Mobile Applications in X-KLAIM

Lorenzo Bettini[1] Rocco De Nicola[1] GianLuigi Ferrari[2] Rosario Pugliese[1]

[1]Dipartimento di Sistemi e Informatica, Università di Firenze – e-mail: {bettini,denicola,pugliese}@dsi.unifi.it
[2]Dipartimento di Informatica, Università di Pisa – e-mail: giangi@di.unipi.it

Abstract

Networking has turned computers from isolated data processors into powerful communication and elaboration devices, called global computers*; an illustrative example is the World–Wide Web. Global computers are rapidly evolving towards programmability. The new scenario has called for new programming languages and paradigms centered around the notions of* mobility *and* location awareness*. In this paper, we briefly present* X-KLAIM*, an experimental programming language for global computers, and show a few programming examples.*

1. X-KLAIM Overview

X-KLAIM (eXtended Kernel Language for Agents Interaction and Mobility) is an experimental programming language that can be used for programming distributed systems and mobile agents interacting through multiple tuple spaces [7]. The language design philosophy and the underlying computational model, KLAIM, are presented in [6]. X-KLAIM implementation in Java [1] is briefly described in [3] and presented in detail in [2]. In this section we summarize the main features of the language; in Section 2 we briefly introduce the framework for programming in X-KLAIM and in Section 3 we show three programming examples.

X-KLAIM programs are structured around the notions of *localities*, *tuples*, *tuple spaces*, *processes* and *nets*.

Localities can be thought of as the symbolic names for *sites* (or net addresses). X-KLAIM programs are distributed across sites, and localities allow programmers to focus on the distributed structure of their programs while ignoring the precise physical allocations. A distinguished locality, `self`, is assumed. Processes can use `self` to refer to their current execution site. Logical localities are mapped to sites by means of *allocation environments*.

Tuples are sequences of information items, called *fields*. We distinguish between actual fields (e.g. 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.

Tuple spaces are multisets 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 identical. For instance, the tuple `("Camera", "Shop", 300)` matches the tuple `("Camera", "Shop", !Price)`.

Processes are the active computational units; they can be executed concurrently both at the same site or at different sites, and can be exchanged as first class data. X-KLAIM has primitives for variable declaration, assignment, sequential, conditional and iterative composition.

There are six basic operations that processes can perform over tuple spaces and nodes of nets. The operation which retrieves information from a tuple space has two variants. Operation **in**`(t)@l` evaluates tuple `t` and looks for a matching tuple `t1` in the tuple space located at `l` (`l` is a locality and gives the logical address of the tuple space). Whenever the matching tuple `t1` is found, it is removed from the tuple space. The corresponding values of `t1` are then assigned to the variables in the formal fields of `t` and the operation terminates. If no matching tuple is found, the operation is suspended until one matching tuple is available. Operation **read**`(t)@l` differs from **in**`(t)@l` because the tuple `t1` selected by pattern–matching is not removed from the tuple space. Operation **out**`(t)@l` adds the tuple resulting from the evaluation of `t` to the tuple space located at `l`.

There are two operations that permit sending processes for execution on, possibly remote, nodes: operation **eval**`(P)@l` spawns the process with code `P` at the node located at `l`; operation **go**`@l` causes the migration of the exe-

cuting process to location `l` where its execution will be resumed. Finally, operation **newloc**`(u)` creates a "new" node that can be accessed only by referring to locality `u`.

The addition of tuples to a tuple space never blocks the execution of processes, while the retrieval of tuples does. However, programmers have the possibility of specifying the *allowance* of a blocking operation: a parameter can be added with the maximum waiting time, expressed in milliseconds. If the operation does not complete within the specified *time-out*, alternative activities can be performed. For instance the allowance of an input operation can be programmed as follows

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

In this way, non blocking operations can be simulated by specifying a zero time-out.

Nets are finite sets of nodes. Each *node* consists of a site, a set of parallel processes, a tuple space and an allocation environment. It is required that the allocation environment of a node always maps the reserved locality `self` to the site of the node. 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 a node only if there is a locality mapped to s by the local allocation environment.

2. X-KLAIM implementation

The implementation of the programming language X-KLAIM consists of two layers:

- a Java package, called KLAVA, which contains all the classes that implement the X-KLAIM runtime system and operations;
- the X-KLAIM compiler that translates X-KLAIM programs into Java programs that use the package KLAVA.

The structure of the framework for our language is depicted in Figure 1. X-KLAIM distribution is available on line at `http://rap.dsi.unifi.it`. KLAVA is briefly described in [3] and presented in detail in [2].

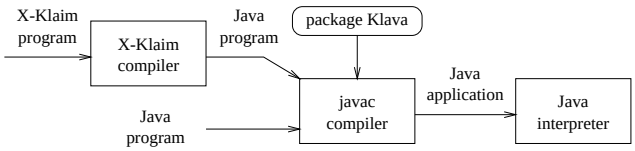

Figure 1. The framework for X-KLAIM

Thus, if X-KLAIM source code is stored in the file called `foo`, it can be compiled by means of the command

```
xklaim foo
```

that will create the file `foo.java`. On its turn, this last file can be compiled and executed by means of the standard jdk commands:

```
javac foo.java
java foo <host> <port>
```

where the host and the port number of the *Net server*, which is a class in the package KLAVA, are also specified. This server keeps track of the physical localities of the nodes which are part of the net and must be started before any node.

X-KLAIM can be used to write the highest layer of distributed applications while KLAVA can be used to customize certain behaviors, by specializing KLAVA classes.

3. Programming Examples

In this section, by means of a few programming examples, we show how our language can be used to control code mobility and to coordinate distributed applications [1].

3.1. An Electronic Marketplace

Our first example shows an autonomous X-KLAIM agent, called `Collector`, that travels over the nodes of a net for performing a search and returning the result. A motivating scenario can be the following: suppose that someone wants to buy a specific product at a market made of geographically distributed shops. To decide at which shop to buy, she/he activates a migrating agent which is programmed to find and return the name of the closest shop (i.e. the shop within the chosen area, determined by a maximal distance parameter) with the lowest price.

`Collector` takes as parameters the product name, the maximal distance and the locality where the result of the search must be returned. The agent is sent for execution at the node containing the marketplace Directory, where it asks for the list of the shops in the selected shopping area. Then, `Collector` migrates to the first shop in the list. At each shop, `Collector` checks the price of the wanted product, possibly updating the information about the lowest price and the shop that offers it, and migrates to the next shop in the list. If there are no more shops to visit, `Collector` sends the result of the search back to the locality received as parameter. The code of agent `Collector` is:

```
rec Collector[ ProductMake : str, distance : int,
               retLoc : loc ]
  declare
    var shopList : TS ;
    var nextShop, CurrentShop, thisShop : loc ;
```

[1] In the code fragments, comments start with symbol #.

```
  var again : bool ;
  var CurrentPrice, newCost : int
begin
  # ask for a list of shops that are not
  # such far away than a certain distance
  out( "cshop", distance )@self;
  in( "cshop", !shopList )@self;
  again := true ;
  CurrentPrice := 0 ;
  CurrentShop := self ;
  while ( again ) do
    # while there are shops in the list
    if in( ! nextShop )@shopList within 0 then
       thisShop := nextShop ;
       go@nextShop ; # migrate to the next shop
       if read( ProductMake, ! newCost )@self
                             within 10000 then
         if ( CurrentPrice = 0 OR
              newCost < CurrentPrice ) then
           # update the best price
           CurrentPrice := newCost;
           CurrentShop := thisShop
         endif
       endif
    else
       again := false ;
       # done: send back the results
       out( ProductMake, CurrentShop,
            CurrentPrice )@retLoc
    endif
  enddo
end
```

Notice that `Collector` uses a variable of type 'tuple space' to implement the shops list and uses time-outs both to retrieve the next shop in the list and to look for the price of the wanted product (this avoids the agent to wait forever when the shop list is empty or when the shop does not stock the wanted product).

If we fail to retrieve the information about the desired product we could also think of leaving a *pending* agent in that site which keeps on searching for that information, while the main agent visits the other shops in the list:

```
       if read( ProductMake, ! newCost )@self
                             within 10000 then
          ...
       else
          eval(
            read( ProductMake, ! newCost )@self ;
            out( newCost, self )@PAsite
            +
            in( "KILL", self )@PAsite
          )@self ;
          out( thisShop )@pendingAgents ;
       endif
```

The pending agent tries to read (a tuple with) the wanted information or a termination signal (`"KILL"`); The operator for non-deterministic composition, +, is borrowed from process algebras [11]: $P_1 + P_2$ spawns both processes P_1 and P_2, but only one of the two will continue its execution. In our case, non-deterministic composition only makes sense if both processes start with a blocking operation: the process that firstly finds a matching tuple will continue.

Later, after having visited all shops in its list, the main agent can go back to the site of the client and recontact the pending agents (collected in the list `pendingAgents`) to see if these agents were able to find the wanted information; if they still failed to retrieved such information they can be terminated:

```
       ...
       # go back to the site of the client
       go@retLoc ;
       # and recontact pending agents
       while ( again ) do
         # while there are pending agents
         if in( ! shop )@pendingAgents
                within 0 then
           if read( ! newCost, shop )@PAsite
                    within 0 then
             if ( CurrentPrice = 0 ... ) then
               ... # update current price and shop
             else
               out( "KILL", shop )@PAsite
             endif
           endif
         else
           again := false ;
       enddo ;
       # done: communicate the results
       out( ProductMake, CurrentShop,
            CurrentPrice )@self
       ...
```

Notice that a private locality, `PAsite`, is used in order to communicate with pending agents, so that there cannot be any interference with other processes. For instance, the main agent may have created this locality before leaving the site of the client.

3.2. Load Balancing

In this second scenario, we figure out remote clients that send processes for execution to a server node which distributes the received processes over a group of processors by using, each time, the idlest one. This is determined by using the *Leaky Bucket Of Credits* [14] pattern. When entering the system, each processor sends a number of 'credits' to the server. This number corresponds to the processor willingness to perform computations on behalf of the server. The server stores the number of credits in a database and, when a processor is needed, the one with the highest number of credits is chosen and the credit number is decreased:

```
rec DeliverProcess[ ProcessorDB : ts ]
declare
 var P : process ;
 var HighestCredit, NumOfProcessors,
        ProcessorNumber, Credits, i : int ;
 var Processor, HighestProcessor, screen : loc ;
 var sentProcess : bool
begin
 while ( true ) do
   in( !P )@self ; # wait for a process
   sentProcess := false ;
   while ( sentProcess != true ) do
     HighestCredit := 0 ;
     in( !NumOfProcessors )@ProcessorDB ;
     i := NumOfProcessors ;
     # search for the processor
     # with highest number of credits
     while ( i > 0 ) do
       read( !Processor, !Credits, i )@ProcessorDB ;
       if ( Credits > HighestCredit ) then
```

```
        HighestCredit := Credits ;
        HighestProcessor := Processor
      endif ;
      i := i - 1
    enddo ;
    if ( HighestCredit = 0 ) then
      # no more credits for any Processor...
      # ...wait for new credits
      out( "WAIT" )@ProcessorDB ;
      out( NumOfProcessors )@ProcessorDB ;
      # wait for notification of new credits
      in( "WAKE UP" )@ProcessorDB
    else
      out( eval( P )@HighestProcessor )
        @HighestProcessor ;
      # update its credits
      in(HighestProcessor, HighestCredit,
         !i)@ProcessorDB ;
      out(HighestProcessor, HighestCredit - 1,
          i)@ProcessorDB ;
      sentProcess := true ; # found a free Processor
      out( NumOfProcessors )@ProcessorDB
    endif
  enddo
enddo
end
```

Notice that the server may exhaust all credits; in that case it waits (`out("WAIT")@ProcessorDB`) until it is notified that new credits have arrived (`in("WAKE UP")@ProcessorDB`). The code executed is:

```
while( true ) do
 in( "SERVER", "CREDIT",
     !Processor )@CreditLoc ; # get a credit
 out( "Credit from: " )@screen ;
 out( Processor )@screen ;
 in( !NumOfProcessors )@ProcessorDB ; # locks DB
 if in( Processor, !NumOfCredits,
        !ProcessorNumber )@ProcessorDB within 0 then
   # update this Processor's credits number
   out( Processor, NumOfCredits + 1,
        ProcessorNumber )@ProcessorDB ;
   if in( "WAIT" )@ProcessorDB within 0 then
    # someone's waiting to be notified
    out( "WAKE UP" )@ProcessorDB
   endif
 endif ; # ignore errors
 # release DB
 out( NumOfProcessors )@ProcessorDB
enddo
```

When a processor receives a process, it immediately starts executing it and, after an "appropriate" waiting, sends a credit back to the server. This is implemented by the following code fragment

```
rec ReceiveProcess[ server : loc ]
  declare
    var P : process ;
    var screen : loc
  begin
    while ( true ) do
      in( !P )@self ;
      out( "Received Process\n" )@screen ;
      eval( P )@self ;
      Sleep( 700 ) ; # wait for a while
      out( "Sending Credit to Server\n" )@screen ;
      # send one credit
      out( "SERVER", "CREDIT", self )@server
    enddo
  end
```

This pattern is based on the (heuristic) fact that if a processor is busy, it cannot send a credit back, or at least it does not send it immediately.

We want to point out that locality `screen` is actually attached to the output device. Hence, operation `out("Received Process\n")@screen` corresponds to printing tuple `("Received Process\n")` on the screen. Indeed, I/O operations in X-KLAIM are implemented as tuple space operations as well. Figure 2 presents a screen shot of the server and three processors.

Figure 2. Load balancing: a screen shot of the server and some processors

3.3. A Chat System

In our last scenario we have a simple chat system consisting of a few clients and a server. The server registers the clients and delivers messages to them. Clients pop messages from the net by executing

```
while ( true ) do
 (
  in( "MESSAGE", !message, !from )@self ;
  out( "(" )@screen ;
  out( from )@screen ;
  out( ") " )@screen ;
  out( message )@screen ;
  out( "\n" )@screen
  +
  in( "PERSONAL", !message, !from )@self ;
  out( "PERSONAL " )@screen ;
  out( "(" )@screen ;
  out( from )@screen ;
  out( ") " )@screen ;
  out( message )@screen ;
```

```
  out( "\n" )@screen
 )
enddo
```

and keep track of the users currently in the chat-room by executing the following code that intercepts server's messages of a certain form:

```
while ( true ) do
 (
  in( "ENTER", !nickname, !from )@self ;
  out( nickname )@screen ;
  out( " entered chat\n" )@screen ;
  out( nickname )@usersList
  +
  in( "LEAVE", !nickname, !from )@self ;
  out( nickname )@screen ;
  out( " left chat\n" )@screen ;
  in( nickname )@usersList
 )
enddo
```

Again, we want to point out that both localities `screen` and `usersList` are attached to output devices.

The server receives messages from clients and delivers these messages to every client:

```
while ( true ) do
 in( "MESSAGE", !message, !from )@self ;
 if read( ! num, !sender, from )@usersDB within 0 then
   BroadCast( "MESSAGE", message, sender, usersDB )
 endif
enddo
```

When new clients enter the chat, the server accepts them only if there is no other client with the same nickname, and in case the access is granted, every client is notified about the new client; note that the new client is also provided with the list of the clients currently in the chat.

```
while ( true ) do
 in( "SERVER", "ENTER", !nickname, !from )@self ;
 in( ! userNum )@usersDB ; # lock DB
 if read( ! num, nickname, !user )@usersDB within 0 then
   out( false )@from ;
   out( userNum )@usersDB # release DB
 else
   userNum := userNum + 1 ;
   out( userNum, nickname, from )@usersDB ;
   out( true )@from ;
   out( userNum )@usersDB ; # release DB
   SendUserList( from, usersDB ) ;
   out( nickname )@users ;
   out( "Enetered Chat : " )@screen ;
   out( nickname, from )@screen ;
   BroadCast( "ENTER", nickname, "server", usersDB )
 endif
enddo
```

A screen shot of a chat session is reported in Figure 3. The input text areas (with buttons) and the list of currently connected users are accessible through tuple spaces. The code that reacts to input events is:

```
while ( true ) do
 in( !message )@messageKeyb ;
 # is there someone selected?
 out( "getSelectedItem", ID )@usersList ;
 in( "getSelectedItem", ID, !selected )@usersList ;
 if ( selected != "" ) then
   newloc( selectedUsers ) ;
   out( selected )@selectedUsers ;
   # there's some one selected
   out( "PERSONAL", message, selectedUsers,
        self )@server
 else
   out( "getSelectedItems", ID )@usersList ;
   in( "getSelectedItems", ID,
       !selectedUsers )@usersList ;
   if read( !selected )@selectedUsers within 0 then
     # there's some one selected
     out( "PERSONAL", message, selectedUsers,
          self )@server
   else
     # no one selected: broadcast
     out( "MESSAGE", message, self )@server
   endif
 endif
enddo
```

A string entered in the message input text area is retrieved by means of command `in( !message )@messageKeyb`, where `messageKeyb` is the tuple space "connected" to that field. If a user in the list is selected, the message is dispatched only to him/her, otherwise it is delivered to every client. The list of the selected users is retrieved by means of tuples with string `"getSelectedItems"` as first field.

4. Related Work

General Magic's *Telescript* [15] has been, probably, the first well-known language designed for network programming with mobile agents. A central concept in *Telescript* is that of *place*, which in X-KLAIM corresponds to site. *Telescript* agents migrate from one place to another by invoking the **go** operation, which is similar to the X-KLAIM **go** operation. The main advantage of X-KLAIM's approach are the notions of (logical) locality and allocation environment, which provide powerful abstractions over the precise physical distribution of sites. *Telescript* has been replaced first by *Odyssey* [8], a Java [1] framework that implements all of *Telescript* functionalities, and then by *ADF* [10], an agent scripting language based on *XML*.

There are currently a number of other Java frameworks and libraries that implement useful functionalities for programming distributed and mobile applications. The IBM *Aglets* [9] is specifically designed for programming mobile agents. *Jada* [5] implements a dialect of Linda with multiple tuple spaces; there is neither distinction between logical and physical addresses, nor dynamic creation of tuple spaces, nor support for process mobility. *MARS* [4] adds programmability to tuple spaces by associating reaction rules to communication events.

Other languages which exploit the multiple tuple spaces paradigm [7] to coordinate mobile agents are *Lime* [13], that allows processes to transiently share their own tuple spaces, and *TuCSoN* [12], that permits programming the behavior of tuple spaces in response to communication events.

Figure 3. A chat system: a screen shot of the server and some clients

References

[1] A. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[2] 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.

[3] 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 IEEE Seventh International Workshop on Enabling Technologies: Infrastructure for Collaborating Enterprises*, pages 110–115. IEEE Computer Society Press, 1998.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer-Verlag, Berlin, 1998.

[5] P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 213–228. Springer-Verlag, Berlin Germany, 1997.

[6] 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.

[7] D. Gelernter. Multiple Tuple Spaces in Linda. In J.Hartmanis G. Goos, editor, *Proceedings, PARLE '89*, volume 365 of *LNCS*, pages 20–27, 1989.

[8] General Magic, Inc. Odyssey, 1998. `http://www.genmagic.com/technology/-odyssey.html`.

[9] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[10] D.B. Lange, T.Hill, and M. Oshima. A New Internet Agent Scripting Language Using XML. In *Proc of AAAI-99 Workshop on AI in Electronic Commerce*, July 1999.

[11] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[12] A. Omicini and F. Zambonelli. Coordination of mobile information agents in Tucson. *Journal of Internet Research*, 8(5), 1998.

[13] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, 1999.

[14] J.M. Vlissides, J.O. Coplien, and N.L. Kerth, editors. *Pattern Languages of Program Design 2*. Addison-Wesley Publishing Company, Reading, MA, USA, 1996.

[15] J. E. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.