Interactive Mobile Agents in X-KLAIM

Lorenzo Bettini  Rocco De Nicola  Rosario Pugliese  
Dipartimento di Sistemi e Informatica  
Università di Firenze  
{bettini,denicola,pugliese}@dsi.unifi.it  

GianLuigi Ferrari  
Dipartimento di Informatica  
Università di Pisa  
giangi@di.unipi.it

Abstract

Mobile agents are processes which can migrate and execute on new hosts. Mobility is a key concept for network programming; it has stimulated much research about new programming languages and paradigms. X-KLAIM is an experimental programming language, inspired by the Linda paradigm, where mobile agents and their interaction strategies can be naturally programmed. In this paper, a prototype implementation of X-KLAIM is presented together with a few examples introducing the reader to the new programming style.

1. Introduction

Networking has turned computers from isolated data processors into powerful communication and elaboration devices and has led to so-called global computers and global information structures [4]. Global structures/computers are rapidly evolving towards programmability; an illustrative example is the World–Wide Web (WWW). One could easily imagine applications with programs running at different sites and needing continuous interactions or applications that have to take decisions according to information retrieved from the global environment. This new scenario has called for new programming languages and paradigms for global programming (see e.g. [1,3,15], we refer the reader to [14] for a detailed survey).

This paper presents the X-KLAIM (eXtented Kernel Language for Agents Interaction and Mobility) global programming language. X-KLAIM is a general purpose language, inspired by the Linda paradigm [11,5], for programming distributed systems composed of several components interacting through multiple tuple spaces.

The requirements and the design philosophy of the language are presented in [9]; preliminary presentations can be found in [7,8]. The X-KLAIM paradigm identifies processes as the primary points of computation, and nets as the coordinators of process interactions. Both processes and nets are programmed in X-KLAIM.

A global X-KLAIM program, called a net, is structured as a collection of nodes. A node consists of a process component and a tuple space component. Processes may access tuple spaces through explicit naming: operations over tuple spaces are indexed with the locality of the tuple space. Localities are the symbolic names for nodes and, hence, programmers are not required to know the precise mapping of localities into physical addresses (i.e., IP addresses).

This enables programmers to concentrate on the distributed structure of their programs while ignoring the precise physical allocations. The net primitives are designed to provide the programmer with the distributed infrastructure for handling all issues related to physical distribution, scoping and mobility of processes (e.g., the visibility of localities, the allocation policies of tuple spaces, scoping disciplines of mobile agents).

We argue that the separation between process primitives and net (coordination) primitives provides a clean abstraction device for global programming languages. The design of X-KLAIM’s primitives in terms of process primitives and net primitives was influenced by recent works on coordination models and languages (see, e.g., [10] and the references therein).

The rest of the paper is organized as follows. Sections 2 briefly discusses X-KLAIM primitives and introduces, via simple examples, mobile agent programming in X-KLAIM. The main aim of these examples is pointing out the impact of specific scoping disciplines on programming mobile agents. A more advanced programming example is presented in Section 3. Section 4 describes KLAVA (that stands for KLAIM in Java), the prototype implementation [2] of X-KLAIM using the Java programming language [1]. The implementation consists of two layers: the X-KLAIM compiler and the intermediate language KLAVA that is obtained by extending Java with a new package, called KLAVA. The KLAVA package contains all the classes which implement X-KLAIM runtime system and operations. The main classes of the KLAVA package are illustrated in Section 4.
2. X-KLAIM Overview

In this section we briefly discuss the X-KLAIM primitives. We describe the features of the language mainly by key examples; the complete syntax of the language can be found in [9]. Programs are structured around the notions of localities, tuples, processes and nets.

**Localities** must be thought of as the symbolic names for sites (or physical localities). X-KLAIM programs are distributed across sites, and localities allow programmers to concentrate on the distributed structure of their programs while ignoring the precise physical allocations. Localities are explicit in the syntax (i.e. processes and tuples can directly contain localities) while sites are not. A distinguished locality, self, is assumed. Processes can use self to refer to their current execution site.

**Tuples** are sequences of information items, called fields. We distinguish between actual fields (i.e. expressions, string values, localities, processes) and formal fields (i.e. variables). Syntactically, a formal field is written as “!ide”, where ide is a variable identifier. For instance, the sequence ("Shop", Q(x,y,10), !Price) is a tuple with three fields. The first is a string value, the second is a process (with three parameters), and the third field is a formal.

A tuple space is a multiset of tuples. 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 variables. Variables match any value of the same type, and two values match only if they are identical. For instance, the tuple ("Camera", "Shop", 300) matches the tuple ("Camera", "Shop", !Price).

**Processes** are the active computational units. Processes may be executed concurrently both at the same site or at different sites. X-KLAIM provides a high-level syntax for processes; it allows variable declarations and assignments, conditional process expressions, sequential composition, iterative statements (while loops). Special syntax for standard primitive types (like integer or strings) is also provided. More interesting are the basic operations, called functions. We describe the features of the language mainly by examples.

The operation which puts information within a tuple space has, again, two variants. Action out(⟨t⟩)@l adds the tuple resulting from the evaluation of t to the tuple space located at l. Action eval(P)@l spawns a process (whose code is given by P) in the tuple space located at l.

Finally, action newloc(⟨u⟩) creates a "new" site that can be accessed only via locality u.

X-KLAIM allows programmers to specify also the allowance of an action, namely the maximum time (timeout) programmers are willing to wait to perform one operation on the tuple space. If the action does not complete within the specified time-out alternative activities can be performed. For instance the allowance of an input action can be programmed as follows

if in(t)@l within delta then P else Q endif

Notice that non blocking actions can be simulated by specifying a zero time-out. In the actual implementation, time-outs are expressed in milliseconds.

A set is a finite set of nodes. A node has the form (s,P,T,ρ) where s is a site, P is a process, T is a tuple space and ρ is the allocation environment, i.e. a (partial) function from localities to sites. It is required that, for a given node (s,P,T,ρ), ρ(self) = s. For instance, the heading of a node (the declaration of the site and of the allocation environment), programmed by the coordinator, is

MarketPlace :: { l1 ~ s1, l2 ~ s2, l3 ~ s3 }

where l~s is used to indicate the binding between locality l and site s in the environment. The binding of self is implicit in the syntax.

Processes at each site can potentially access any other site of the net; however, site visibility is (locally) controllable via the allocation environment: a site s′ is visible at node (s,P,T,ρ) only if s′ belongs to the image of ρ.

Nets have to be understood as defining X-KLAIM coordination language: each node is a coordinator and coordinators can be composed in parallel. X-KLAIM coordination language is designed to handle all the issues related to the physical distribution of processes and tuple spaces. Moreover, it is used to control changes of network configurations. Changes may be due to the addition/deletion of software components and sites, or to the transmission of programs and resources.

Let us now examine some simple examples that illustrate how X-KLAIM deals with resource bindings when processes migrates. A more advanced programming example will be presented in the next section.

Our first example illustrates the case of a process that moves from one node of the net to another while maintaining the bindings of the original node (i.e. static scoping discipline). We consider a net consisting of two sites s and
A client process P is allocated at site s and a server process S is allocated at site s1. The server S can accept client processes for execution. The client process sends process Q to the server. The X-KLAIM code of the processes is presented below.

\[ P = \text{out}(Q)@l1 \]
\[ Q = \text{in}("foo", !x)@self; \text{out}("foo", x+1)@self \]
\[ S = \text{in}(!x)@self; \text{eval}(x)@self \]

The behaviour of the processes above depends on the allocation environments of the sites of the nets. Assume that the tuple spaces located at s and s1 both contain the tuple ("foo", 1), and that the local allocation environment \( \rho \) of site s maps \texttt{self} into s, and l1 into s1, while the allocation environment \( \rho_1 \) of site s1 maps \texttt{self} into s1.

The client process P sends process Q for execution at the server node (locality l1 is bound to s in \( \rho \)). After the execution of the operation \texttt{out}(Q)@l1, the tuple space at site s1 contains a tuple where the process identifier Q is packaged with the allocation environment \( \rho_1 \) where \texttt{self}=s and \( l_1=s_1 \). This tuple can be properly interpreted as a closure as it consists of the code of the process together with a table associating localities in the code to their site values. Hence, when executed at server site the mobile process Q modifies the value of tuple "foo" at client’s site. In other words, the meaning of localities is determined by the bindings at client site and not by the bindings at the execution site.

Our second example illustrates that a dynamic scoping strategy is used when mobile agents migrate from one computing environment to another. In this case the client process P is programmed as \texttt{eval}(Q)@l1; when the \texttt{eval}(Q)@l1 operation is executed, the process Q is spawned at the remote node (which l1 is bound to) without carrying the allocation environment \( \rho \) with it. Therefore, the execution of Q depends only on the allocation environment \( \rho_1 \) (i.e. process Q modifies the value of "foo" at the server site).

### 3. An Electronic Marketplace

In this section we code in X-KLAIM a program where an autonomous agent makes a search by travelling among sites and then returns the results of its search to the original site.

Assume that a human being wants to buy a specific camera. To decide where to purchase the camera, she/he activates a migrating agent which is programmed to find and return the name of the closest (within the chosen area) camera shop with the lowest price for the camera.

First we define a client process, \texttt{CameraClient}, that given the camera make, the maximal distance, and the locality where the results must be placed, activates a mobile agent (through the \texttt{eval}) to operate on the Electronic Marketplace. Figure 1 and Figure 2 give the X-KLAIM code of the \texttt{CameraClient} process. Notice that the environment associated with the node \texttt{CameraClient} maps the logical locality \texttt{market} to the physical one \texttt{MarketPlace}. The mobile agent, called \texttt{MarketPlaceAgent}, once at destination, asks for the list of the Shops in the selected shopping area. If this list is empty, it simply sends back this information, otherwise it sends a new agent, \texttt{ShopAgent}, to the first shop in the list. The \texttt{ShopAgent} searches the price of the camera and updates the lowest price. Then, it activates a new \texttt{ShopAgent} agent on the next shop in the list, or, if there are no more shops, sends the result back to the \texttt{CameraClient}. In the code of \texttt{MarketPlaceAgent} and \texttt{ShopAgent} we can see an example of the use of a tuple space as a place holder (a list). The retrieval operations of shops from the shops list is programmed via a time-out. Also the \texttt{ShopAgent}, that looks for the lowest price of the camera, makes use of a time-out: the shop may not have that camera make, and the agent does not want to wait forever. A \texttt{MarketPlace} process waits for requests, and once it gets one it constructs the required shop list. A \texttt{Shop} process simply stores the prices of the cameras in its tuple space. Figure 3 gives a pictorial representation of the program.

### 4. KLAVA Overview

While the preceding sections summarized issues related to the language primitives and programming paradigm, in
this section we turn our attention to KLAVA, the prototype implementation of X-KLAIM [2]. The X-KLAIM language has been implemented on top of the Java programming language [1]. The X-KLAIM implementation consists of two layers: the X-KLAIM compiler and the KLAVA intermediate language. KLAVA is essentially Java extended with a new package, called Klava, which contains all classes implementing the runtime system support for X-KLAIM. The compiler translates X-KLAIM nets into KLAVA. Hence, the efficiency of the X-KLAIM implementation relies exclusively on the efficiency of Java. The X-KLAIM compiler is quite standard, thus, here we concentrate on the presentation of the main features of KLAVA.

The class Tuple implements methods for handling tuples (creating a tuple, adding elements to a tuple, getting an element of a tuple, etc...). The method match, which given another tuple as a parameter, does the matching with the current tuple, is a method of this class.

The class TupleSpace provides us with the methods to store tuples in a tuple space, and to update a tuple space. In particular, the Linda operations in, out, and read are implemented as methods of this class.

Any class which represents an object that can be stored as a field of a tuple must implement the interface TupleItem. This interface contains methods to check whether a field is a formal, to update a formal field and to test if two fields are equal. These methods are used in the method match of the class Tuple.

To use standard types (e.g. Integer, String, Vector) the Klava package provides classes that implement the interface TupleItem. The names of these classes start with a K and continues with the name of type they represent, e.g. KInteger, KString, etc. Basically, these classes act as wrappers for the original Java classes. Notice that the classes we described so far, are sufficient for using tuples and tuple spaces, and the standard Linda operations on them.

The class Net implements X-KLAIM coordination language: a Net object is a server which contains the code for registering the nodes of a net. Hence, there must be a program that runs a net object on a certain host. Every node which wants to join a particular net, must login into that net, by specifying its own physical locality (namely, its site). The net object takes care that unique physical localities are used.

A net object is programmed in the standard multithread-server paradigm: a main thread is listening for incoming connections; when a new connection request arrives, a new thread (a NodeHandler object), that will handle that node, is spawn and the main thread will keep on listening for incoming connections.

In KLAVA, localities (both logical and physical) are nothing but strings. A logical locality is mapped into a physical locality, and a physical locality is mapped into an Internet address. However, the only Internet address that has to be known is the address (and port) of the host where the net (the server program) is running. In the remaining cases only localities are used. Using strings instead of Internet addresses makes X-KLAIM programs more portable and reusable: if a node is executed on a different host, but the node is registered with the same name (physical locality), the other nodes need not to be changed in order to access it.

There are three classes to handle localities. The abstract class Locality is the base class. The other two classes LogicalLocality and PhysicalLocality are derived from this base class. These classes wrap the string which represents the locality. A variable which represents a locality should always be declared as a Locality: in this way polymorphism can be used extensively. Clearly, Locality implements TupleItem interface, and, therefore, localities (both logical and physical) can be fields of a tuple.

Processes are derived from the class KlavaProcess by overriding the method execute, which is invoked to activate a process (similarly, in Java one derives from the class Thread and overrides the run method).

Class Node represents a node of a net. A Node object contains a (unique) tuple space as data member and exports methods to access this tuple space, that is, the methods in,
processes, this means that a class which returns a brand-new locality, is also provided by the Net server on which the net server is running must be specified. When a node object is constructed, the physical location of that node in the net and the IP address of the host are collected: an object of this class includes source and destination localities, an opcode that identifies the type of message and a content field, which, since it is declared as object, may contain any object from any class. Serialization is used to send messages through the network. Hence, every object that we want to include in a message must implement java.io.Serializable.

In particular, also a KlavaProcess can be sent in a message. However, in this case, at the destination node the class of that particular process may be unknown, and an exception may be thrown if we try to use the received process. Hence, a process must be sent together with its class binary information, and recursively, with class informations of all the objects the process uses. Clearly, only the informations of user defined classes are to be sent, as the other ones are common to every Klava application. This information is obtained through class introspection (reflection). The following pseudo code shows how process classes binary data are collected:

```
for every field type class, every parameter type class, every return type class do
    collect this class binary data (the contents of .class files)
    call recursively this routine on this class
end
```

call recursively this routine on parent class.

All the nodes that are willing to accept remote processes (this is optional because of possible security problems) must have a custom class loader; i.e. an object of NodeClassLoader class. If such a class loader does not find class information neither in the standard Java classes nor in the local file system, it tries with a local table of class binary data. Therefore, when a process is received from the network, before using it, the node must add class data (received together with the process) to its class loader’s table. A similar approach was adopted in the implementation of the Aglets library [12]. The Java’s approach for applet class loader, namely “when you need a class and you cannot find it, fetch it from the network”, has not been followed because, due to its extensive use of network communications, it turns out to be inefficient.\footnote{Efficiency has always been kept in mind for Klava implementation. This is the reason why Java RMI (Remote Method Invocation) mechanism has not been used: also a non blocking operation, such as out, would have blocked the process for network accesses.}

KLAVA uses multithreading extensively: when a message is sent, it is not actually delivered immediately, but it is inserted in a queue. A different thread (a MessageDeliverer) will take care of actually sending the messages that are on the queue (see Figure 4). If this thread is delayed (or blocked) due to high traffic in the network, the main process can do any non blocking operation meanwhile. Security restrictions can be applied to remote processes, by using and possibly customizing the SecurityManager, that is provided in KLAVA.

Figure 4. Indirect Communication.

out, and read. These methods will call tuple space object’s homonymous methods. The difference is that these methods also accept a locality as a parameter. The newloc method, which returns a brand-new locality, is also provided by the class Node. A Node object is also a container of running processes, this means that eval is provided as a method. Allocation environments are simply implemented as hash tables. When a node object is constructed, the physical locality of that node in the net and the IP address of the host on which the net server is running must be specified. If the physical locality already exists in that net, the request will be refused; otherwise the node will join the net.

Communications between nodes can take place in two ways:

- Indirectly, through the Net server: a node can send a message to a certain locality via its NodeHandler. The NodeHandler will forward the message to the NodeHandler that manages the node at that locality, and, finally, the latter will deliver the message to the destination node (see Figure 4).

- Directly: a node establishes a direct connection to a particular node and sends messages to that node directly through this connection. To establish this connection a node has to ask the Net server the IP address of the other node.

Notice that in case of firewalls or network restrictions, the access to a remote site is allowed only through a server host, namely only the first solution can be adopted.

Node communications rely on objects of the class NodeMessage. An object of this class includes source and destination localities, a content field that identifies the type of message, and a field that contains any object from any class. Serialization is used to send messages through the network. Hence, every object that we want to include in a message must implement java.io.Serializable.
5. Concluding Remarks and Related Work

We have designed and implemented a global programming language based on a two level paradigm: processes and net coordinators. Our experience showed that this is a simple and promising approach to global programming.

Security has been a major concern in the design of X-KLAIM. In [8] a sophisticated type system has been defined that can be used to statically enforce security properties. The type system permits one to check whether the operations X-KLAIM processes intend to perform over the nodes of a net do comply with their access rights. In this paper we did not address typing issues and we presented the simplified untyped language. To provide some security support to the language, we are currently developing the static type checker for access types. However, there are still other major security requirements (e.g. cryptographic primitives and dynamic transmission of access rights) that have to be studied and developed.

Despite the different programming paradigms, there are interesting similarities between Telescript and X-KLAIM. General Magic’s Telescript [15] is an object oriented language designed for network programming. A central concept in Telescript is that of place, which corresponds to X-KLAIM site. A place can be thought of as the stationary process that can accept mobile agents. Agents travel from one place to another by invoking the go operation. This operation requires the agent’s destination place (the ticket) and the route of the trip. The main advantage of X-KLAIM’s approach is that the “possible stationary processes” can be programmed via the notion of locality without requiring the precise physical distribution of places. In other words, localities provide a powerful abstraction mechanism over sites. There are also some analogies between our eval/out operations and Telescript go operation: both allow mobile agents to be programmed. Java does not permit saving and serializing the execution context of a thread (stack pointer, program counter, etc.), so a process cannot be started from the exact point of its migration (like go does); this is not a problem in KLAVA, as eval make a process run from the beginning, upon arrival at destination. More recently, General Magic developed Odyssey [13], a Java library that implements the functionalities of Telescript.

Java has also been used to implement a dialect of Linda called Jada [6]. Jada supports a version of Linda with multiple tuple spaces. Tuple spaces are the key notion of Jada; they are autonomous entities, distributed over the nodes of a net and identified by the internet address of the nodes where they are placed. Processes use tuple spaces by connecting to the nodes where they are placed and by invoking their methods. However, in Jada there is no distinction between logical and physical addresses, and there is no dynamic creation of tuple spaces. Moreover, Jada does not support process mobility, namely the eval primitive is not implemented and processes cannot be exchanged in communications.

The Aglets library [12] is another Java package. It has been specifically designed for programming mobile agents; KLAVA, instead, is a package oriented to program more general, possibly distributed applications.

Acknowledgments This work has been partially supported by Esprit Working Groups CONFÉRED and COORDINA, by CNR Project Speciale Metodologie e Strumenti di Analisi, Verifica e Validazione di Sistemi Software Affidabili.

References