Since processes are expressed as Java object trees, the SAM intrpreter simply stop executing a process, serialize it on another node, and resume it, merging it to the contents of an agent, to be executed.

Nodes are the hosting structure of the SAM distributed machine, providing just a mail box for local agents to dialog and scheduling all agents on the local JVM. Nodes form a p2p community and can exchange processes, start new agents, and when a message is directed to a remote agent, is the node local to the sender that places it in the remote mail-box (controlled by a remote node).

8.4.4 Resource Management

Since SAM is just the implementation of a calculus, no other resources than the basic constructs of SA are available to the programmer: this means that no local (nor remote) resource other than SAM agents, are accessible from inside a SAM program.

8.4.5 Quality of Service

Still to be studied for SA/SAM.

8.5 Security

Security is usually understud as an orthogonal issue for ambient-based calculi.

Adding encription at the level of our hosting structure (nodes) should be quite direct and with hardly any visible effect for the SAM programmer. Up to now we don't cover this subject in SAM.

8.5.1 Execution Errors

Since we base our abstract machine on typed SA, its the type system itself that rules out run-time errors: typed SA have no grave interferences. Typed agents are well-behaved in SAM.

8.5.2 Identification and Authentication

It is name-based (as in SA).

8.5.3 Access Rights

Access rights are capability-based in SA.

8.5.4 Communication Security

Cryptography can be introduced for any message exchange. Again this are extensions and whould affect only the host level, not the SAM interpreter itself.

8.6 Proof of Correctness

A proof of correctness for SAM wrt typed SA can be found in the paper "A distributed abstract machine for Safe Ambients." (see section 8.1).

8.7 Simple Case Study

Suppose we want to model a traveller that starts on a base site, goes to distant server, interacts with its services, and then comes back reporting a result. For simplicity, we suppose that interaction just consists in loading a value v that is also the final result. The distant server is defined thus:

$$SERVER \stackrel{\text{def}}{=} s[\,\overline{\texttt{in}}\,s.\texttt{open}\,go.\overline{\texttt{out}}\,s.P|v\,!\,]$$

The base site and the traveller T are:

$$\begin{array}{rcl} BASE & \stackrel{\text{def}}{=} & n[\,T|\overline{\texttt{out}}\,n.\overline{\texttt{in}}\,n.\texttt{open}\,return.(x)R\,] \\ T & \stackrel{\text{def}}{=} & go[\,\texttt{out}\,n.\texttt{in}\,s.\overline{\texttt{open}}\,go.(x)return[\,\texttt{out}\,s.\texttt{in}\,n.\overline{\texttt{open}}\,return.x\,!\,]\,] \end{array}$$

We have:

$$\begin{array}{rcll} SERVER|BASE & \Longrightarrow & s[\,\overline{\texttt{out}}\,s.P|v\,!\,| & \\ & & (x)return[\,\texttt{out}\,s.\texttt{in}\,n.\overline{\texttt{open}}\,return.x\,!\,]\,]| & \\ & & n[\,\texttt{in}\,n.\texttt{open}\,return.(x)R\,] & (1) \\ & \Longrightarrow & s[\,P\,]|n[\,(x)R|v\,!\,] & \\ & \longrightarrow & s[\,P\,]|n[\,R\{v/x\}\,] & \end{array}$$

In the abstract machine execution, the initial configuration is (omitting some local processes of s and n)

$$h{:}\,s[\,\ldots\,]_{\texttt{root}} \parallel k{:}\,n[\,T|\,\ldots\,]_{\texttt{root}}$$

and the first step is to spawn off a new location hosting the traveller, obtaining:

$$h{:}\,s[\,\ldots\,]_{\texttt{root}} \parallel k{:}\,n[\,\ldots\,]_{\texttt{root}} \parallel \ell{:}\,go[\,\ldots\,]_k$$

In the reductions in 1, two *open* are executed. When this is modeled in SAM, in the first *open* the (derivative of the) traveller physically moves into the distant server so to load v; in the second, the (derivative of the) traveller comes back to deliver its result to R.

9 Jcl and JoCaml

This section shortly reviews the two main implementations of the JOIN-calculus, JCL (Join Calculus Language) and JOCAML, which present similar architectures, though the implementations differ, since JCL was designed from the start as a distributed run-time, while JOCAML is an extension of an existing programming language (Objective Caml) with primitives for distributed programming. In the sequel, we shall only distinguish between the two languages when necessary.

Those implementations aim to provide the programmer with primitives allowing transparent communication and migration, while keeping predictable program behaviors, relying on the strong semantics of the underlying calculus.

- JCL comes in the form of a compiler, linker, and interpreter (VM) written in Objective Caml using as a source language the JOIN-calculus with basic data types. Most of the functions provided by the Objective Caml standard library are built-in primitives, to which can be added any Objective Caml module using a simple interface.

- JOCAML is an extension of the Objective Caml language, which includes the JOIN-calculus primitives as a library.

9.1 Main References

- Conchon, S., and F. Le Fessant, "JoCaml: Mobile Agents for Objective-Caml". In *Symposium on Agent Systems and Applications, Mobile Agents (ASA/MA'99)* (1999), 22–29, IEEE Computer Society.

- Fournet, C., and G. Gonthier, "The Reflexive CHAM and the Join-Calculus". In *Proceedings 23rd ACM Symposium on Principles of Programming Languages (POPL'96)* (1996), 372–385, ACM Press.

- Fournet, C., G. Gonthier, J.-J. Levy, L. Maranget, and D. Remy, "A Calculus of Mobile Agents". In *Proceedings 7th International Conference on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science **1119** (1996), Springer-Verlag.

- Fournet, C., and L. Maranget, "The Join-Calculus Language". Available electronically at `http://pauillac.inria.fr/join`, 1997.

- Fournet, C., "The Join-Calculus: a Calculus for Distributed Mobile Programming". Ph.D. thesis, Ecole Polytechnique, 1998.

- Le Fessant, F., and L. Maranget, "Compiling Join-Patterns". In *Proceedings 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, Electronic Notes in Theoretical Computer Science **16 (3)** (2000), Elsevier.

- Le Fessant, F., "The JoCaml System Prototype". Software and documentation available from `http://join.inria.fr/jocaml`, 1998.

- Levy, J.-J., "Some Results in the Join-Calculus". In *Proceedings 3rd International Symposium on Theoretical Aspects of Computer Software (TACS'97)*, Lecture Notes in Computer Science **1281** (1997), Springer-Verlag.

- Peskine, G., "JAM : the JoCaml Abstract Machine". Private Communication, 2002.

9.2 Execution Structure

9.2.1 General Architecture

The VM is fairly standard and allows the execution of bytecode threads. The VM memory can be divided into three main segments (figure 8):

1. A *code segment* containing the bytecode of the program to be run.
2. A *stack segment* containing the stack of the current thread.
3. A *heap* containing various data structures used in the language such as channel definitions, locations, a queue of runnable threads, or a table of global values ...

9.2.2 Processor Abstraction

A JOIN-calculus process P is represented at system-level using a *code closure* (containing the set of free variables in P, and a pointer towards the code of P) and implemented using threads which can be executed concurrently. At each instant, a single thread is running, the others being either in a runnable state, waiting for execution in the run queue, or blocked, waiting to be awakened.

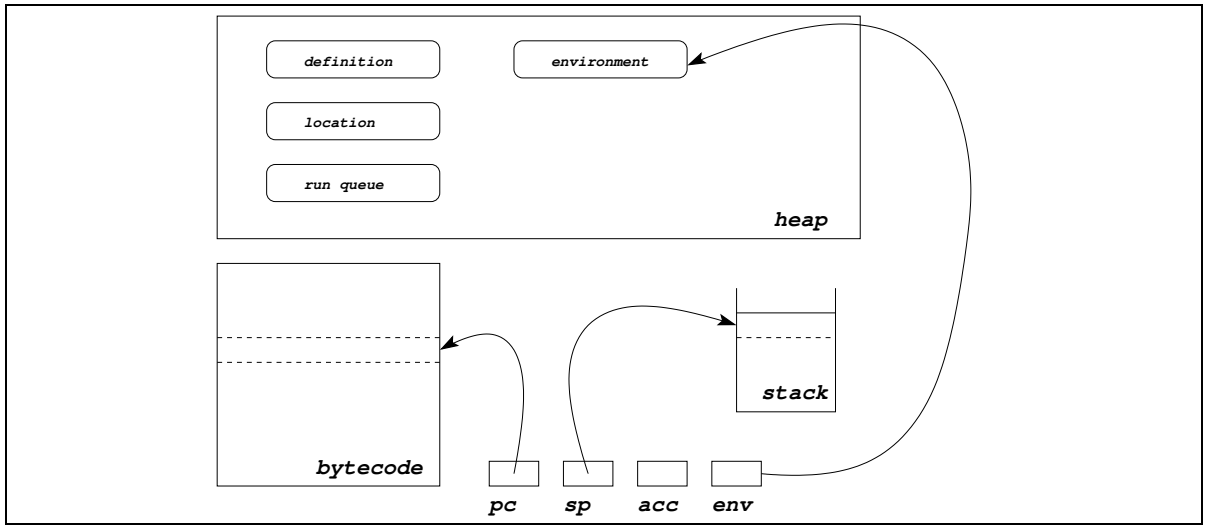

Figure 8: Structure of the JCL run-time

Four registers allow to manipulate the current thread:

- The *code pointer* `pc` refers to the next value in the code segment to interpret, which can be the next instruction to run, or be used as an argument of the current instruction.
- The *stack register* `sp` is a pointer to the top of the stack of the current thread.
- The *environment register* `env` contains the address of the current code closure, i.e., of the free variables of the code currently run.

- The *accumulator* is used to compute values.

A description of JoCaml as an abstract machine is described in [Pes02].

9.2.3 Basic Objects

The main execution structures of the VM are the following:

- *Definitions* are the main data structure, and allow to simultaneously define several *channels* (communication links between agents), along with synchronization constraints (*join-patterns*). Pattern matching is implemented using automata that will fire transitions upon message reception. Definitions can be seen as closures with several entry points corresponding to the code of the automaton associated with each channel [FM00]. In Jcl, definitions are implemented as vector structures (figure 9): the first slot holds a header containing the definition status and a pointer towards the location of the definition. The following slots correspond to the channel entry points (header and pointer to the channel automaton). Finally, one finds pointers to channel queues of pending messages, and the set of free variables in the definition. Message sending on channel x is similar to a function call: message arguments are stacked, and control is transferred to the automaton code for x. That code will either store the message in the channel queue for x, or fire a guarded process. In JoCaml, the implementation of definitions is still structured as a vector but does not use automata for channels, which are represented using a pointer to a definition and an index. Definition activability status is checked and updated at run-time, whenever a new message arrives.

- *Threads* represent active computations contained within locations, and can be identified with their stacks. A scheduler ensures time-sharing between runnable threads in the VM run queue.

- *Locations* are defined as units of mobility and failure, and can be seen as a logical space organized into a tree structure, and hosting a number of channel definitions and threads. Locations are implemented by an address (containing the address of the site of creation of the location), and a set of pointers towards the contained objects, i.e., definitions and threads (figure 10). The concept of location both captures physical and logical distribution, since a physical site can be identified with the top-level location in the location tree.

- *The tables of primitives and global values*: each program is made of a set of compilation units called *modules*, each defining an interface and exporting a set of values through that interface. This interface is implemented using a table called `global_value`. The system also maintains a global table to allow programs to access external code (Objective Caml library for Jcl, C functions for JoCaml).

9.2.4 Communication Primitives

Communication occurs using asynchronous message passing on channel names. Local communication is implemented using function calls (see "Communication Mechanism" section).

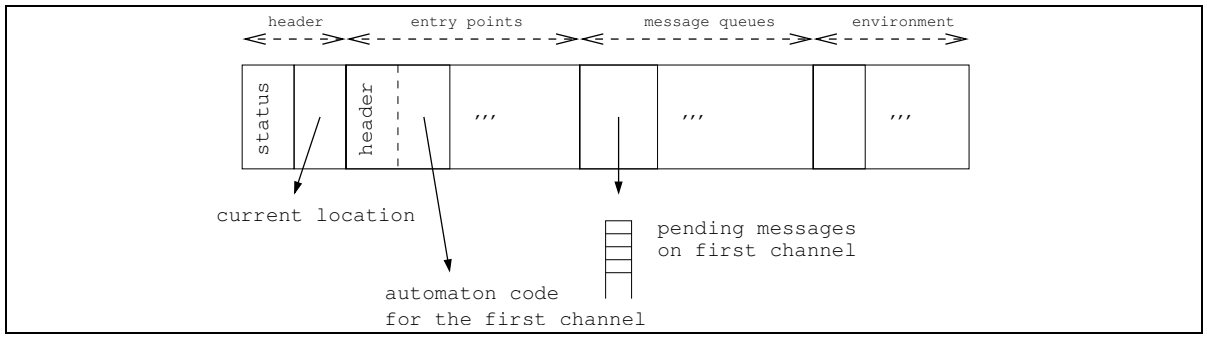

Figure 9: Structure of a definition in JCL

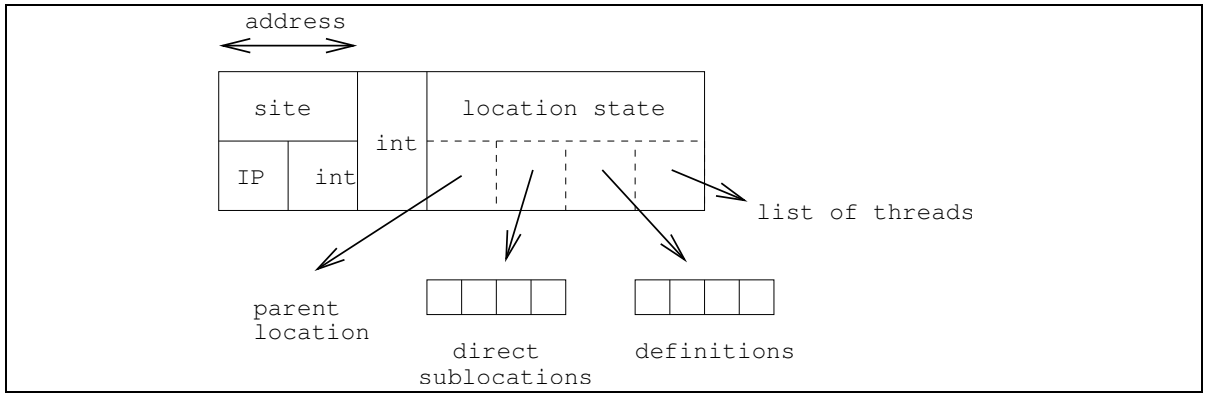

Figure 10: Structure of a location in JCL

9.2.5 Execution Environment

The JCL machine is based on a bytecode interpreter which allows to execute a program independantly of the underlying host. From the VM, access to the thread state is possible, which makes the implementation of strong migration easier.

9.2.6 Memory Management

Each VM locally benefits from the memory management of Objective Caml. JCL does not implement any garbage collection mechanism. However, JOCAML provides a garbage collector for chains of remote references, with an extension allowing to collect cycles between different run-times.

9.3 Distribution

9.3.1 Naming and Identification

At system level, entities from the calculus (variables, channels, or locations) are designated using some notion of *reference*, which stands for a name, and can be created using the `let` (`let def` in JOCAML) and `loc` (`let loc` in JOCAML) primitives. The name space for references is global. As references are unique (for channel or location names), so is their scope. Note that for references, a local scope can become distributed using scope extrusion rules. However, for the primitives of Objective Caml, scope is restricted to the site of execution.

Values are potentially transparent to localization. References exhibit a *unicast* semantics, i.e., they allow to designate a single object. However, since a location contains a cluster of locations and that such a cluster can be detected and stopped using a single reference, the reference semantics is also potentially multicast.

Naming facilities include a service on each physical site to create fresh names, and a mechanism to share names between sites (naming service). The naming service is a full part of the system, and allows to register or lookup a given name, which can reference any object of the language: channel, location, ... The naming service is centralized in JCL and can be possibly distributed in JoCaml.

9.3.2 Communication Mechanism

Channel are "asymmetric links": on a given channel, a process can only emit, message reception occuring only in a receiver process which was defined simultaneously with the channel (guarded process). Message sending can be synchronous (the process waits for a reply), or asynchronous (emission is non-blocking). Each message sent will be received exactly once.

Two separate mechanisms are used according to whether communication is local or remote. In both cases, communication is fully transparent for the user who cannot distinguish between local and remote channels. Local interaction is implemented as function calls. Remote message send introduces an extra stage for message transmission between sites based on TCP/IP and Unix sockets. In JoCaml, stubs and surrogates according to the SSP (Stub-Scion Pairs) Chains model are automatically created to ensure a transparent distributed communication.

The only synchronization mechanism provided by the language is found in definitions, where the join-patterns allow to express synchronization schemes between channels, and to represent data structures which can be used concurrently (semaphores, ...). As channel names are unique, synchronization is purely local which avoids the use of a distributed consensus algorithm.

9.3.3 Fault Tolerance

In JoCaml, locations are units of failure, which follow a fail-stop failure model. Failures can be provoked using the `fail` primitive of the calculus. The use of SSP Chains for remote pointers ensures to a certain extent fault-tolerant remote communications.

9.4 Mobility

Locations are the only units of mobility in the system, in a code mobility sense. Though, a notion of mobility of names, in the π-calculus fashion, is also present: channels are first class values, and thus can be sent as arguments of messages. This has the consequence of extending the scope of the channel name, which can eventually become distributed.

9.4.1 Execution Unit and Hosting Structure

The location both represents a notion of physical site, of mobile code, and of mobile agent according to whether the location is empty (it then serves as a destination address), only contains channel definitions (representing the code of a program), or also contains threads.

9.4.2 Type of Mobility

Access to the stack part of a thread allows JCL and JOCAML to support strong migration (migration of threads), impossible to achieve properly in Java: thus, thread execution can resume directly at the point where it was interrupted due to the migration request. The language allows to express a subjective form of mobility for the execution units: the decision to migrate comes from the inside of the execution unit using a `go` primitive, and not from the outside (objective form of mobility).

JCL and JOCAML allow three types of code mobility:

- *Function shipping (Remote Evaluation)* is possible since the client can transmit to the server a location only containing a set of definitions representing the program to execute. The server can then send back the result using a simple message.
- *Function fetching (Dynamic Download)* occurs when, upon client demand, the server forces the migration of a location containing the program to execute.
- *Mobile agents* can also be programmed when locations contain active code. Thus execution units can migrate from hosting structure to hosting structure.

9.4.3 Resource Management

The references defined by the language are network transparent. However, the references for the Objective Caml primitives are purely local, since each VM contains its own table of primitives.

An execution unit can contain references to external resources. The distributed lexical scoping discipline of the implementations, along with the use of references to represent locations allow an easy migration of some of those external resources.

When an execution unit migrates, message forwarders are left on the initial site to ensure the preservation of communication links during all migrations.

9.5 Security

The only security mechanism present is strong static typing, based on an inference system, which guarantees that well-typed programs will be executed without typing errors, even in a distributed setting. However, a security model is not present in the implementations, nor are access control mechanisms. Similarly, commmunication on channels are not protected, without any kind of guarantees of secrecy or integrity.

9.6 Proof of Correctness

Up to our current knowledge, a proof of correctness for the JCL/JOCAML implementations has not yet been carried out, but is mentionned as future work in [Pes02].

9.7 Simple Case Study

We reproduce here some excerpts from a simple tutorial for JOCAML taken from [CF99]:

"In the following examples, we will make different JoCaml programs work together. There are different ways to initiate the communication between these programs. [...] We use a simple program, the name-server, bound to a well-known port of a well-known machine, which can either be specified by the environment, by an URL, or found by multicasting the local network. JoCaml provides two simple functions `Ns.register` and `Ns.lookup`, to respectively register named values in the name-server, and request these values from their names. [...] A special keyword, `vartype`, has been added to provide the type information necessary for the `Ns.register` and `Ns.lookup` calls. Here is a small example, of two programs exchanging a simple string through the name-server facility:

```
(* SERVER *)
let host = Unix.gethostname ();;
Ns.register "server host" host vartype;;
(* CLIENT *)
let host = Ns.lookup "server host" vartype;;
printf "The server is on %s" host;;
```

[...] JoCaml extends the Objective Caml language with the notion of *process*. Indeed, Objective Caml only uses *expressions*, i.e., terms which are computed to a value. On the contrary, the execution of a JoCaml process does not yield any value. Instead, it is used for its side-effects, mainly for communication and synchronization tasks.

The main reason for separating expressions and processes is to force the programmer to pay attention to concurrency problems. [...] Thus, JoCaml processes are syntactically different from Objective Caml expressions. They are introduced by curly braces {...} containing both expressions and sub-processes, while | denotes parallel composition of processes. The `spawn` { P } construct is used inside OCaml expressions to create a new thread to execute the process P.

[...] JoCaml programs communicate and synchronize through messages sent on channels, called *names* in the JOIN-calculus terminology. [...] Mainly, channels are asymmetric links between one receiver and multiple senders: when messages are sent on a channel by different agents, they are all received by the same agent. This mechanism is not modified if the channel name is exchanged between agents on different machines, nor if the receiver-agent migrates: messages are simply forwarded by the network to the correct agent. A simple synchronous channel definition has the syntax:

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

which creates both the channel name *name*, which can be used to send messages, and a receiver, which will create a new thread in the agent to execute P for each message received, with *args* being the message content. Since Objective Caml is able to marshal and unmarshal any datatype, a message can contain any value, provided the program is correctly typechecked. The process must contain a reply statement to specify the returned value, since the message is synchronous (a la RPC). Asynchronous channels are defined by simply suffixing the channel name with a ! symbol. In the following example, we define a simple `print` channel, on which other agents can send messages to be printed.

```
(* SERVER *)
let def print s = { print_string s; reply};;
Ns.register "Printer" print vartype;;
(* CLIENT *)
let printer = Ns.lookup "Printer" vartype;;
printer "Hello ";;
printer "I'm the client";;
```

[...] Channel definitions also have some synchronization power, thanks to the use of *join-patterns*. Join-patterns are used to express rendez-vous between message reception on several channels. For this, several channels names, separated by | symbols, can be specified to trigger a process in a `let def` construct. Moreover, several processes can be specified for different configuration of messages on the channels.

For example, in the following definition, P may be triggered when messages are present on both u and v, and Q may be triggered when messages are present on both u and w. Since u appears in both join-patterns, there is a possible synchronization between P and Q: if there is only one u message available, P and Q threads will not be executed in parallel when messages are available on both v and w.

```
let def u(a) | v(b) = P(a,b)
or u(a) | w(b) = Q(a,b)
```

[...] We simply refine the previous example of a `print` channel, by requiring that only one `print` process is executed at any time. This is done by adding an asynchronous `lock` channel, whose unique message is consumed by the process, and released when the process is finished. In the client, a rendez-vous is used on three asynchronous messages to decide when the client can be terminated. [...]

```
(* SERVER *)
let def print! (str, cont) | lock! () =
{ print_string str; { cont () | lock () }};;
spawn { lock () };;
Ns.register "Printer" print vartype;;

(* CLIENT *)
let printer = Ns.lookup "Printer" vartype;;
let def cont1! () | cont2! () | cont3! () = { exit 0; }
spawn { printer ("Hello", cont1) |
printer ("I'm a client", cont2) |
printer ("I must do my job", cont3) }
```

[...] In JoCaml, sites, agents and groups of agents are represented by a single abstraction, the *location*. The syntax of a location definition is:

```
let loc agent
do process
```

which creates a new agent, named *agent*, executing the thread *process*. In the following example, the server defines a synchronous `print` channel, queries the name server for a location registered by the client, and then creates an agent which will migrate to that location. The agent calls twice the `Unix.gethostname` function, once before the migration, to know its current hostname, and once after the migration, to learn its new hostname. This shows the difference between JoCaml synchronous channels and OCaml functions, since agents can use local resources simply by calling local functions, whereas channels are always bound to the same receiver.

On the other hand, the agent uses twice the `print` channel, once before the migration, and once after the migration. In both cases, the message will be sent to the server, which will print the message on its terminal. This illustrates that agents can migrate without breaking their communication channels, thus keeping complete network transparency.

```
(* SERVER *)
let def print s = print_string s; reply;;
let target = Ns.lookup "client" vartype;;
let loc agent do {
print (sprintf "I'm on computer %s" (Unix.gethostname ()));
go target;
print (sprintf "I'm on computer %s" (Unix.gethostname ()));
print_string "Hello";
};;

(* CLIENT *)
let loc target do {};;
Ns.register "client" target vartype;;
```

Furthermore, agents have to communicate. However, since channels are uni- directional, agents can only send messages to others. To receive messages, the syntax of location defnition is extended with channels declarations:

```
let loc agent
def channels
do { process }
```

which creates a new agent composed of threads and channel definitions. All these definitions are lexical scoped, as in Obliq. This means that each channel declared in a location has a binding that never change, wherever this agent or the others using this name migrate. This is the transparency mechanism used in JoCaml that eases the programmer task to distribute applications. In the next example we point out the simplicity and easiness of handling agents in JoCaml. This example makes use of agents in a simple "talk" program. The server waits for two connections, and then, creates and sends a speaker agent to each client. After that, the server can be shutdown without disturbing the discussion of the clients, or can stay available for other pairs of clients.

```
(* SERVER *)
```

```
let def create_speaker(l,name) =
(* This channel is used to create a new speaker agent with two
channels:
print: to print messages on agents' display
speak: allows the client to enter a new sentence *)
let loc agent def (* A new speaker agent *)
print! (s,send) = { printf "%s\n" s; speak(send) }
and speak! send = { printf "%s>" name;
let s = read_line () in
send(s,send) }
do { go l; } (* Agent migrates to the location l *)
in (* The "interface" to the agent is return to the server *)
{ reply print,speak };;
(* The connexion part of the server *)
let def talk!(l1,n1) | no_client!() = one_client(l1,n1)
or talk!(l2,n2) | one_client!(l1,n1) =
{ let print1,speak1 = create_speaker(l1,n1) in (* creation of *)
let print2,speak2 = create_speaker(l2,n2) in (* two agents *)
speak1(print2) | no_client ()};;
spawn{no_client()};;
Ns.register "talk" talk vartype;;

(* CLIENT *)
let username = getenv "USER";;
let talk = Ns.lookup "talk" vartype ;;
let loc client do { talk(client,username) } ;; ''
```

References

[AG98] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. SRC Research Report 149, Digital Equipment Corporation Systems Research Center, 1998.

[AGH00] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

[ATLLW96] A. Adl-Tabatabai, G. Langdale, S. Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of SIGPLAN '96 Conference on Programming Language Design and Implementation*, 1996.

[BD01] L. Bettini and R. De Nicola. Translating Strong Mobility into Weak Mobility. In G. Picco, editor, *Mobile Agents*, number 2240 in LNCS, pages 182–197. Springer, 2001.

[BD02] L. Bettini and R. De Nicola. A Middleware for Secure Distributed Tuple Spaces. Draft, available at `http://music.dsi.unifi.it/papers.html`, 2002.

[BDFP98] L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-Klaim. In P. Ciancarini and R. Tolksdorf, editors, *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 110–115. IEEE Computer Society Press, 1998.

[BDL02] L. Bettini, R. De Nicola, and M. Loreti. Formulae meet Programs over the Net: a Framework for Reliable Network Aware Programming. Work in progress, 2002.

[BDP01] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Framework for Distributed and Mobile Applications. Submitted, 2001.

[Bet98] L. Bettini. Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile. Master's thesis, Dip. di Sistemi e Informatica, Univ. di Firenze, April 1998.

[BNOW95] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software–Practice and Experience*, 25(S4):87–130, 1995. Also available as Digital Systems Research Center Research Report 115.

[Car95] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.

[CF99] 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.

[CGPV97] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In Vitek and Tschudin [VT97].

[Che91] Bill Cheswick. An Evening with Befferd in which a Cracker is Lured, Endured, and Studied. In *Proc. of the Usenix Winter 1992 Technical Conference*, pages 163–174. Usenix Association, 1991.

[CPV97] A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In R. Taylor, editor, *Proc. of the 19th Int. Conf. on Software Engineering (ICSE '97)*, pages 22–33. ACM Press, 1997.

[DABW99] Leendert van Doorn, Martín Abadi, Michael Burrows, and Edward Wobber. Secure Network Objects. In Vitek and Jensen [VJ99], pages 395–412.

[DFP98] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[DFP99] R. De Nicola, G. Ferrari, and R. Pugliese. Types as Specifications of Access Policies. In Vitek and Jensen [VJ99], pages 117–146.

[DFPV00] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[DL02] R. De Nicola and M. Loreti. A Modal Logic for Mobile Agents. *ACM Transaction of Computational Logics*, 2002. to appear.

[FGS96] W. M. Farmer, J. D. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In *Proc. 19th NIST-NCSC National Information Systems Security Conference*, pages 591–597, 1996.

[FM00] F. Le Fessant and L. Maranget. Compiling Join-Patterns. In *Proceedings 3rd International Workshop on High-Level Concurrent Languages (HLCL'98)*, volume 16 (3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GGM97] B Garbinato, R. Guerraoui, and K. Mazouni. Garf: A Tool for Programming Reliable Distributed Applications. *IEEE Concurrency*, 1997.

[Gla99] G. Glass. ObjectSpace Voyager Core Package Technical Overview, 1999. White Paper.

[Gon99] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.

[HCK94] C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research Report 19887, IBM Research Division, 1994.

[HM92] John Hannan and Dale Miller. From operational semantics to abstract machines. *Journal of Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[HSHB99] M. Hughes, M. Shoffner, D. Hamner, and U. Bellur. *Java Network Programming.* Manning Pubblications Co., 2nd edition, 1999.

[Kna96] F. Knabe. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS workshop on Analysis and Verification of Multiple - Agent Languages*, number 1192 in LNCS. Springer-Verlag, 1996.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design principles and Patterns.* Addison-Wesley, 2nd edition, 1999.

[LO99] D. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, March 1999.

[LVS01] Luís Lopes, Vasco T. Vasconcelos, and Fernando Silva. Fine grained multithreading with process calculi. *IEEE Transactions on Computers*, 50(9):229–233, August 2001.

[Mic] Microsoft Corporation. .net framework. `http://msdn.microsoft.com/library/`.

[MLC98] Dejan S. Milojičíc, William LaForge, and Deepika Chauhan. Mobile Objects and Agents (MOA). In *Proc. of the 4th Conf. on Object-Oriented Technologies and Systems (COOTS-98)*, pages 179–194. USENIX Association, 1998.

[MPW92] R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.

[Obj98] Object Management Group. Corba: Architecture and specification. `http://www.omg.org`, 1998.

[OH] Piet Obermeyer and Jonathan Hawkins. Microsoft .net remoting: A technical overview. `http://msdn.microsoft.com/library/`.

[Ord96] J.J. Ordille. When agents roam, who can you trust? In *Proc. of the 1st Conf. on Emerging Technologies and Applications in Communications*, 1996.

[Pes02] G. Peskine. JAM : the JoCaml Abstract Machine. Private Communication, 2002.

[PR98] A.S. Park and P. Reichl. Personal Disconnected Operations with Mobile Agents. In *Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*, 1998.

[PT97] B Pierce and D. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical report, Indiana University, 1997.

[Rou96] F. Rouaix. A Web navigator with applets in CAML. *Computer Networks and ISDN Systems*, 28(7-11):1365–1371, 1996.

[SBH96] M. Straßer, J. Baumann, and F. Hohl. Mole – A Java Based Mobile Agent System. In *Proc. of the 2nd ECOOP Workshop on Mobile Object Systems*, 1996.

[Sew97] Peter Sewell. On implementations 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.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[Spa89] Eugene H. Spafford. The Internet Worm Program: An Analysis. *Computer Communication Review*, 19(1):17–57, January 1989. Also Purdue Technical Report, Department of Computer Science, Purdue University, Number CSD-TR-823.

[ST98] T. Sander and C. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer, 1998.

[Suna] Sun Microsystem. Sun open net environment. `http://wwws.sun.com/software/sunone/`.

[Sunb] Sun microsystems. RMI, Remote Method Invocation. `http://java.sun.com/products/jdk/rmi`.

[Tho97] Tommy Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997. Also Technical Report 1083, University of Rennes IRISA.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, LFCS, University of Edinburgh, June 1996. CST-126-96 (also published as ECS-LFCS-96-345).

[VJ99] J. Vitek and C. Jensen, editors. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS. Springer-Verlag, 1999.

[VT97] J. Vitek and C. Tschudin, editors. *Mobile Object Systems - Towards the Programmable Internet*, number 1222 in LNCS. Springer, 1997.

[W3C] W3C. Soap version 1.2 part 1: Messaging framework w3c working draft 26 june 2002. `http://www.w3.org/TR/2002/WD-soap12-part1-20020626/`.

[WPWD97] D. Wong, N. Paciorek, T. Walsh, and J. DiCelie. Concordia: An Infrastructure for Collaborating Mobile Agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proc. of the 1st Int. Workshop, MA '97*, number 1219 in LNCS, pages 86–97. Springer-Verlag, 1997.

[WS00] Pawel Wojciechowski and Peter Sewell. Nomadic pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, 2000.

[Yee99] B.S. Yee. A Sanctuary For Mobile Agents. In Vitek and Jensen [VJ99], pages 261–273. Also Technical Report CS97-537, University of California at San Diego.