SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 4, pp. 13–35. http://www.scpe.org
ISSN 1895-1767

IMPLEMENTING MOBILE AND DISTRIBUTED APPLICATIONS IN X-KLAIM*

LORENZO BETTINI, ROCCO DE NICOLA AND MICHELE LORETI

Abstract. In this paper we present X-KLAIM, an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple distributed tuple spaces and mobile code. The language consists of a set of coordination primitives inspired by Linda, a set of operators for building processes borrowed from process algebras and a few classical constructs for sequential programming. We present some programming examples in X-KLAIM, dealing with mobile code programming paradigms, such as client-server, code mobility and mobile agents.

1. Introduction. Technological advances of both computers and telecommunication networks, and development of more efficient communication protocols are leading to an ever increasing integration of computing systems and to diffusion of so called Global Computers [19]. These are massive networked and dynamically reconfigurable infrastructure interconnecting heterogeneous, typically autonomous and mobile components, that can operate on the basis of incomplete information. Designing and implementing applications over a global network is inherently different from designing and implementing stand–alone ones. Network programming has to deal with the following additional issues [20]:

- the physical distribution of hosts and data can be essential, and local and remote behaviors can be significantly different;
- systems are asynchronous and less predictable: a temporary disconnection of a remote host cannot be distinguished from a system fault;
- different forms of program termination have to be considered; applications can terminate due to missing permissions or to the low level of quality of services.

Global Computers are thus fostering a new style of distributed programming that has to take into account variable guarantees for communication, cooperation and mobility, resource usage, security policies and mechanisms for dealing with failures. This has stimulated the proposal of new theories, computational paradigms, linguistic mechanisms and implementation techniques. We have thus witnessed the birth of many calculi and kernel languages intended to support programming according to the new style and to provide tools for formal reasoning over the modeled systems.

In this paper we present X-KLAIM, an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple tuple spaces and mobile code (possibly object-oriented). X-KLAIM is based on KLAIM the *Kernel Language for Agents Interaction and Mobility* [23, 8]. The distinguishing features of the approach is the explicit use of localities for accessing data or computational resources. KLAIM can be seen as an asynchronous higher–order process calculus whose basic actions are the original Linda [28] primitives enriched with explicit information about the location of the nodes where processes and tuples are allocated.

The *blackboard* approach, of which tuple space based models are variants, is one of the most appreciated model for dealing with mobile agents (see, e.g., [26], that examines several messaging models for mobile agents) also because of its flexibility. The Linda *asynchronous* communication model permits

- *time uncoupling*: tuples' life time is independent of the producer process' life time,
- *destination uncoupling*: the creator of a tuple is not required to know the future use or the destination of that tuple,
- *space uncoupling*: communicating objects need to know a single interface, i. e., the operations over the tuple space. This approach is also called *flow-of-objects* [4] as opposed to *method invocation*, which requires many interfaces for the operations supplied by remote objects.

When moving to open distributed systems and large-scale, multi-users applications, the Linda coordination model suffers from the lack of *modularity* and *scalability*: identification tags of tuples, which are conceptually part of different contexts, may collide. In other words, processes of different computations could interfere and a mechanism to structure communication and hide information, e.g., to create areas restricted to a subset of the processes, is needed. Explicit localities enable the programmer to distribute and retrieve data and processes to and from the sites of a net and to structure the tuple space as multiple, located spaces. Moreover, localities, considered as first–order data, can be dynamically created and communicated over the network. The overall outcome is a powerful programming formalism that, for example, can easily be used to model encapsulation. In fact, an encapsulated module can be implemented as a tuple space at a private locality, and this ensures controlled accesses to data.

*The work presented in this paper has been partially supported by EU Project Software Engineering for Service-Oriented Overlay Computers (SENSORIA, contract IST-3-016004-IP-09).

In the rest of the paper we present the main features of X-KLAIM, and some programming examples dealing with code and agent mobility (e.g., a load balancing system, Section 3.3, and a mobile agent based information retrieval, Section 3.2) and with distributed applications in general (a chat system, Section 5). The last two examples show two more involved systems: a mobile agent based system for distributed document updates (Section 6; this is a modified version of the system presented in [12]) and an distributed implementation of the strategy game Cluedo (Section 7).

For a more complete description of the programming language X-KLAIM we refer the interested reader to the tutorial that can be found in [10] (from where we borrow modified versions of some examples shown in this paper).

2. An overview of X-KLAIM**.** X-KLAIM (*eXtended* KLAIM) [11, 10] 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*) [8] and is inspired by the coordination language Linda [28], hence it relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are sequences 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 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*. 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 distinct logical locality, `self`, can be used by processes to refer the node where they are executing on.

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 of 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. An allocation environment has the shape $\{\ldots, l_i \sim s_i, \ldots\}$, where l_i are logical localities and s_i are physical localities.

KLAIM processes may run concurrently, both at the same node or at different nodes, and can execute the following operations over tuple spaces and nodes:

- **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**(`proc`)$@l$: spawns process `proc` 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.

X-KLAIM extends KLAIM with the typical constructs of high level programming languages: variable declarations, assignments, conditionals, sequential and iterative process composition. Moreover, X-KLAIM provides specific statements to simplify multiple access to tuple spaces (**forall**). Please notice that, all the X-KLAIM constructs can be *encoded* within KLAIM. However, introducing the new statements in the X-KLAIM syntax makes the programmers life easier.

The implementation of X-KLAIM is based on KLAVA, a Java [5] package that provides the run-time system for X-KLAIM operations, and on a compiler, which translates X-KLAIM programs into Java programs that use KLAVA. The

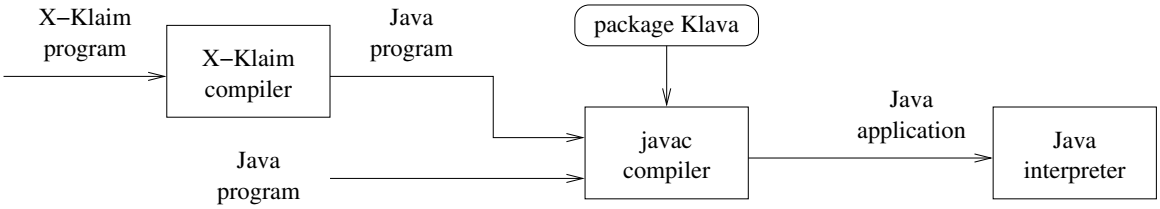


FIG. 2.1. *The framework for* X-KLAIM.

structure of the KLAIM framework is outlined in Figure 2.1. 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 X-KLAIM paradigm. With this respect, by using KLAVA directly, the programmer is able to exchange, through tuples, any kind of Java object, and implement a more fine-grained kind of mobility, as shown in [14]. X-KLAIM provides both *weak mobility* (via operation **eval**) and *strong mobility* (via operation **go**, explained later in this section). Conversely, KLAVA supports weak mobility only; indeed, Java does not allow to save and restore the execution state. X-KLAIM and KLAVA are available on line at `http://music.dsi.unifi.it`. KLAVA is presented in detail in [14, 7].

TABLE 2.1
X-KLAIM *process syntax. Syntax for other standard expressions is omitted.*

RecProcDefs	::=	**rec** id formalparams procbody | **rec** id formalparams **extern**
	|	RecProcDefs **;** RecProcDefs
formalParams	::=	**[ ]** | **[** paramlist **]**
paramlist	::=	id **:** type | **ref** id **:** type | paramlist **,** paramlist
procbody	::=	declpart **begin** proc **end**
declpart	::=	ε | **declare** decl
decl	::=	**const** id **:=** expression | **locname** id | **var** idlist **:** type | decl **;** decl
idlist	::=	id | idlist **,** idlist
proc	::=	KAction | **nil** | id **:=** expression | **var** id **:** type | proc **;** proc
	|	**if** boolexp **then** proc **else** proc **endif**
	|	**while** boolexp **do** proc **enddo**
	|	**forall** Retrieve **do** proc **enddo**
	|	procCall | **call** id | **(** proc **)** | **print** exp
KAction	::=	**out(** tuple **)@**id | **eval(** proc **)@**id | Retrieve | **go@**id | **newloc(** id **)**
Retrieve	::=	Block | NonBlock
Block	::=	**in(** tuple **)@**id | **read(** tuple **)@**id
NonBlock	::=	**inp(** tuple **)@**id | **readp(** tuple **)@**id | Block **within** numexp
boolexp	::=	NonBlock | *standard bool exp*
tuple	::=	expression | proc | **!** id | tuple **,** tuple
procCall	::=	id **(** actuallist **)**
actuallist	::=	ε | expression | proc | id | actuallist **,** actuallist
expression	::=	$*$ expression | *standard exp*
id	::=	*string*
type	::=	**int** | **str** | **loc** | **logloc** | **phyloc** | **process** | **ts** | **bool**

X-KLAIM syntax is shown in Table 2.1. We just briefly recall the more relevant features. Local variables of processes are declared in the **declare** section of the process definition. Standard base types are available (**str**, **int**, etc.) as well as X-KLAIM typical types: **loc** for generic locality variables (without specifying whether it is logical or physical), **logloc** (resp. **phyloc**) for logical (resp. physical) localities, **process** for process variables and **ts**, i. e., tuple space, for implementing data structures by means of tuple spaces, e.g., lists, that can be accessed through standard tuple space operations. Logical locality constants are declared by using the type **locname**. Finally, Comments start with the symbol **#**.

A locality variable can be initialized with a string that will correspond to its actual value. Logical localities are basically names, while physical localities must have the form <IP_address>:<port>, so a physical locality variable has to be initialized with a string corresponding to an Internet address.

Logical locality resolution can be performed by putting the operator $*$ in front of the locality that has to be evaluated:

```
l := *output; # retrieve the physical locality associated to output
out(*output)@self; # insert the physical locality associated to output
```

However, logical localities used as "destination" are still evaluated automatically in both network models, i. e., if the locality used after the `@` is a logical one, it is first translated to a physical locality.

Apart from standard KLAIM operations, X-KLAIM also provides non-blocking version of the retrieval operations, namely **readp** and **inp**; these act like **read** and **in**, but, in case no matching tuple is found, the executing process does not block but `false` is returned. Indeed, **readp** and **inp** can be used where a boolean expression is expected. These variants, used also in some versions of Linda [21], are useful whenever one wants to search for a matching tuple in a tuple space with no risk of blocking. For instance, **readp** can be used to test whether a tuple is present in a tuple space.

A timeout (expressed in milliseconds) can be also 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 (these boolean expressions can be combined in order to execute more complex retrieval operations):

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

Time-outs can be used when retrieving information to avoid that processes block because of network latency or of missing tuples.

X-KLAIM provides the construct **forall** that can be used for iterating actions through a tuple space by means of a specific template. Its syntax is:

```
forall Retrieve do
 proc
enddo
```

We refer the reader to Table 2.1 for the syntax of "Retrieve". The informal semantics of this operation is that the loop body "proc" is executed each time a matching tuple is available. Even duplicate tuples are repeatedly retrieved by the **forall** primitive; it is however guaranteed that each tuple is retrieved only once. Thus, instead of the while-based code above, we write:

```
forall readp(!i, !s)@self do
 out(i + 1, s)@l
enddo
```

Now, if the tuple space contains three matching tuples (of which two are identical): (10, `"foo"`), (10, `"foo"`), (20, `"bar"`), after the execution of the loop instruction the tuple space at `l` will contain the tuples (11, `"foo"`), (11, `"foo"`), (21, `"bar"`).

Notice however that the tuple space is not blocked when the execution of the **forall** is started, thus this operation is not atomic: the set of tuples matching the template can change before the command completes. A locked access to such tuples can be explicitly programmed (see, e.g., Listing 6.3). Our version of **forall** is different from the one proposed in [17] since parallel processes are not created for each retrieved tuple (this would not be consistent with the "iterating" nature of **forall**; a similar functionality could be easily achieved by using **eval** in the loop body). Our **forall** is similar to the **all** variations of retrieval operations in *PLinda* [3].

The **forall** primitive has a different semantics depending on the nature of the retrieval operation: if a blocking action is used the process executing **forall** is blocked until another (never retrieved) tuple becomes available; instead, when a nonblocking action is used, the process exits from the **forall** loop and continues its execution when no other matching tuple is available.

Data structures can be implemented by means of the data type **ts**; a variable declared with such type can be considered as a tuple space and can be accessed through standard tuple space operations, apart from **eval** that would not make sense when applied to variables of type **ts**. Furthermore **newloc** has a different semantics when applied to such a variable: it empties the tuple space.

forall is then useful for iterating through such data structures; for instance the following piece of code transforms a list, stored in the variable `list` of type **ts**, containing data of the shape (**str**, **int**) into a list containing data of the shape (**int**, **str**):

```
declare
  var s : str;
  var i : int;
  var list : ts;
...
forall inp(!s, !i)@list do
  out(i, s)@list
enddo
```

Notice that the non-blocking version of **in** is used, otherwise the process would be blocked when it finishes iterating through the list.

The action **go**@l [9] makes an agent migrate to l and resume its execution at l from the instruction following the migration. This action permits modeling strong mobility. Thus in the following piece of code an agent retrieves a tuple from the local tuple space, then migrates to the locality `l` and inserts the retrieved tuple into the tuple space at locality `l`:

```
in(!i, !j)@self;
go@l;
out(i, j)@self
```

I/O operations are implemented as tuple space operations. For instance the logical locality *screen* is actually attached to the output device. Hence, operation **out**("foo\n")@*screen* corresponds to printing the string "foo\n" on the screen. Similarly, the locality *keyboard* can be attached to the input device, so that a process can read what the user typed with a **in**(!s)@*keyboard*. Further I/O devices, such as files, printers, etc., can also be handled through the locality abstraction.

A node of an X-KLAIM net can be specified as follows:

```
physloc :: { ... , l~ s, ... } init_processes
```

where `physloc` is the physical locality of the node and `{ ... , l ~ s, ... }` is its allocation environment. Notice that `self` is automatically associated to the physical locality of the node and it does not have to be specified in the environment. `init_processes` are the processes executed automatically when the node is started; basically they have the same functionality of `main` in C and Java. Throughout the paper we will omit the definition of nodes when it is not strictly relevant.

3. Mobility Examples. In this section we show a few programming examples taking advantage of process mobility, implemented in X-KLAIM.

News gathering. The first example is a *news gatherer*, that relies on mobile agents for retrieving information on remote sites. We assume that some data are distributed over the nodes of an X-KLAIM net and that each node either contains the information we are searching for, or, possibly, the locality of the next node to visit in the net.

The agent *NewsGatherer* first tries to read a tuple containing the information we are looking for, if such a tuple is found, the agent returns the result back home; if no matching tuple is found within 10 seconds, the agent tests whether a link to the next node to visit is present at the current node; if such a link is found the agent migrates there and continues the search, otherwise it reports the failure back home.

The implementation of the agent exploiting strong mobility (by means of the migration operation **go**) is reported in Listing 3.1. Notice that the use strong mobility makes the source quite clear.

Information retrieval. The next example is still an autonomous information retrieval agent in the context of a virtual *market place*: 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. The implementation of the agent `MarketPlaceAgent` is shown in Listing 3.2.

The `MarketPlaceAgent` 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 (by means of an **eval** not shown here) for execution at the node containing the marketplace directory, where it asks for the list of the shops in the selected shopping area. Then, `MarketPlaceAgent` migrates to the first shop in the list. At each shop, `MarketPlaceAgent` 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, `MarketPlaceAgent` sends the result of the search back to the locality received as parameter. The list of nodes to visit is stored in a list (implemented through a **ts**) and **forall** is used for iterating over this list.

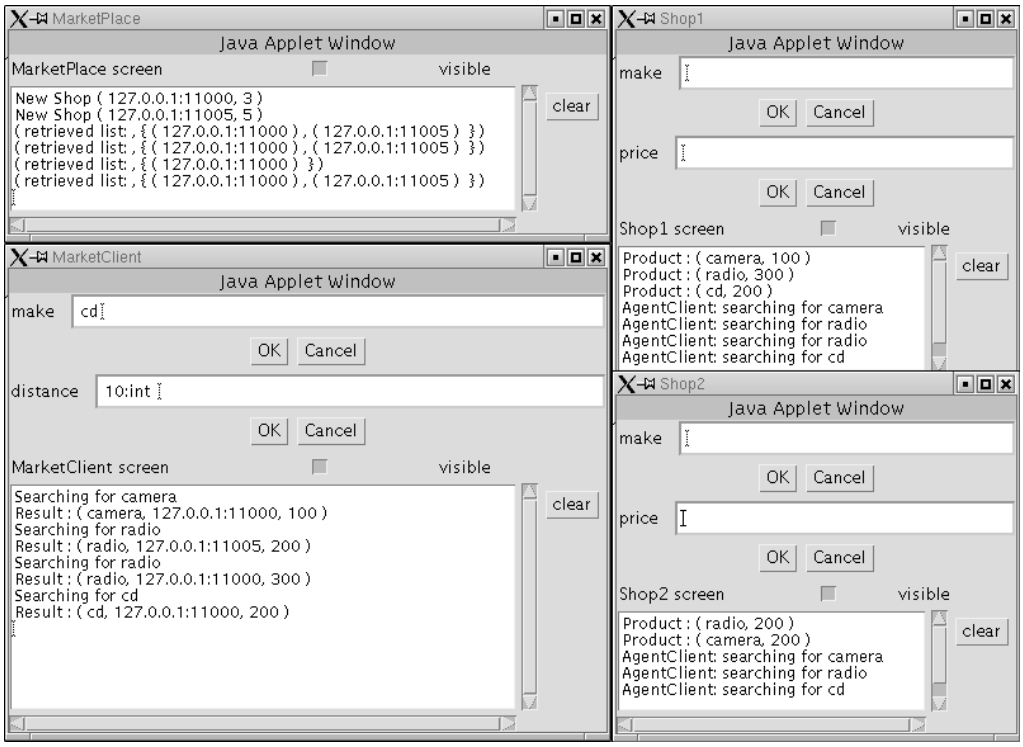
LISTING 3.1
X-KLAIM *implementation of a news gatherer using strong mobility.*

```
rec NewsGatherer[ item : str, retLoc : loc ]
 declare
  var itemVal : str ;
  var nextLoc : loc ;
  var again : bool
 begin
  again := true;
  while again do
   if read( item, !itemVal )@self within 10000 then
    go@retLoc;
    print "found " + itemVal;
    again := false;
   else
    if readp( item, !nextLoc )@self then
     go@nextLoc
    else
     go@retLoc;
     print "search failed";
     again := false
    endif
   endif
  enddo
 end
```

LISTING 3.2
X-KLAIM *implementation of an agent visiting shops of a virtual market place searching for an item with the lowest price.*

```
rec MarketPlaceAgent[ ProductMake : str, retLoc : loc, distance : int ]
 declare
  var shopList : TS ;
  var nextShop, CurrentShop, thisShop : loc ;
  var CurrentPrice, newCost : int ;
  locname screen
 begin
  out( "cshop", distance )@self; # ask for a list of shops within a distance
  in( "cshop", !shopList )@self;
  out( "retrieved list: ", shopList )@screen;
  CurrentPrice := 0 ;
  CurrentShop := self ;
  forall inp( ! nextShop )@shopList do  # while there are shops to visit
   thisShop := nextShop ;
   go@nextShop ; # migrate to the next shop ;
   out( "AgentClient: searching for ", ProductMake )@screen ;
   if read( ProductMake, ! newCost )@self within 10000 then
    if ( CurrentPrice = 0 OR newCost < CurrentPrice ) then
     CurrentPrice := newCost; # update the best price
     CurrentShop := thisShop
    endif
   endif
  enddo ;
  out( ProductMake, CurrentShop, CurrentPrice )@retLoc # OK, let's send the results
 end
```

Screenshot 3.1 shows a client that performs some searches through the `MarketPlaceAgent` in two shops. In this example there are two shops affiliated to the market place: `Shop1` at physical locality `127.0.0.1:11000` with a distance of 3, and `Shop2` at physical locality `127.0.0.1:11005` with a distance of 5; this information is shown in the window of the market place directory (up left). The client sends the agent searching for a `camera` within a distance of 10, so the market place directory provides the agent with a list made of the localities of the two shops, and after visiting both, the agent reports home that the first shops sells the searched item at the lower cost. The second query has basically the same parameters but the agent has to search for a `radio` and this time the second shop sells it at the lower price. Then it still searches for a radio but within a closer distance (e.g., 4) and this time the second shop is not even visited (since its distance is 5, so the market place directory does not put it into the list communicated to the agent). Finally a `cd` is searched for (within a wider distance) and when visiting the second shop a timeout is raised, since that shop does not sell that item.



SCREENSHOT 3.1. *The market place directory (up left), the market client (down left) and two shops of the virtual market place.*

Load balancing. We conclude this section by presenting an example that uses the remote evaluation paradigm, thus, the code does not to autonomously migrate: it is moved by another process. This example implements a *load balancing system* that dynamically redistributes mobile code among several processors: we suppose that remote clients send processes for execution to a server node that distributes the received processes among a group of processors by using, each time, the (estimated) idlest one. Each processor sends a number of "credits" to the server (this number corresponds to the processor availability to perform computations on behalf of the server); the server stores the number of credits in a database and, when needed, it chooses the processor with the highest number of credits and decreases this number.

When a processor receives a process, it immediately starts executing the process (in a parallel thread) and sends a credit back to the server. Indeed, the system is based on the heuristic that if a processor is busy, it cannot send a credit back, or at least it does not send a credit immediately (this is also known as *Leaky Bucket Of Credits* pattern [2]).

This example is implemented by the code fragment in Listing 3.3 that shows the server that dispatches the received process to the idlest processor (left) and the processor that receives a process for execution from the server and sends a credit back to it. The code presented here is simplified in order to concentrate on the code mobility related parts (e.g., it does not handle cases such as all credits are exhausted for all processors). Notice that processes are exchanged by means of **out** and **in**.

4. Node Connectivity in X-KLAIM**.** The original KLAIM model of [23] has been extended in [15] to deal more directly with *open nets*. The original formalism is enriched with explicit connectivity actions and with a new kind of processes, that we called *Node Coordinators*, which are the only ones allowed to perform privileged connectivity actions. This distinction provides a fine-grain separation between the coordination level and the standard action execution level. Furthermore, the allocation environment can be modified dynamically with the primitive **bind**.

In the hierarchical model, any node plays both the role of computational environment (for processes and tuples), and a *gateway*, (for managing subnets of other nodes). Nodes can act both as clients (belonging to a specific subnet) and as servers (taking charge of, possibly private, subnets). Logical localities represent the names that client nodes can specify when entering the subnet of a server node, and allocation environments, that can be dynamically updated with such information, actually represent dynamic tables mapping logical names (possibly not known in advance) into physical addresses; these mappings are allowed to change during the evolution. The client-server relation among nodes smoothly leads to a hierarchical model, also because of the way logical names are "resolved": in order to find the mapping for a

LISTING 3.3

Load balancing: (left) the server receives a process and dispatches it to the idlest processor; (right) the processor node receives a process and executes it locally and sends a credit back to the server.

```
rec DeliverProcess[ ProcessorDB : ts ]
 declare
  var P : process ;
  var HighestCredit, Credits : int ;
  var Processor, HighestProcessor : loc
 begin
  while ( true ) do
   in( !P )@self ; # wait for a process
   HighestCredit := 0 ;
   forall readp( !Processor, !Credits )@ProcessorDB do
    if ( Credits > HighestCredit ) then
     HighestCredit := Credits ;
     HighestProcessor := Processor
    endif
   enddo ;
   out( P )@HighestProcessor ;
   # update its credits
   in( HighestProcessor, HighestCredit )@ProcessorDB ;
   out( HighestProcessor, HighestCredit − 1 )@ProcessorDB
  enddo
 end
```

```
rec ReceiveProcess[ server : loc ]
 declare
  var P : process ;
  locname screen
 begin
  while ( true ) do
   in( !P )@self ;
   eval( P )@self ;
   out( "SERVER", "CREDIT",
       self )@server
  enddo
 end
```

locality, allocation environments of nodes in this hierarchy are now inspected from the bottom upwards. This resembles name resolution within DNS servers. We shall consider further this issue in Section 4, where we will describe how node connectivity is managed in X-KLAIM.

X-KLAIM provides all the primitives for explicitly dealing with node connectivity. Consistently with the hierarchical model of KLAIM such actions can be performed only by *node coordinators*. The syntax of node coordinators is shown in Table 4.1, and is basically the same of standard X-KLAIM processes (Table 2.1) apart from the new privileged actions. We briefly comment these new actions:

TABLE 4.1

X-KLAIM *node coordinator syntax. This syntax relies on standard process syntax shown in Table 2.1.*

NodeCoordinator	::=	**rec** NodeCoordDef
NodeCoordDef	::=	**nodecoord** id formalparams declpart nodecoordbody
	|	**nodecoord** id formalparams **extern**
nodecoordbody	::=	**begin** nodecoordactions **end**
nodecoordaction	::=	*standard process action* | **login(** id **)** | **logout(** id **)**
	|	**accept(** id **)** | **disconnected(** id **)** | **disconnected(** id **,** id **)**
	|	**subscribe(** id **,** id **)** | **unsubscribe(** id **,** id **)**
	|	**register(** id **,** id **)** | **unregister(** id **)**
	|	**newloc(** id **)** | **newloc(** id **,** nodecoordactions **)**
	|	**newloc(** id **,** nodecoordactions **,** num **,** classname **)**
	|	**bind(** id **,** id **)** | **unbind(** id **)**

- **login**(*loc*), where *loc* is an expression of type **loc**, logs the node where the node coordinator is executing at the node at locality *loc*; **logout**(*loc*) logs the node out from the net managed by the node at locality *loc*. **login** can be used as a boolean expression in that it returns **true** if the login succeeds and **false** otherwise.
- **accept**(*l*) is the complementary action of **login** and indeed, the two actions have to synchronize in order to succeed; thus a node coordinator on the server node (the one at which other nodes want to log) has to execute **accept**. This action initializes the variable *l* to the physical locality of the node that is logging. **disconnected**(*l*) notifies that a node has disconnected from the current node; the physical locality of such node is stored in the variable *l*. **disconnected** also catches connection failures. Notice that both **accept** and **disconnected** are blocking in that they block the running process until the event takes place. Instead, **logout** does not have to synchronize with **disconnected**.

An example of these four operations is shown in Listing 4.1, where the node coordinators executing on the client are presented on the left, and the complementary ones executing on the server are presented on the right. Notice that the process that executes the **login** communicates with the one that has to execute the **logout** by using a tuple. **accept** and **disconnected** are initializers for the corresponding variables.

LISTING 4.1
*An example showing **login** and **logout** (left) and the corresponding **accept** and **disconnected**.*

```
rec nodecoord SimpleLogin[ server : loc ]
 begin
  print "try to login to " +
      server + "...";
  if login( server ) then
   print "login successful";
   out("logged", true)@self
  else
   print "login failed!"
  endif
 end

rec nodecoord SimpleLogout[ server : loc ]
 begin
  in("logged", true)@self;
  print "logging off from " +
      server + "...";
  logout(server);
  print "logged off."
 end
```

```
rec nodecoord SimpleAccept[]
 declare
  var client : phyloc
 begin
  print "waiting for clients...";
  accept(client);
  print "client " + client + " logged in"
 end

rec nodecoord SimpleDisconnected[]
 declare
  var client : phyloc
 begin
  print "waiting for disconnections...";
  disconnected(client);
  print "client " + client +
" disconnected."
 end
```

- **subscribe**(*loc*, *logloc*) is similar to **login**, but it also permits specifying the logical locality (*logloc* is an expression of type **logloc**) with which a node wants to become part of the net coordinated by the node at locality *loc*; this request can fail also because another node has already subscribed with the same logical locality at the same server. **unsubscribe**(*loc*, *logloc*) performs the opposite operation.
- **register**(*pl*, *ll*), where *pl* is a physical locality variable and *ll* is a logical locality variable, is the complementary action of **subscribe** that has to be performed on the server; if the subscription succeeds *pl* and *ll* will respectively contain the physical and the logical locality of the subscribed node. The association $pl \sim ll$ is automatically added to the allocation environment of the server. **unregister**(*pl*, *ll*) records the unsubscriptions. Notice that an alternative version of **disconnected**, namely **disconnected**(*pl*, *ll*) is supplied, in order to detect lost connections with nodes, that also specifies the logical locality with which a node was subscribed. As the other **disconnected** explained above, this action is more powerful in that it is able to catch also connections brutally closed without an **unsubscribe**. Let us observe that **disconnected** catches also the events of **unregister** so if program uses both, it is up to the programmer to coordinate the two notification actions (an example of such a scenario is shown in Section 5).

bind(*logloc*, *phyloc*) allows to dynamically modify the allocation environment of the current node: it adds the mapping $logloc \sim phyloc$. On the contrary, **unbind**(*logloc*) removes the mapping associated to the logical locality *logloc*. These two operations privileged and only node coordinators can execute them.

In this version of X-KLAIM **newloc** has become a privileged action and is supplied in three forms in order to make programming easier: apart from the standard form that only takes a locality variable, where the physical locality of the new created node is stored, also the form **newloc**(*l*, *nodecoordinator*) is provided. Since **newloc** does not automatically logs the new created node in the net of the creating node, this second form allows to install a node coordinator in the new node that can perform this action (or other privileged actions).

Notice that this is the only way of installing a node coordinator on another node: due to security reasons, node coordinators cannot migrate, and cannot be part of a tuple. In order to provide better programmability, this rule is slightly relaxed: a node coordinator can perform the **eval** of a node coordinator, provided that the destination is `self`.

Finally the third form of **newloc** takes two additional arguments: the port number where the new node is going to be listening (and this also determines its physical locality, since the IP address will be the same of the creator node), and the (Java) class of the new node. Since I/O devices can be abstracted into nodes, this form of **newloc** enables to construct, for instance, the graphical interface of a node, made up of several I/O sub-nodes. For an example, see Section 5, where some I/O logical localities are used as interfaces for text areas, and input text boxes and lists.

5. A Chat System with Connectivity Actions. In this section we present the implementation in X-KLAIM of a chat system. The chat system we present in this section is simplified, but it implements the basic features that are present in several chat systems. The system consists of a `ChatServer` and many `ChatClients`.

The system is dynamic because new clients can enter the chat and existing clients may disconnect. The server represents the gateway through which the clients can communicate, and the clients logs in the chat server by specifying their "nickname", represented here by a logical locality. A client that wants to enter the chat must subscribe at the chat

LISTING 5.1
Node coordinators of the chat server dealing with clients' subscriptions.

```
rec nodecoord HandleLogin[ usersDB : ts ]
 declare
  var nickname : logloc ;
  var client : phyloc ;
  locname users, screen, server
 begin
  while ( true ) do
   if register( client, nickname ) then
    out( nickname, client )@usersDB ;
    out( true )@client ;
    SendUserList( client, usersDB ) ;
    out( (str)nickname )@users ;
    out( "Entered Chat : " )@screen ;
    out( nickname, client )@screen ;
    BroadCast( "USER", "ENTER",
          nickname, server, usersDB )
   endif
  enddo
 end

rec SendUserList[ newEnter : phyloc, usersDB : ts ]
 declare
  var nickname : logloc ;
  var userLoc : phyloc ;
  var userList : ts
 begin
  newloc( userList ) ;
  forall readp( !nickname, !userLoc )@usersDB do
   if ( userLoc != newEnter ) then
    out( nickname )@userList
   endif
  enddo ;
  out( userList )@newEnter
 end

rec nodecoord HandleDisconnected[ usersDB : ts ]
 declare
  var nickname : logloc ;
  var client : phyloc ;
  locname screen
 begin
  while ( true ) do
   disconnected(client, nickname);
   out("disconnected: ", nickname, client)@screen;
   RemoveClient(nickname, usersDB)
  enddo
 end

rec nodecoord HandleUnregister[ usersDB : ts ]
 declare
  var nickname : logloc ;
  locname screen
 begin
  while ( true ) do
   unregister(nickname);
   out("unsubscription: ", nickname)@screen;
   RemoveClient(nickname, usersDB)
  enddo
 end

rec RemoveClient[ nickname : logloc, usersDB : ts ]
 declare
  var client : phyloc ;
  locname screen, users, server
 begin
  if inp( nickname, !client )@usersDB and
      inp( (str)nickname )@users then
    out( "Left Chat : " )@screen ;
    out( nickname, client )@screen ;
    BroadCast( "USER", "LEAVE",
          nickname, server, usersDB )
  endif
 end
```

server. The server must keep track of all the registered clients and, when a client sends a message, the server has to deliver the message to every connected client. If the message is a private one, it will be delivered only to the clients in the list specified along with the message.

The Chat Server. When a new client issues a subscription request, the server accepts it 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; moreover the new client is also provided with the list of the clients currently in the chat (Listing 5.1). The server keeps a database of all connected clients in a variable `usersDB` of type **ts** where there is a tuple of the shape `(nickname, locality)` for each client, where `nickname` is a logical locality and `locality` is a physical one. Notice that all the processes running on the chat server share this database.

The server uses two (node coordinator) processes for intercepting clients' disconnections: `HandleUnregister` and `HandleDisconnected`. The second one would be useless if the network communications are reliable (i. e., no communication suddenly crashes without further notice); however, this assumption may be too strong in a realistic scenario. Thus `HandleDisconnected` intercepts also this kind of disconnections. As we said above the **disconnected** action returns even after an ordinary unsubscription, so the process `RemoveClient` has to further check whether a client has already been removed from the database.

The broadcasting of messages to clients is managed by two processes running on the `ChatServer` node: `BroadCast` and `BroadCastTo` (Listing 5.2): the former sends a message to all connected clients while the latter sends a message only to the clients specified in the list `to`. This second version is useful when delivering *personal* messages.

All messages have the following tuple shape:

```
(communication_type, message_type, message, from)
```

where `communication_type` and `message_type` specify the type of message (e.g., the values `"USER"` together with `"ENTER"` indicate that a user entered the chat, while `"MESSAGE"` and `"ALL"` indicate a chat message that is destined to every client). `message` is the content of the message (e.g., the nickname of the user that entered the chat or the body of a chat message) and `from` is the nickname (logical locality) of the client that originated the message.

LISTING 5.2
Processes on the server dealing with message dispatching.

```
rec HandleMessage[ usersDB : ts ]
 declare
  var message : str ;
  var sender : logloc ;
  var from : phyloc
 begin
  while ( true ) do
   in( "MESSAGE", !message, !from )@self ;
   if readp( !sender, from )@usersDB then
    BroadCast( "MESSAGE", "ALL",
          message, sender, usersDB )
   endif # ignore errors
  enddo
 end

rec HandlePersonal[ usersDB : ts ]
 declare
  var message : str ;
  var sender : logloc ;
  var from : phyloc ;
  var to : ts
 begin
  while ( true ) do
   in( "PERSONAL", !message, !to, !from )@self ;
   if readp( !sender, from )@usersDB then
    BroadCastTo( "MESSAGE", "PERSONAL",
          message, to, sender, usersDB )
   endif
  enddo
 end

rec BroadCast[ communication_type : str, message_type : str,
          message : str, from : logloc, usersDB : ts ]
 declare
  var nickname : logloc ;
  var user : phyloc
 begin
  forall readp( !nickname, !user )@usersDB do
   out( communication_type, message_type,
      message, from )@user
  enddo
 end

rec BroadCastTo[ communication_type : str, message_type : str,
          message : str, to : ts, from : logloc, usersDB : ts ]
 declare
  var nickname : str ;
  var user : phyloc
 begin
  forall inp( !nickname )@to do
   # recipients are specified as strings in the "to" list
   # so we have to convert them first
   if readp( (logloc) nickname, !user )@usersDB then
    out( communication_type, message_type,
       message, from )@user
   endif
  enddo
 end
```

Messages are received by the chat server by means of two processes `HandleMessage` and `HandlePersonal` (respectively for standard chat messages and for personal messages) also shown in Listing 5.2. When a client wants to send a personal message it has to specify also a list (a **ts** tuple field) containing the nicknames of the clients it is destined to). These processes are responsible for delivering a message to all the recipient clients.

The Chat Client. A chat client executes two processes for handling messages dispatched by the server (Listing 5.3): `HandleMessages` takes care of processing chat messages and `HandleServerMessages` handles server messages informing of new clients joining the chat or existing clients leaving (the list of connected clients is updated accordingly). This information is printed on the screen of the client (attached to the locality `screen`).

The user can insert messages for the server (i. e., commands for entering and exiting from the chat) and standard chat messages in two text fields that are attached, respectively, to the localities `serverKeyb` and `messageKeyb`. For each of these localities there is a process, respectively `HandleServerKeyboard` and `HandleMessageKeyboard` (also in Listing 5.3) that read the input of the user and communicate with the server. When `HandleServerKeyboard` reads a tuple of the shape `("ENTER", nickname)` it tries to subscribe at the chat server with that specific nickname. On the contrary, if the tuple contains `"LEAVE"` it unsubscribes.

A user can specify that a chat message is destined only to a restricted number of clients by selecting them from the list of connected clients. Such list is indeed attached to the locality `usersList` that, in turn, is a special tuple space that provides a sort of interface for accessing the items of such list (in the KLAVA implementation this tuple space is an interface for a `java.awt.List` object). Thus a process can access the elements of such a list through tuples that start with the string `"command"` and consist of a specific command and its arguments. For each command the template of the tuple is different. If the result of a command has to be retrieved the request is issued with an **out** and the response retrieved with an **in**. An identifier has to be provided so that a process does not retrieve the result of the request of another process. For instance the following two lines retrieve multiple selected items in the list (the result is stored in the **ts** variable `selected`):

```
out( "command", "getSelectedItem", ID )@usersList ;
in( "command", "getSelectedItem", ID, !selected )@usersList ;
```

If there is some client selected in this list, the message is sent as `"PERSONAL"` and the list of recipients is sent along with the message; otherwise the message is considered destined to all connected clients.

Screenshot 5.1 shows three chat clients and the chat server.

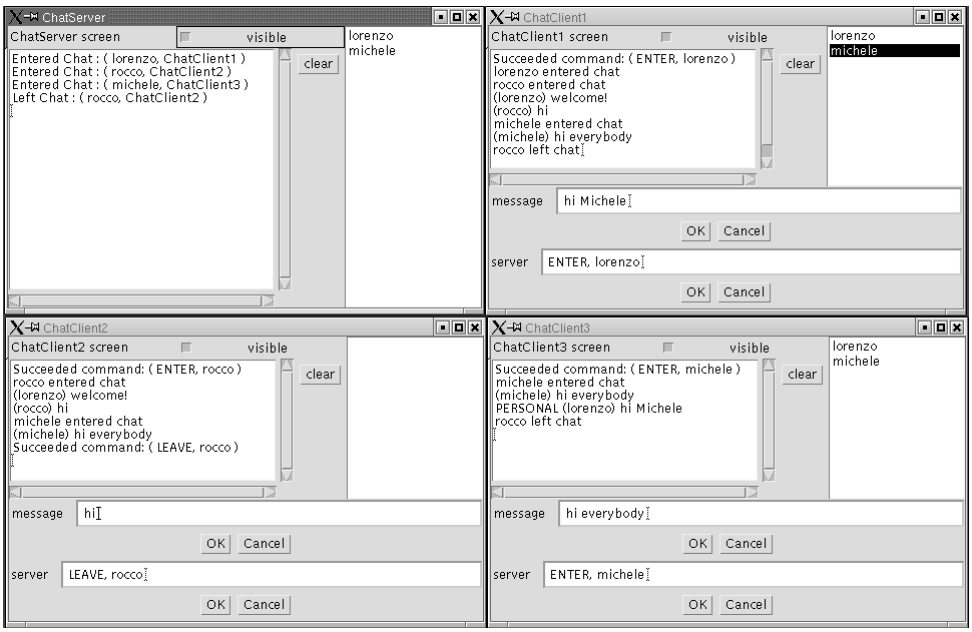
6. A mobile agent based system for document update. In this section, we describe how to use mobile agents to develop a prototype system that permits maintaining *up to date* different documents stored on several heterogeneous

LISTING 5.3
Node coordinators and processes running on a chat client.

```
rec HandleMessages[]
 declare
  locname screen ;
  const standard_message := "MESSAGE";
  var message, message_type : str ;
  var from : logloc
 begin
  while ( true ) do
   in( standard_message, !message_type,
       !message, !from )@self ;
   if message_type = "PERSONAL" then
    out( "PERSONAL " )@screen
   endif;
   out( "(" )@screen ;
   out( (str)from )@screen ;
   out( ") " )@screen ;
   out( message )@screen ; out( "\n" )@screen
  enddo
 end

rec HandleServerMessages[]
 declare
  locname screen, usersList ;
  const user_message := "USER" ;
  var command, nickname : str;
  var from : logloc
 begin
  while ( true ) do
   in( user_message, !command,
       !nickname, !from )@self ;
   if command = "ENTER" then
    out( nickname )@screen ;
    out( " entered chat\n" )@screen ;
    if not readp(nickname)@usersList then
     out( nickname )@usersList
    endif
   else
    if command = "LEAVE" then
     out( nickname )@screen ;
     out( " left chat\n" )@screen ;
     inp( nickname )@usersList
     # ignore non existing names
    endif
   endif
  enddo
 end

rec nodecoord HandleServerKeyboard[]
 declare
  locname server, screen, serverKeyb, usersList;
  var command, nick : str ;
  var nickname : logloc ; var chat_server : phyloc ;
  var response : bool ; var userList : ts
 begin
  chat_server := *server;
  while ( true ) do
   in( !command, !nick )@serverKeyb ;
   if ( command != "ENTER" and command != "LEAVE" ) then
    out( "Unknown command: " )@screen ;
    out( command )@screen ;
    out( "\n" )@screen
   else
    # nick was entered as a string
    nickname := (logloc) nick;
    if command = "ENTER" then
     if subscribe( chat_server, nickname ) then
      out( "Succeeded command: " )@screen ;
      in( !userList )@self ;
      UpdateUserList( userList )
     else
      out( "Failed command: " )@screen
     endif
    else # it is a LEAVE
     unsubscribe( chat_server, nickname ) ;
     out("command", "removeAll")@usersList
    endif ;
    out( command, nickname )@screen
   endif
  enddo
 end

rec HandleMessageKeyboard[]
 declare
  const ID := "messageKeyboard" ;
  var message, selected : str ;
  var selectedUsers : ts ;
  locname messageKeyb, usersList, server
 begin
  while ( true ) do
   in( !message )@messageKeyb ;
   out( "command", "getSelectedItem", ID )@usersList ;
   in( "command", "getSelectedItem", ID, !selected )@usersList ;
   if ( selected != "" ) then
    newloc( selectedUsers ) ;
    out( selected )@selectedUsers ;
    out( "PERSONAL", message, selectedUsers, *self )@server
   else
    out( "command", "getSelectedItems", ID )@usersList ;
    in( "command", "getSelectedItems",
        ID, !selectedUsers )@usersList ;
    if readp( !selected )@selectedUsers then
     out( "PERSONAL", message, selectedUsers, *self )@server
    else
     out( "MESSAGE", message, *self )@server
    endif
   endif
  enddo
 end
```

computers distributed over a network. Documents are installed and updated only on the central server, where clients register for them. When a new version of a document is stored on the server, some agents are scattered along the network to update the corresponding documents on the clients. These mobile agents implement a *push strategy*: subscribed customers are provided with the latest software as soon as it is available.

Upon subscription the client will get the most recent version of the requested documents. Subscription may require a registration and possibly a payment, but we are not addressing these issues, that can be easily added to the system. The delivery of a document and of new versions are made by means of mobile agents, that will migrate to the client's site, and install all the necessary modules. We want to avoid distributed transactions for guaranteeing the correct storage of documents and of new versions. One of the advantages of mobile agents is that they encapsulate transactions, which will take place locally, with no other network connections, apart from the one for sending the agent to a remote computer.



SCREENSHOT 5.1. *Three chat clients and the chat server.*

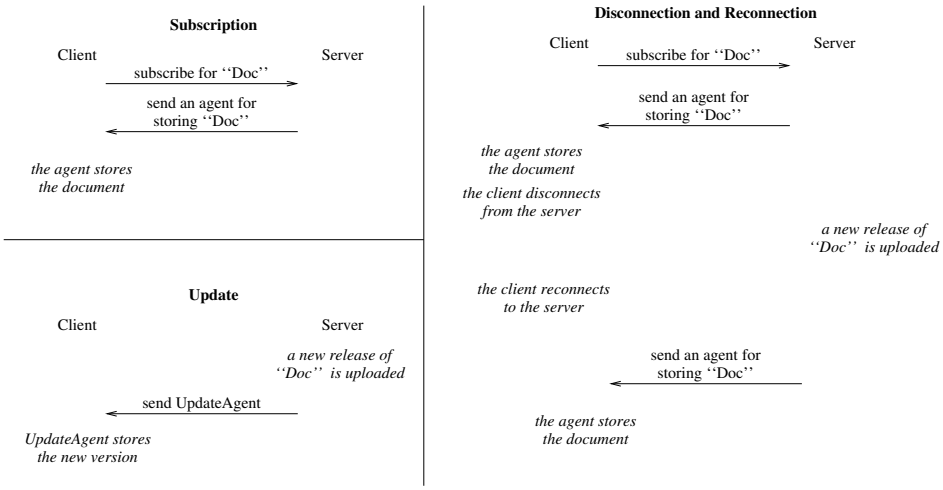


FIG. 6.1. *Subscription and Update*

When the update agent arrives at the client's site, if the document is currently opened, the agent also notifies that a new version of the document is available. Please notice that, the client may decide to disconnect from the server; indeed another advantage of mobile agents is the easy implementation of *disconnected operations*. When the client reconnects to the server, all the documents, which in the meantime have been updated, are sent to the client. Subscription and update (even after disconnection and reconnection) are depicted in Figure 6.1.

6.1. A prototype implementation in X-KLAIM**.** Since this is to be intended as a prototype system, and we are just interested in the design of this kind of applications, not in the details of the implementation, we are not considering advanced features that can be added to the system afterwards. In the X-KLAIM implementation the server waits for connections from clients by executing process `AcceptAgent` (Listing 6.1).

When a new client gets connected with the server, agent `AcceptAgent` verifies if this client has been already registered. Indeed, a list of registered clients is stored in the tuple space `clientlist` where, together with client location, is also maintained the status of the client connection (e.g. `"ON-LINE"` or `"OFF-LINE"`). If the client has been already connected to the server, the status of the connection is changed and a process that updates the documents stored in the client

LISTING 6.1
The process that waits for client connections at the server

```
rec nodecoord AcceptAgent[ ]
 declare
  var client : phyloc;
  locname screen, clientlist
 begin
  while( true ) do
   out( "Waiting for incoming connections...\n" )@screen;
   accept( client );
   out( "Connection established with "+client+"\n")@screen;
   if (inp( client , "OFF-LINE" )@clientlist) then
    eval( CheckForClientUpdate( client ) )@self;
    out( client , "ON-LINE" )@clientlist
   else
    eval( SubscriptionAgent( client ) )@self
   endif
  enddo
 end
```

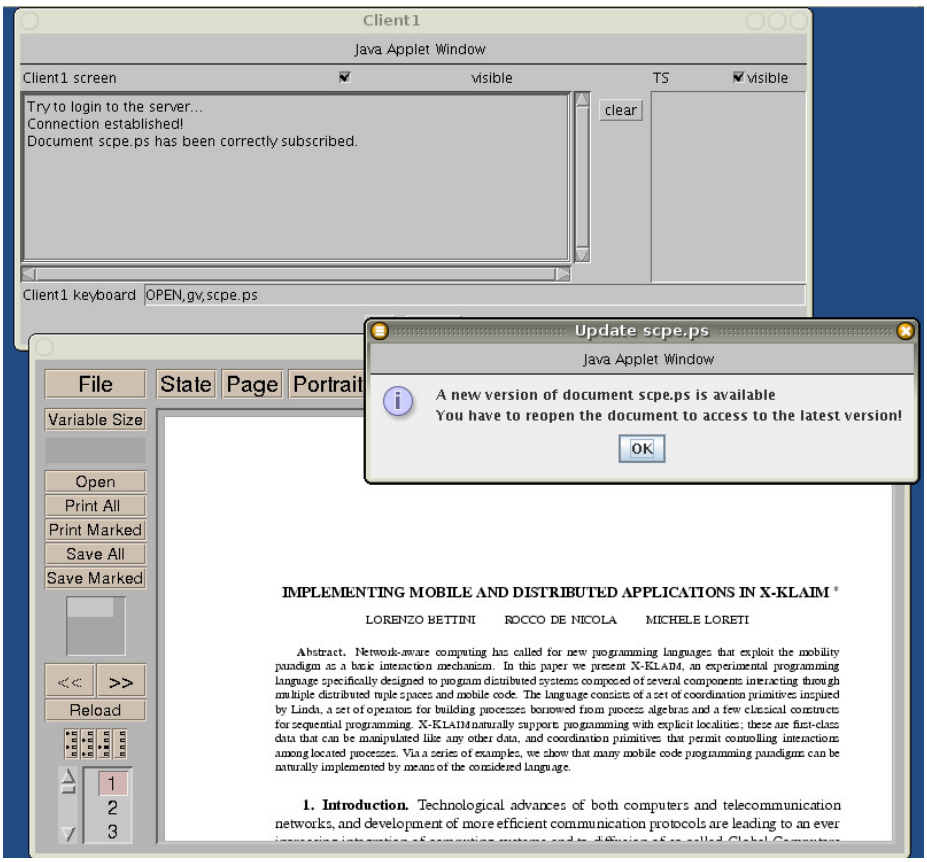
LISTING 6.2
The process that handles document subscriptions

```
rec SubscriptionAgent[ ClientLoc : loc ]
  declare
    locname ClientDB ;
    var Version : int ;
    var foo: int ;
    var Document: process ;
    var DocumentName : str ;
    locname screen
  begin
   while( true ) do
    in( "SUBSCRIBE" , !DocumentName )@ClientLoc;
    # chooses the document, according to DocumentName,
    # and the current version
     if read( DocumentName, !Document , !Version )@ClientDB within 2000 then
      out( "A new request for "+DocumentName+" has been received\n")@screen;
      inp( ClientLoc , DocumentName, !foo )@ClientDB;
      out( ClientLoc , DocumentName, Version )@ClientDB;
      out( "Client "+ClientLoc+
" informations have been successfully stored\n")@screen;
      eval(
         out( DocumentName , true )@self ;
       out( DocumentName, Version )@self;
         eval( Document )@self
      )@ClientLoc;
      out( "Document "+DocumentName+
" has been successfully sent remotely\n")@screen
     else
      out( DocumentName , false )@ClientLoc
     endif
    enddo
  end
```

is executed (see below). Otherwise, when a new client gets connected, its location is stored in the client list. Moreover, process `SubscriptionAgent` is activated Listing 6.2.

When connected, a client can subscribe a document named `docname` by inserting tuple (`"SUBSCRIBE"`, *docname*) in its tuple space. The `SubscriptionAgent` of the client will retrieve such a tuple and, if the document exists, an agent that will store the document is evaluated at client side. The client subscription is also stored at `ClientDB`.

`ClientDB` is a logical locality that is mapped, on the server, to a private physical locality which is known only to the server. It is used to register all the clients and to store information about them (e.g., the documents they subscribed to and



SCREENSHOT 6.1. *A document update*

their current version numbers). Since this locality is not known to the other clients, the server is sure that a client is not able to know the documents installed in another client, and that clients cannot interfere with each other. The secrecy of this locality is obtained by exploiting the locality evaluation mechanism provided by KLAIM. Indeed, that locality will only be mapped to a physical locality dynamically (at run time), through the allocation environment of the server, which is unaccessible by the other nodes.

Each document stored in the server is wrapped inside an agent (`Document` in Listing 6.2). This agent, when executed at a locality, first stores the document into a temp file, hence looks (at `self`) for a tuple of the form:

```
( "OPEN" , app , name )
```

where `app` is the application to use for viewing the document, while `name` is the actual name of the document. For instance, to view a document named `scpe.ps` with `gv`, the following tuple has to be used:

```
( "OPEN" , "gv" , "scpe.ps" )
```

An agent wrapping `document` can receive an update through the tuple

```
("UPDATE",DocName, CurrentVersion, NewVersion)
```

at the private locality. At this point, if the current document is `opened`, the user is notified that a new version of the document is available. Finally, the agent containing the old version of the document produces tuple (`"UPDATE_OK"`, `DocumentName`) and then terminates its execution. The update agent can so store the latest version of the document.

6.2. The update agents. When a new release of a document is installed on the server, by inspecting `ClientDB`, the server will be able to know all the clients that have to be updated, and an update agent is spawned on every such client's site (Listing 6.3).

Upon arrival on the client's site, the update agent (Listing 6.4) first of all verifies that the its version is really new with respect to the one stored locally. If so, it notifies its presence, so that it can be granted permission to update the document. When this update is completed the agent also records that a new version is installed in this node, and then notifies the server that this client has the new version.

LISTING 6.3
The process for spawning an agent on every registered agent.

```
rec CheckUpdate[ ]
 declare
  var DocumentName : str ;
  var Version, ClientVersion : int ;
  var ClientLoc : loc;
  var Document: process;
  var OldDocument: process;
  locname updateKeyb;
  locname ClientDB;
  locname screen
 begin
  while ( true ) do
   in( "STORE" , !DocumentName , !Document )@ClientDB ;
   if inp( DocumentName, !OldDocument, ! Version )@ClientDB then
    Version := Version + 1
   else
    Version := 1
   endif ;
   out( DocumentName, Document , Version )@ClientDB ;
   forall readp( !ClientLoc , DocumentName , !ClientVersion )@ClientDB do
    if Version > ClientVersion then
     eval( UpdateAgent( DocumentName, Version, Document , *self ) )@ClientLoc
    endif
   enddo
  enddo
 end
```

LISTING 6.4
The update agent.

```
rec UpdateAgent[ DocumentName : str, Version : int, Document : process, server : loc ]
 declare
  var CurrentVersion : int;
  var flag: bool
 begin
  in( DocumentName, ! CurrentVersion )@self;
  if ( CurrentVersion < Version ) then
     out( "UPDATE", DocumentName , Version )@self ;
     in( "UPDATE_OK", DocumentName )@self ;
     eval( Document )@self ;
     out( DocumentName, Version )@self ;
     out( "UPDATED", DocumentName, Version, self )@server
  else
     out( DocumentName , CurrentVersion )@self
  endif
 end
```

6.3. Handling client disconnections. Each client can disconnect from the server for work off-line (for instance the client can use a dial-up connection). When off-line, a client will use the local version of the document. Client disconnections are handled by process `DisconnectionManager` (Listing 6.5) that takes care of changing the client status in `clientlist`.

Obviously, when a client works off-line, it cannot receive new updates. However, when a client reconnects to the server, process `AcceptAgent` (Listing 6.1) will execute agent `CheckForClientUpdate` (Listing 6.6) that, by inspecting `ClientDB`, sends to the clients all the documents that have been updated.

7. Implementing Cluedo in X-KLAIM: XKLUEDO. In this section we show how X-KLAIM can be used for developing an agent based implementation of the Cluedo game.

7.1. The game. Cluedo is a crime fiction board game where a set of players have to solve a murder. The board represents a mansion (Figure 7.1 (a)) composed of nine rooms: *Kitchen*, *Ball Room*, *Conservatory*, *Dining Room*, *Billiard*

LISTING 6.5
The agent that handles clients disconnection.

```
rec nodecoord DisconnectionManager[ ]
 declare
  var client : phyloc;
  locname screen, clientlist
 begin
  while (true) do
   disconnected( client );
   out( "Client "+client+" is now disconnected.\n" )@screen;
   in( client , "ON-LINE" )@clientlist;
   out( client , "OFF-LINE" )@clientlist;
  enddo
 end
```

LISTING 6.6
The update procedure after a client connection.

```
rec CheckForClientUpdate[ ClientLoc : loc ]
  declare
    var DocumentName : str ;
    var Version, ClientNum, ClientVersion : int ;
    var Document : process;
    locname ClientDB,screen
  begin
    forall readp( ClientLoc,  !DocumentName, !ClientVersion )@ClientDB do
       read(DocumentName, !Document , !Version )@ClientDB ;
       if (ClientVersion<Version) then
         eval( UpdateAgent( DocumentName, Version, Document , *self ) )@ClientLoc
       endif
    enddo
  end
```

Room, *Library*, *Lounge*, *Hall* and *Study*. Each player represents a character (Miss Scarlet, Professor Plum, Colonel Mustard, Rev. Green, Mrs. White and Mrs. Peacock) that has to discover who killed *Mr. Brown*, the weapon used and the room where murder has taken place. Possible murder weapons are *The Rope*, *The Lead Pipe*, *The Knife*, *The Spanner*, *The Candlestick* and *The Revolver*.

To play the game a pack of cards is used. This consists of 21 cards, each for each room, weapon and character. At the beginning, three cards (one character, one weapon, and one room) are randomly chosen and put into a *special envelope*, so that no-one knows the content of the envelope. The selected cards represent the true facts of the case. The remainder of the cards are distributed among the players.

The aim of the game is to deduce the details of the murder, i. e., the cards in the envelope. To do that, each player announces suggestions to other players, for instance "I suggest it was Mrs. White, in the Library, with the Rope." The other players must then disprove the suggestion, if they can, by showing a card containing one of the suggestion components.

Once a player thinks to know the solution, he/she can make an accusation. The accusing player checks the validity of the accusation by verifying the cards. If the player made a correct accusation, the solution cards are shown to the other players and the game ends. Conversely, if the accusation is incorrect, the player can continue the game but can not submit new accusations (hence he/she cannot win).

7.2. X-KLAIM **implementation.** The idea is to develop an agent-based game where each player executes an agent which migrates among the rooms for implementing a specific game strategy. Please notice that in the standard Cluedo players use dices for moving from a room to another. For the sake of simplicity, we let agent free to move among different rooms.

7.2.1. The pack of cards. Each card is modelled as a tuple composed of two fields $(type, name)$. The former indicates the kind of the card and can assume a value among `"ROOM"`, `"CHARACTER"` and `"WEAPON"`. The latter (*name*) contains the name of the card, e.g., "Conservatory", "Miss Scarlet", "The Revolver", etc. For instance, (`"ROOM"`, `"Kitchen"`) denotes the card corresponding to the *room Kitchen*.

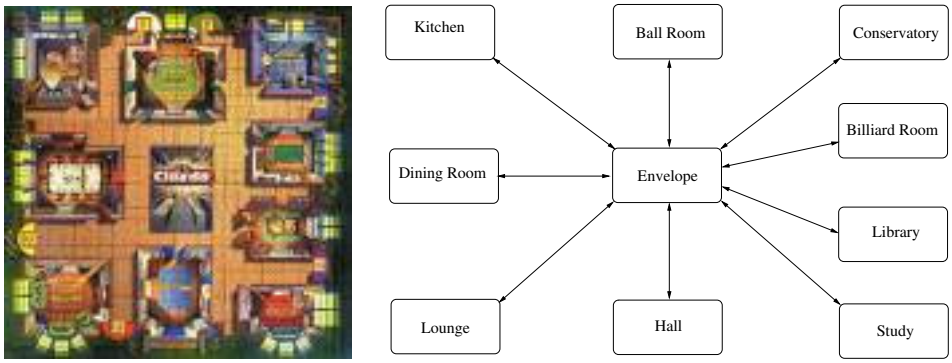


FIG. 7.1. *Cluedo board and* X-KLAIM *topology*

The pack of cards is implemented by means of a tuple space referenced by a local variable of type **ts**. To choose the fact of the case, three cards are retrieved by using X-KLAIM actions and pattern matching. Let `pack` be the tuple space containing all the cards, the following are the instructions initializing variables `room`, `character` and `weapon` that will respectively refer to the place of the murder, the killer and the used weapon:

```
in( "WEAPON" , !weapon )@pack;
in( "ROOM" , !room )@pack;
in( "CHARACTER" , !character )@pack;
```

Each player will receive his/her cards inside a tuple space. Indeed, after a player joins the game, six cards are retrieved from the main `pack` and inserted into a `new` tuple space (`player_pack`). At the end this is sent to the player location (`player_loc`):

```
newloc( player_pack );
out( "COUNT" , 0 )@player_pack;
while (not inp( "COUNT" , 6 )@player_pack) do
  in( !c_type , !c_name )@pack;
  out( c_type , c_name )@player_pack;
  in( "COUNT" , !count )@player_pack;
  out( "COUNT" , count+1 )@player_pack
enddo;
out( "CARDS" , new_board )@player_loc
```

7.2.2. The board. The game board is implemented as an X-KLAIM net containing a node for each room in the mansion. An extra node (named `Envelope`) is used as a central server to which other nodes get connected to take part of the game. The rooms architecture is presented in Figure 7.1 (b).

Incoming connections at `Envelope` are managed by agent `Accept_agent` (Listing 7.1). This agent takes as parameters the cards in the envelope (*room*, *character* and *weapon*) and the main pack of cards.

When a remote node (located at `l`) gets connected with `Envelope`, this agent looks for a tuple indicating the kind of the node. Indeed, two kinds of nodes can get connected with the main node: room nodes, which contain a tuple of the form `("ROOM", name)`, and character nodes, which contain a tuple of the form `("CHARACTER", room)`. If the connected node corresponds to a room, the allocation environment of `Envelope` is updated in order to consider the new binding between the room name with the node location. Please notice that, for the sake of simplicity, we do not consider the case where more nodes try to get connected with the same name.

Agents migrate over the different rooms to acquire the information needed for resolving the case. Agents interact each other by means the tuple spaces located at room nodes by announcing and disproving suggestions. Each agent can make suggestions concerning the room where it is currently located. A suggestion is a tuple of the form (`"SUGGEST"`, *type*, *name*). For instance, tuple (`"SUGGEST"`, `"WEAPON"`, `"Knife"`) in the tuple space located at `Conservatory` indicates that some one suggests that the murder has taken place in the `Conservatory` and that the "Knife" has been used. An agent can disprove a suggestion by removing the corresponding tuple and adding tuple (`"DISPROVE"`, *type*, *name*).

LISTING 7.1
The agent accepting connections at `Envelope`

```
rec nodecoord Accept_agent[ room : str , character : str , weapon : str , board : ts ]
declare
  var l : phyloc;
  var name: str ;
  locname screen, rooms, characters
begin
  while (true) do
    out( "Waiting for new connections...\n" )@screen;
    accept( l );
    out( "New connection established with "+(l)+"\n" )@screen;
    if (inp( "CHARACTER" , !name )@l) then
      out( "PLAYING" , name )@characters;
      out( "Login: "+name+"\n" )@screen;
      eval( Distribute_cards( board , l ) )@self;
      eval( Envelope_agent( room , character , weapon , l , name ) )@self
    else
      if (inp( "ROOM" , !name )@l ) then
        out( "Login: "+name+"\n" )@screen;
        out( name )@rooms;
        bind( name , l )
      else
        out( "Unknown...\n" )@screen
        disconnect( l )
      endif
    endif
  enddo
end
```

To guarantee that one agent at time accesses the tuple space of a `room`, a token is used. Indeed, an agent has to withdraw tuple (`"LOCK"`) from the local tuple space before announces or disproves a suggestion. The token is then reinserted in the tuple space at the end of the operations.

7.2.3. The characters. An X-KLAIM node is associated to each character. The location of these nodes are known only to the agent that accepts the connections. This permits guaranteeing *private* interactions between any character and the main node.

When a character joins the game, i. e., it gets connected with node `Envelope`, `Accept_agent` (Listing 7.1) sends, by using agent `Distribute_cards`, a tuple space containing the cards distributed to the player. After that, process `Envelope_agent` is activated (Listing 7.2).

This agent, which knows the content of the envelope, is waiting for a tuple

```
( "ACCUSE" , !a_room , !a_character , !a_weapon )
```

in the tuple spaces located at the player node.

When a client makes an accusation, `Envelope_agent` verifies its correctness by comparing formal parameters `room`, `character` and `weapon` with retrieved values `a_room`, `a_character` and `a_weapon`. If a player gives the correct accusation he wins the game and the agent registers the player name in the local tuple space. Conversely, if an incorrect accusation has been provided, the agents notifies the player of the mistake and terminates the execution. This way the player cannot win the game anymore. To guarantee that only one accusation for time is considered, tuple `("LOCK")` is used to coordinate the different instances of the agent.

7.2.4. Implementing a player strategy. After a character has received the cards from the `Envelope`, an agent, which implements a specific game strategy, is executed. In Listing 7.3 we present a possible implementation for a simple strategy. This agent migrates over the rooms until it is able to give an accusation. When the agent arrives at a node, it first tries to disprove one of the available suggestions and then formulate a new hypothesis.

This agent has four parameters: `home`, that is a location referring to the player's node, `my_cards`, that is a tuple space containing the player's cards, `suspects` that is a tuple space containing all the suspects (rooms, characters and weapons) that are not yet disproved, and `rooms`, that is a tuple space containing a list of all the rooms. At the beginning `suspects` contains all the cards but those in `my_cards`. Please notice that, an agent can give an accusation when only one card for each type is available in `suspects` tuple space.

LISTING 7.2
The agent that verifies player accusations

```
rec Envelope_agent[ room : str , character : str , weapon : str , l : phyloc , name : str]
 declare
  var a_room : str;
  var a_character : str;
  var a_weapon : str;
  var winner: str;
  locname screen
 begin
  in( "ACCUSE" , !a_room , !a_character , !a_weapon )@l;
  in( "LOCK" )@self;
  if( read( "WINNER" , !winner )@self ) then
    out( "WINNER" , winner )@l
  else
    if ( a_room = room ) and (a_character = character) and (a_weapon = weapon)
    then
     out( "OK" )@l;
     in( "RUNNING" )@self;
     out( "WINNER" , name )@self;
     out( "The winner is "+name )@screen
    else
     out ( "FAIL" )@l
    endif
  endif;
  out( "LOCK" )@self
 end;
```

The agent first retrieves a room, a character and a weapon from `suspects`. Then tests whether other rooms, characters or weapons can be suspected. If there are no other suspects the agent migrates at `home` and makes the accusation. Otherwise, it migrates over the rooms where it makes its suggestions and rejects the ones of other agents.

To migrate to another room each agent first migrates at `Envelope` and then migrates to the locality referenced by name `room`. Indeed, the allocation environment of node `Envelope` stores the binding between the room names and the physical addresses of the corresponding nodes.

When an agent reaches the remote rooms, it first updates the suspect list by removing each card that has been disproved:

```
forall readp( "DISPROVE" , !c_type , !c_name )@self do
  inp( c_type , c_name )@suspects
enddo;
```

After that, the agent tries to disprove one of the available suggestions:

```
flag := true;
while (flag and inp( "SUGGEST" , !c_type , !c_name )@self) do
  if (readp( c_type , c_name )@my_cards) then
    out( "DISPROVE" , c_type , c_name )@self;
    flag := false
  else
    out( "SUGGEST" , c_type , c_name )@self
  endif
enddo;
```

Finally, the agent makes its suggestions by retrieving the corresponding cards from the suspects

```
read( "WEAPON" , !weapon )@suspects;
read( "CHARACTER" , !character )@suspects;
Suggest( weapon , character );
```

Agent `Suspect` is invoked for announcing a suggestion and permits guaranteeing that only a tuple for each suggestion is available in room tuple spaces.

LISTING 7.3
An agent implementing a game strategy

```
rec Player_One[ home : loc , my_cards : ts , suspects : ts ]
 declare
  var room, weapon, character, c_type, c_name : str;
  var flag, try_again : bool;
  var next_room: logloc
 begin
   room := ""; weapon := ""; character := "";
   try_again := true;
   while (try_again) do
    in( "ROOM" , !room )@suspects;
    in( "WEAPON" , !weapon )@suspects;
    in( "CHARACTER" , !character )@suspects;
    # the test below permits verifying if other
    # suspects are available
    if (readp( "ROOM" , !c_name )@suspects or
     readp( "WEAPON" , !c_name )@suspects or
     readp( "CHARACTER" , !c_name )@suspects )
    then
     forall readp( !room )@rooms do
      go@envelope;
      next_room := (logloc) room;
      go@next_room;
      in( "LOCK" )@self;
      flag := true;
      forall readp( "DISPROVE" , !c_type , !c_name )@self do
       inp( c_type , c_name )@suspects
      enddo;
      while (flag and inp( "SUGGEST" , !c_type , !c_name )@self) do
       if (readp( c_type , c_name )@my_cards) then
        out( "DISPROVE" , c_type , c_name )@self;
        flag := false
       else
        out( "SUGGEST" , c_type , c_name )@self
       endif
      enddo;
      read( "WEAPON" , !weapon )@suspects;
      read( "CHARACTER" , !character )@suspects;
      Suggest( weapon , character );
      out( "LOCK" )@self
     enddo
    else
     # In this case the agent can go at home
     # and give the accusation
     try_again := false
    endif
   enddo;
   go@home;
   out( room , character , weapon )@self
 end;
```

8. Conclusions and Related Works. We have presented X-KLAIM, a programming language for implementing distributed applications that can exploit mobile code and run over an heterogeneous network environment. X-KLAIM provides support for moving processes (with strong mobility) and all the code they will need for execution at remote sites. An interesting spin-off of our approach is that, since X-KLAIM is based upon the KLAIM formal model [8], some properties of systems can be formally proved (e.g., in [13] we prove some formal properties of a chat system similar to the one presented in Section 5 and of a mobile agent based software update system). Indeed, a modal logic for KLAIM is being studied [25] and a system to automatically prove KLAIM system properties is under development.

There are currently a number of Java packages, libraries and frameworks that implement functionalities for programming distributed and mobile systems, and that are based on the Linda communication model. In the rest of this section, we review some of them and discuss their relationships with our system.

Jada [22] is a coordination toolkit for Java where coordination and communication among distributed objects is achieved via shared *ObjectSpaces* that are implementations of tuple spaces. Remote access to ObjectSpaces is achieved by specifying the complete IP address and port number, i. e., no locality abstraction is used. Private ObjectSpaces can be dynamically created. No code mobility is supplied by Jada that aims at providing a coordination kernel for implementing more complex Internet languages and architectures.

MARS [18] is a coordination tool for Java-based mobile agents that defines Linda-like tuple spaces programmable to react when accessed by agents. Such a mechanism can be used to control accesses to specific tuples. In X-KLAIM, this is obtained either by using dynamically created private tuple spaces or by adding to the language the capability-based type system presented in [24].

Jini [6] is a connection technology that enables many devices to be plugged together to form a community on a network in a scalable way and without any planning, installation, or human intervention. Each device defines services that other devices in the community may use and drivers that can be downloaded when needed. Jini is developed on top of the *JavaSpaces* [4] technologies, a framework for using Linda-like communication. JavaSpaces introduces some extensions of the Linda original paradigm, such as *event notification*, which allows a process to register its interest in future occurrences of some event and then to receive communication when the event occurs, and *blocking operations with timeouts* and *leasing*, which allows the presence of a tuple in a tuple space, or a notification request, to be granted only for a period of time. Leasing can be obtained also in our language by means of timeouts: a process can sleep for some time (using timeout), and then can take a tuple away from the tuple space (if it is still available). JavaSpaces *transactions* can be programmed in X-KLAIM, by means of dedicated tuples, which represent transaction life time.

IBM *T Spaces* [27] is a network middleware package that supplies tuple space-based network communication with database capabilities; it is implemented in Java by relying on its portability. T Spaces is basically a message processor, in fact a client's view of T Spaces is that of a message center and a message database. A DBMS could be implemented in X-KLAIM by means of a process listening for requests (e.g., SQL strings) passed via tuples, to obtain a similar behavior.

Lime [31] exploits the multiple tuple spaces paradigm [29] to coordinate mobile agents and adds mobility to tuple spaces: it allows processes to have private tuple spaces and to transiently share them. Although in X-KLAIM tuple spaces are bound to nodes and nodes cannot move, processes can have objects of the class `TupleSpace` as data members and, hence, when processes move, `TupleSpace` objects move as well. However, `TupleSpace` objects are never shared and merged automatically.

Systems such as [16, 30, 32, 1], implement strong mobility in Java, by modifying the Java Virtual Machine, to access, save and restore the execution state of threads. However, this solution can jeopardize one of the most desirable advantages of Java: portability across platforms. Indeed, one needs to run the modified version of the JVM in order to use such agents. This is the reason why we preferred not to include strong mobility in KLAVA; however, this feature is available in X-KLAIM and it is implemented on top of KLAVA by means of an appropriate precompilation phase [9].

A feature that is present in systems such as *MARS*, *Lime*, *Sumatra* and *T Spaces*, but not in X-KLAIM, is the ability to react to events such as the insertion of a tuple. This could be programmed by means of a process waiting for a certain tuple, but this does not exactly implement reactions due to the non-determinism in the selection of the process waiting for a tuple.

REFERENCES

[1] A. ACHARYA, M. RANGANATHAN, AND J. SALTZ, *Sumatra: A Language for Resource-aware Mobile Programs*, in Vitek and Tschudin [33], pp. 111–130.

[2] M. ADAMS, J. COPLIEN, R. GAMOKE, R. HANMER, F. KEEVE, AND K. NICODEMUS, *Fault-tolerant telecommunication system patterns*, in Pattern Languages of Program Design 2, J. Vlissides and J. Coplien, eds., Addison-Wesley, 1996, pp. 549–562.

[3] B. G. ANDERSON AND D. SHASHA, *Persistent Linda: Linda + Transactions + Query Processing*, in Proc. of Research Directions in High–Level Parallel Programming Languages, J. P. Banatre and D. Le Metayer, eds., vol. 574 of LNCS, Springer, 1992, pp. 93–109.

[4] K. ARNOLD, E. FREEMAN, AND S. HUPFER, *JavaSpaces Principles, Patterns and Practice*, Addison-Wesley, 1999.

[5] K. ARNOLD, J. GOSLING, AND D. HOLMES, *The Java Programming Language*, Addison-Wesley, 3rd ed., 2000.

[6] K. ARNOLD, B. O'SULLIVAN, R. SCHEIFLER, J. WALDO, AND A. WOLLRATH, *The Jini Specification*, Addison-Wesley, 1999.

[7] L. BETTINI, *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*, PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`

[8] L. BETTINI, V. BONO, R. DE NICOLA, G. FERRARI, D. GORLA, M. LORETI, E. MOGGI, R. PUGLIESE, E. TUOSTO, AND B. VENNERI, *The* KLAIM *Project: Theory and Practice*, in Global Computing. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC 2003, Revised Papers, C. Priami, ed., vol. 2874 of LNCS, Springer, 2003, pp. 88–150.

[9] L. BETTINI AND R. DE NICOLA, *Translating Strong Mobility into Weak Mobility*, in Mobile Agents, G. P. Picco, ed., no. 2240 in LNCS, Springer, 2001, pp. 182–197.

[10] ———, *Mobile Distributed Programming in* X-KLAIM, in Formal Methods for Mobile Computing, Advanced Lectures, M. Bernardo and A. Bogliolo, eds., vol. 3465 of LNCS, Springer, 2005, pp. 29–68.
[11] L. BETTINI, R. DE NICOLA, G. FERRARI, AND R. PUGLIESE, *Interactive Mobile Agents in* X-KLAIM, in Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), P. Ciancarini and R. Tolksdorf, eds., IEEE Computer Society Press, 1998, pp. 110–115.
[12] 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, pp. 32–36.
[13] ———, *Formulae Meet Programs Over the Net: A Framework for Correct Network Aware Programming*, Automated Software Engineering, 11 (2004), pp. 245–288. Special Issue on Distributed and Mobile Software Engineering.
[14] L. BETTINI, R. DE NICOLA, AND R. PUGLIESE, KLAVA*: a Java package for distributed and mobile applications*, Software—Practice and Experience, 32 (2002), pp. 1365–1394.
[15] L. BETTINI, M. LORETI, AND R. PUGLIESE, *An Infrastructure Language for Open Nets*, in Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications, ACM, 2002, pp. 373–377.
[16] S. BOUCHENAK AND D. HAGIMONT, *Pickling Threads State in the Java System*, in Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS), 2000.
[17] P. BUTCHER, A. WOOD, AND M. ATKINS, *Global Synchronisation in Linda*, Concurrency: Practice and Experience, 6 (1994), pp. 505–516.
[18] G. CABRI, L. LEONARDI, AND F. ZAMBONELLI, *Reactive Tuple Spaces for Mobile Agent Coordination*, in Proc. of the 2nd Int. Workshop on Mobile Agents, K. Rothermel and F. Hohl, eds., vol. 1477 of LNCS, Springer, 1998, pp. 237–248.
[19] L. CARDELLI, *Global computation*, in ACM Computing Surveys, 1996. 28(4es), Article 163.
[20] L. CARDELLI, *Abstractions for Mobile Computation*, in Secure Internet Programming: Security Issues for Mobile and Distributed Objects, J. Vitek and C. Jensen, eds., no. 1603 in LNCS, Springer, 1999, pp. 51–94.
[21] N. CARRIERO AND D. GELERNTER, *How to Write Parallel Programs: A Guide to the Perplexed*, ACM Computing Surveys, 21 (1989), pp. 323–357.
[22] P. CIANCARINI AND D. ROSSI, *Jada - Coordination and Communication for Java Agents*, in Vitek and Tschudin [33], pp. 213–228.
[23] R. DE NICOLA, G. FERRARI, AND R. PUGLIESE, KLAIM*: a Kernel Language for Agents Interaction and Mobility*, IEEE Transactions on Software Engineering, 24 (1998), pp. 315–330.
[24] R. DE NICOLA, G. FERRARI, R. PUGLIESE, AND B. VENNERI, *Types for Access Control*, Theoretical Computer Science, 240 (2000), pp. 215–254.
[25] R. DE NICOLA AND M. LORETI, *A Modal Logic for Mobile Agents*, ACM Transactions on Computational Logic, 5 (2004), pp. 79—128.
[26] D. DEUGO, *Choosing a Mobile Agent Messaging Model*, in Proc. of ISADS 2001, IEEE, 2001, pp. 278–286.
[27] D. FORD, T. LEHMAN, S. MCLAUGHRY, AND P. WYCKOFF, *T Spaces*, IBM Systems Journal, (1998), pp. 454–474.
[28] D. GELERNTER, *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems, 7 (1985), pp. 80–112.
[29] D. GELERNTER, *Multiple Tuple Spaces in Linda*, in Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 89), E. Odijk, M. Rem, and J. Syre, eds., vol. 365 of LNCS, Springer, 1989, pp. 20–27.
[30] H. PEINE AND T. STOLPMANN, *The Architecture of the Ara Platform for Mobile Agents*, in Proc. of the 1st International Workshop on Mobile Agents (MA '97), K. Rothermel and R. Popescu-Zeletin, eds., no. 1219 in LNCS, Springer, 1997, pp. 50–61.
[31] G. PICCO, A. MURPHY, AND G.-C. ROMAN, LIME*: Linda Meets Mobility*, in Proc. of the 21st Int. Conference on Software Engineering (ICSE'99), D. Garlan, ed., ACM Press, 1999, pp. 368–377.
[32] M. RANGANATHAN, A. ACHARYA, S. SHARMA, AND J. SALTZ, *Network-aware Mobile Programs*, in Proc. of the USENIX Annual Technical Conf., USENIX, 1997, pp. 91–103.
[33] J. VITEK AND C. TSCHUDIN, eds., *Mobile Object Systems - Towards the Programmable Internet*, no. 1222 in LNCS, Springer, 1997.

Edited by: Henry Hexmoor, Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 11, 2006