KLAVA: a Java package for distributed and mobile applications. User's manual

Lorenzo Bettini

Dipartimento di Sistemi e Informatica, Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
http://www.lorenzobettini.it

July 1, 2004

Contents

1 The Java package KLAVA **3**

2 KLAVA: Basic Concepts and Architecture **3**
2.1 Tuples . . . 4
2.2 Localities . . . 6
2.3 Tuple Spaces . . . 7
2.4 Nets & Nodes: `NetNode` . . . 8
2.4.1 Node functionalities . . . 8
2.4.2 Net functionalities . . . 10
2.4.3 Locality Resolution . . . 14
2.5 Processes & Node Coordinators . . . 15

3 Code mobility in KLAVA **15**

4 Three Example Applications **19**
4.1 A news gatherer . . . 19
4.2 Load balancing . . . 21
4.3 A chat system . . . 23

5 Performance Assessment **24**

6 Privacy in Distributed Tuple Spaces **27**
6.1 Implementation . . . 30
6.2 Programming Examples . . . 31

References **36**

1 The Java package KLAVA

In this document we illustrate the framework KLAVA, a Java package for implementing distributed applications that can exploit mobile code and run over a heterogeneous network environment. KLAVA is based on the KLAIM coordination paradigm with multiple distributed tuple spaces (*Kernel Language for Agent Interaction and Mobility*) [De Nicola *et al.*, 1998]. Thus, the underlying model enables *space uncoupling*, *time uncoupling* and *destination uncoupling*, and *asynchronous*, *associative* and *anonymous* communication. This programming model is suitable for distributed applications, mobile agents, and, more in general, mobile code (for an overview of communication models for mobile agent systems we refer to [Deugo, 2001]). KLAVA also provides support for moving processes and all the code they will need for execution at remote sites.

KLAVA is also used as the run-time system for the programming language X-KLAIM ([Bettini, 2003b; Bettini, 2003a]) in that the X-KLAIM compiler generates Java code that relies on the KLAVA framework. Thus, X-KLAIM can be used to write the highest layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the KLAIM paradigm.

The framework was originally developed in [Bettini, 1998], and presented in details in [Bettini *et al.*, 2002b]; however, the KLAVA framework we present here differs from the previous presentations in that it relies on the hierarchical model of KLAIM introduced in [Bettini *et al.*, 2002a]. Many parts of the framework did not change and a particular attention was observed in order to keep the Java package backward compatible. The crucial part that has changed deals with node architecture and locality management, while all the functionalities related to message exchange and code mobility are essentially unchanged. This document is based on chapters of [Bettini, 2003a].

2 KLAVA: Basic Concepts and Architecture

In this section, we present the classes of the package `Klava`. Some of `Klava` classes can already be used as they are (e.g., the class `Tuple`), while others have to be specialized through inheritance and method overriding (e.g., the class `KlavaProcess`).

Before describing KLAVA we give a very brief introduction to KLAIM (we refer the interested reader to [De Nicola *et al.*, 1998] and to the KLAIM web page, `http://music.dsi.unifi.it`, for more complete descriptions of the formal model).

KLAIM is based on the notion of *locality* and relies on a Linda-like communication model. Linda [Carriero & Gelernter, 1989b; Gelernter, 1985; Gelernter, 1989] is a coordination language with asynchronous communication and shared memory. The shared space is named *tuple space*, a multiset of *tuples*; These are containers of information items (called *fields*). There can be *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 match: a formal field matches any value of the same type, and two actual fields match only if they are identical (but two formals never match). For instance, tuple $(\text{“foo”}, \text{“bar”}, 100 + 200)$ matches with $(\text{“foo”}, \text{“bar”}, !Val)$. After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *Val* (an integer variable) will contain the integer value 300.

In Linda there is only one global shared tuple space; KLAIM extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. There are two kinds of localities: *physical localities* are the identifiers through which nodes can be uniquely identified within a net; *logical localities* are symbolic names for nodes. A reserved logical locality, `self`, can be used by processes to refer to their execution node. Physical localities have an absolute meaning within the net, while logical localities have a relative meaning depending on the node where they are interpreted and can be thought as aliases for network resources. Logical localities are associated to physical localities through *allocation environments*,

represented as partial functions. Each node has its own environment that, in particular, associates `self` to the physical locality of the node.

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**(P)@l: spawns process P for execution at l.

In the original KLAIM model, and thus also in the original KLAVA framework, *nodes* were merely the execution engines for processes and they had to be part of a *net*. A node in a net was accessible by any other node in the same net, once its (physical) locality was known. This led to a *flat* model in that nodes could not contain other nodes, thus subnets and hierarchical nets could not be directly modeled. Finally, while the separation between concrete and symbolic addresses of nodes implemented a sort of dynamism, still this dynamism did not suit open networks well. Indeed, the model had a static flavor that leads to a sort of "closed world": apart from the creation of new nodes, the topology of KLAIM nets could not be changed.

This new version of KLAVA adopts the hierarchical model of KLAIM, presented in [Bettini *et al.*, 2002a; Bettini, 2003a]: mechanisms for dynamically updating nodes' allocation environments and for explicitly dealing with node connectivity are now supplied. Moreover, a new category of processes, called *NodeCoordinators* is available, which, in addition to the KLAIM operations, can execute coordination operations for establishing new connections, for accepting connection requests and for removing connections.

2.1 Tuples

Tuples are sequences of information items called *fields* and are the basic tools for data elaboration and information exchange. There are two kinds of tuple fields: *actual fields* (i.e., expressions, processes, localities, constants, identifiers) and *formal fields* (i.e., variables).

The class `Tuple` includes methods for handling tuples, such as creating tuples, adding elements to a tuple, getting an element of a tuple, etc. A tuple can be created by passing a `Vector` object, containing all tuple elements, to the `Tuple` constructor, such as:

```
Vector v = new Vector() ;
v.addElement( o1 ) ;
v.addElement( o2 ) ;
v.addElement( o3 ) ;
Tuple t = new Tuple( v ) ;
```

or by first creating an empty tuple and then adding elements using the method `add(Object o)`. To make tuple construction easier, a limited number of overloaded constructors is available such as:

```
public Tuple( Object o1 )
public Tuple( Object o1, Object o2 ) ...
```

so that one can create a new tuple simply by writing:

```
Tuple t = new Tuple( o1, o2, o3 ) ;
```

Tuples are anonymous and content-addressable and *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 and two actual fields match only if they are identical (two formals never match).

After matching, the variable of a formal field will get the value of the field it has matched.

Pattern-matching is implemented through the method `match` of the class `Tuple`. `match` takes a tuple as parameter and checks the matching with the current tuple. The method `match` also performs the binding of the formals, in case the matching succeeds.

The interface `TupleItem` can be used for handling tuple fields. Its declaration is as follows:

```
public interface TupleItem extends java.io.Serializable {
  public boolean isFormal() ; // is it a formal?
  public void setValue(Object o) ; // for updating
  public boolean equals(Object o) ; // are they equal?
}
```

`TupleItem`'s methods are used by the matching algorithm. More specifically, `isFormal` is used to test whether a tuple field is a formal, `setValue` to update a formal field with an actual value, and `equals` to test whether two actual fields match. As usual, the semantics of these methods must be specified by the classes that implement the interface. The package `Klava` makes available some wrapper classes for standard data types that implement this interface: `KString`, `KInteger`, `KBoolean` and `KVector`.

It is assumed that a `TupleItem` created with the default constructor (i.e., with no parameters) is a formal. We think that this is better than having a method `setFormal`, which may cause inconsistencies among aliases. Below we provide an example:

```
KString s = new KString() ; // formal declaration
KInteger i = new KInteger() ; // formal declaration
Tuple t1 = new Tuple( s, i ) ;
Tuple t2 = new Tuple( new KString("Hello"), new KInteger(10) ) ;
t2.match( t1 ) ; // true
System.out.println( "s now is : " + s ) ;
System.out.println( "i now is : " + i ) ;
```

Notice that the values of formal fields are automatically updated (by means of the method `setValue`).

Alternatively, a `Class` object can be used for expressing a formal field. After a successful matching, the method `Object getItem(int index)` (indexes start from 0) can be used to retrieve the value bound to a formal field. Thus, the above program fragment can be also rendered as follows:

```
Tuple t1 = new Tuple( String.class, Integer.class ) ;
Tuple t2 = new Tuple( new String("Hello"), new Integer(10) ) ;
t2.match( t1 ) ; // true
System.out.println( "matched string : " + t1.getItem(0) ) ;
System.out.println( "matched integer : " + t1.getItem(1) ) ;
```

The method `match` relies on types, thus, e.g., a `KString` cannot match a `String`. Indeed, the KLAVA matching mechanism is object based, but not object-oriented, in the sense that subtyping is not considered while checking whether two tuple fields match; namely, the static types must be the same. Thus, the internal structure of objects is no further inspected and, in particular, only the method `equals` is employed in order to test whether two actual fields match. There is only one exception that is needed to avoid that the matching mechanism limits the exchange of code and to make it possible to match a tuple containing processes without knowing their concrete classes in advance: actual processes, that are instances of subclasses of `KlavaProcess` (explained later), are retrieved by means of tuple fields of class `KlavaProcessVar` (see Section 2.4.1).

The KLAVA system automatically assigns a unique identifier to each tuple; such an identifier can be considered as a GUID (*Global Unique Identifier*); after the matching, the identifier of the matching tuple is stored in the template used for retrieving a tuple. This is useful when using the same template for retrieving another tuple: once a tuple has been retrieved through a template tuple, the method `resetOriginalTemplate` can be called to reset the template, i.e., its formal fields are initialized to empty values again, and such template can be used for retrieving another tuple. Since the identifier of a tuple that has already been retrieved is stored in the template, the pattern-matching will consider only for tuples not already inspected. This provides an easy mechanism for iterating through a tuple space. Already retrieved tuples are discarded, during the matching, by the method `preMatch`, that can also be redefined in case a finer filtering mechanism is needed (for instance it is redefined by the class `Tuplex` for dealing with encrypted fields, Section 6.1).

In the following example, after the first matching between `t2` and `t1` the template `t1` is reset; trying the matching with `t2` once again will fail, while it will succeed with another matching tuple, `t3`:

```
KString s = new KString() ; // formal declaration
KInteger i = new KInteger() ; // formal declaration
Tuple t1 = new Tuple( s, i ) ;
Tuple t2 = new Tuple( new KString("Hello"), new KInteger(10) ) ;
Tuple t3 = new Tuple( new KString("World"), new KInteger(20) ) ;
t2.match( t1 ) ; // true
t1.resetOriginalTemplate();
t2.match( t1 ) ; // false: already matched
t3.match( t1 ) ; // true
```

2.2 Localities

Localities are the tools that processes can use for referring to nodes of the net. We distinguish between two kinds of localities:

- *physical localities* are 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 to their execution node.

Intuitively, physical localities have an absolute meaning within the whole 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. The association between logical and physical localities is modeled via a function that we call *allocation environment*; each node has an allocation environment that solves the logical localities there used.

There are three classes that handle localities. The abstract class `Locality` is the base class and implements the interface `TupleItem`. The other two classes `LogicalLocality` and `PhysicalLocality` are derived from this base class. A variable that represents a locality should always be declared as a `Locality` so that polymorphism can be used extensively.

In this version of KLAVA, physical localities are strings of the form <IP>:<port>, and they actually represent the Internet address and the port number where the node is listening for incoming connections (Section 2.4.2). In the original version of the framework [Bettini *et al.*, 2002b] physical localities, just like logical localities, were simply names with which nodes used to register themselves at the net server. We have preferred to use this lower level representation in the new version of KLAVA, because it permits an easier handling of connections and communications; moreover the uniqueness of physical localities is also imposed by the uniqueness of the pair <IP>:<port> in the Internet.

2.3 Tuple Spaces

Tuple spaces are multisets of tuples. The class `TupleSpace` includes methods to place tuples in and retrieve tuples from a tuple space. In particular, the methods

```
public boolean in( Tuple t )
public boolean read( Tuple t )
public void out( Tuple t )
```

implement three operations over tuple spaces, namely **in**(t), **read**(t) and **out**(t):

- **in**(t): looks for a tuple t' that matches t. Whenever the 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): differs from **in**(t) only because the matching tuple t' is not removed from the tuple space.
- **out**(t): adds the tuple t to the tuple space.

Two other methods of this class that implement operations over tuple spaces are:

```
public boolean read_nb( Tuple t )
public boolean in_nb( Tuple t )
```

`read_nb` and `in_nb` act like `read` and `in`, but, if no matching tuple is found, they do not block the executing process and simply return `false`. Some versions of Linda also introduce such operations, called **readp** and **inp** [Carriero & Gelernter, 1989a]. These variants are useful when one wants to search for a matching tuple in a tuple space with no risk of blocking. For instance, `read_nb` can be used to test whether a tuple is present in a tuple space.

Below we report an example that uses a `TupleSpace` object:

```
TupleSpace TS = new TupleSpace() ;
KString s = new KString() ; // formal
KInteger i = new KInteger() ; // formal
Tuple t = new Tuple( s, new KInteger(10) ) ;
TS.out( new Tuple(new KString("Hello"), new KInteger(10)));
TS.out( new Tuple( new KInteger(10) ) ) ;
TS.in( t ) ; // withdraws the first tuple
TS.read( new Tuple( i ) ) ; // reads the second one
```

Iteration on a tuple space can be easily implemented by using the same template tuple and the method `resetOriginalTemplate`, as shown in Section 2.1.

Due to network latency and bandwidth, network communications can be quite slow, hence, retrieving information may require more time than one is willing to wait. Moreover, the absence

of matching tuples could block a process executing an **in**/**read** operation. To put upper bounds to the waiting time, a *time-out* can be used. Therefore, "timed" versions of the blocking methods are supplied:

public void read(Tuple t, **long** TimeOut) **throws** KlavaTimeOutException
public void in(Tuple t, **long** TimeOut) **throws** KlavaTimeOutException

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

Boolean versions of operations with timeouts are also supplied that do not throw a `KlavaTimeOutException` exception in case of timeout but simply return **false**:

public boolean read_t(Tuple t, **long** TimeOut)
public boolean in_t(Tuple t, **long** TimeOut)

2.4 Nets & Nodes: `NetNode`

As hinted in Section 2, any node will now play a double role: it is a computational environment for processes and the container of a tuple space, and a *gateway*, or *server*[1], that can manage a subnet of other nodes (clients). Moreover, 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 locality, allocation environments of nodes in this hierarchy are now inspected from the bottom upwards. This resembles name resolution within DNS servers.

We use a single class, `NetNode`, to play the double role of the client and of the server. This class has both the node and the net functionalities. In the previous version of KLAVA, in order to implement a client-server system, such as, e.g., a chat system, both the server and the clients were at the same level (flat model): the only real server was the net server, while the chat server and the chat clients were just clients of the net server. With the hierarchical model, instead, the roles of server and client reflects exactly also in the implementation. The two implementations are depicted in Figure 1. An example of a chat system implemented in KLAVA is presented in Section 4.3.

In the rest of this section we describe the class `NetNode` highlighting its *node* functionalities and its *nets* functionalities. As hinted above, these two functionalities can also be provided together in a single node, acting both as a server and as a client of a higher level net.

2.4.1 Node functionalities

Nodes are the loci where tuples and processes reside; they are also the execution engines for KLAVA processes. The class `NetNode` contains a single tuple space and exports methods for accessing this tuple space. These methods take as parameters a tuple and the locality of the destination node; if the operation refers to the current execution site, it is simply redirected to the local tuple space, otherwise a message will be sent to the (possibly remote) destination node.

[1] In the following the terms *gateway* and *server* will be used interchangeably.

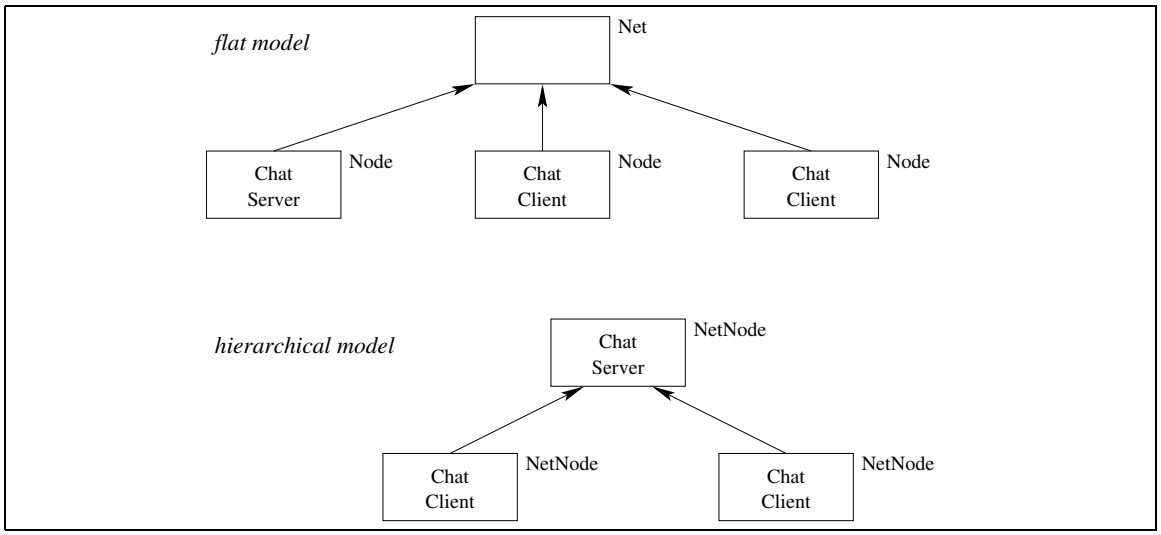


Figure 1: The original flat model (above) and the new hierarchical model (below) implementing a client-server system: a chat system.

```
public void out( Tuple t, Locality l )
public boolean read( Tuple t, Locality l )
public boolean in( Tuple t, Locality l )
```

By using these methods, processes can explicitly address the tuple space where a given operation must be executed. For instance, $\mathbf{out}(t, l)$ means that the tuple t must be placed at the tuple space located at l.

For better programmability, a limited number of overloaded versions of these methods is supplied, so that a tuple does not have to be explicitly built.:

```
public void out( TupleItem i1, Locality l )
public void out( TupleItem i1, TupleItem i2, Locality l )
...
public boolean read( TupleItem i1, Locality l )
public boolean read( TupleItem i1, TupleItem i2, Locality l )
...
```

The class `NetNode` also includes the method

```
public void eval( KlavaProcess P, Locality loc ) throws KlavaException
```

that corresponds to the basic operation $\mathbf{eval}(P, l)$ that spawns process P for execution at node l.

`NetNode` declares two fields: `self` (of class `LogicalLocality`) and `here` (of class `PhysicalLocality`); the latter represents the physical locality of the node within the net. Moreover, each node has an *allocation environment*, namely a sort of partial function that maps logical localities into physical localities. The environment of a `Node` can be initialized with the method `void addToEnv(String logLoc, String phyLoc)` and modified dynamically with the methods `bindloc` and `unbindloc`, explained later. In particular, the allocation environment of a node automatically associates `self` to `here`.

The major difference between $\mathbf{eval}(P, l)$ and $\mathbf{out}(P, l)$ is that $\mathbf{eval}(P, l)$ automatically starts the execution of the process P at the remote site l, while $\mathbf{out}(P, l)$ does not. Indeed, $\mathbf{out}(P, l)$ simply posts a tuple containing the process P at the tuple space of l. The process has to be explicitly retrieved from the tuple space by means of a `KlavaProcessVar` (e.g, by a process executing at l), and explicitly started, as in the following example:

```
KlavaProcessVar PV = new KlavaProcessVar(); // formal
in( PV, self );
eval( PV, self );
```

Notice that **eval** also accepts a `KlavaProcessVar` as a parameter. Moreover, since **out** has a tuple as a parameter, many processes can be delivered to a remote site at once.

In the previous version of KLAVA, accordingly to the original KLAIM model, when evaluating tuples, allocation environments were also used to "close" processes exchanged in communications. Indeed, evaluating a process that occurs in a field of a tuple meant substituting it with its *closure*, namely the process along with the environment of the node where the evaluation is taking place. Hence, a remarkable difference between $\mathbf{out}(P, l)$ and $\mathbf{eval}(P, l)$ was also that **out** was adding the closure of P to the tuple space located at l, while **eval** would send only P, not its closure, for execution at l. This affected the evaluation of logical localities: when a process needs to translate a logical locality into a physical one, first its own allocation environment is used (if it has one) and then, if the translation fails, the environment of the node where the process runs is used. This means that a process delivered with an **out** used a *static scoping* strategy for logical localities while a process remotely spawned with an **eval** used a *dynamic scoping* strategy. Thus, for instance, in the former case, `self` refers to the originating site, while, in the latter case, `self` refers to the current execution site.

This automatic treatment of locality translation and process closure made programming a little bit harder when dynamic scoping was needed (for instance in the scenario of the example presented in Section 4.2). For this reason this automatic mechanism is not the default in the hierarchical KLAIM model (see [Bettini *et al.*, 2002a]), and we supply a finer-grain control on this: the translation of a logical locality and the closure of a process has to be explicitly obtained via, respectively, the methods `getPhysicalLocality` and `closeProcess`. Since this is a big change in the framework, the automatic treatment of these features can be enabled by calling `setAutomaticLocalityEvaluation`, for backward compatibility.

Nodes communicate by means of messages delivered through streams (connected to sockets). The class `NodeMessage` implements messages exchanged in the KLAVA system and has the following structure (the content of a message can be any serializable `Object`):

```
public class NodeMessage implements java.io.Serializable
{
 public PhysicalLocality Source ; // loc of the sender
 public PhysicalLocality Dest ; // loc of the target
 public String ProcessName ; // sender process' name
 public int OpCode ; // operation op code
 public Object Content ; // the content
 ...
}
```

2.4.2 Net functionalities

When a `NetNode` acts as a *gateway* it allows client nodes, belonging to its subnet and possibly executing on different computers, to communicate with each other. In this sense, it can be seen as a multithreaded server that coordinates other KLAVA nodes. To this aim, a different thread (a `NodeHandler`) for each client is executing in the gateway node.

A `NodeHandler` acts as a proxy for the corresponding client node within the gateway node and for this reason the gateway node keeps a table mapping each client node to the corresponding `NodeHandler`. The `NodeHandler` will handle the delivery of the node's messages to other nodes: each client uses a socket actually open directly with its own `NodeHandler` on the gateway. Inter-node communication takes place as follows (see also Figure 2):

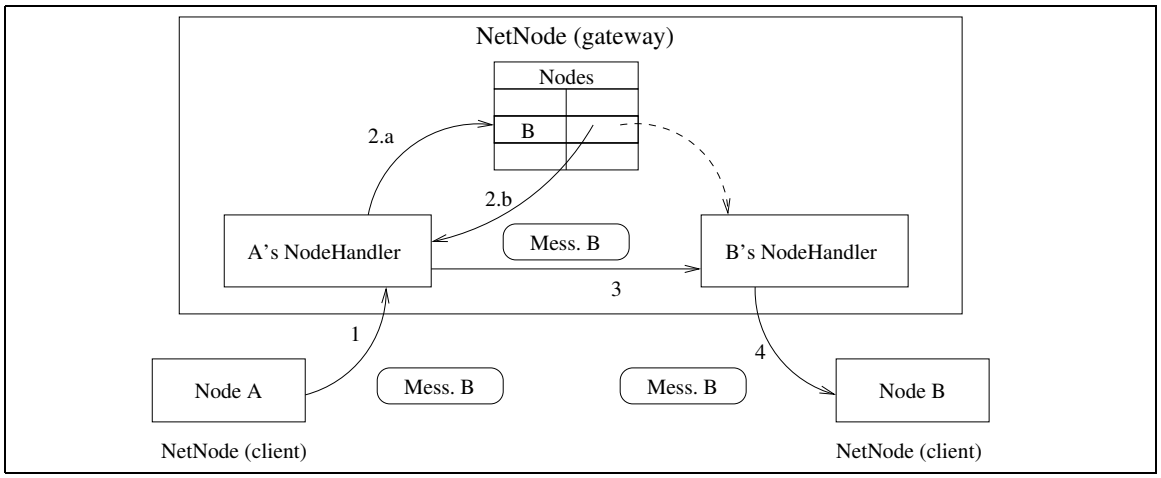


Figure 2: Inter node communication through `NodeHandlers`. The enlarged `NetNode` acts as a *gateway* for the other two nodes (clients).

1. Each node sends the message to its own `NodeHandler`.
2. The `NodeHandler`, after inspecting the message destination field, asks the net server for a reference to the `NodeHandler` of the receiver node (the dashed line in Figure 2 represents a reference).
3. The message is inserted into the message queue of the `NodeHandler` of the destination node.
4. The `NodeHandler` of the destination node takes care of delivering the message to the node it manages.

`NetNode` also provides all the methods for implementing node connectivity. These have to be used when a (client) node wants to enter the subnet managed by another node. Each method for establishing such a relation has also the complementary action that has to be performed by the server to accept a node in its net. All the following methods, implementing *privileged actions*, can be executed only by *node coordinators*, i.e., special processes, with "super user" functionalities, described in Section 2.5.

boolean login(Locality loc) **throws** KlavaException;
void accept(PhysicalLocality loc);

The first method has to be performed at the client node, and succeeds if the server (whose locality is specified as the parameter of **login**) executes an **accept** action. The physical locality of the connected client node is stored in the physical locality that is argument of **accept**. Both operations block the executing processes on each node, until the connection is established. **login** returns **false** in case the connection is not accepted.

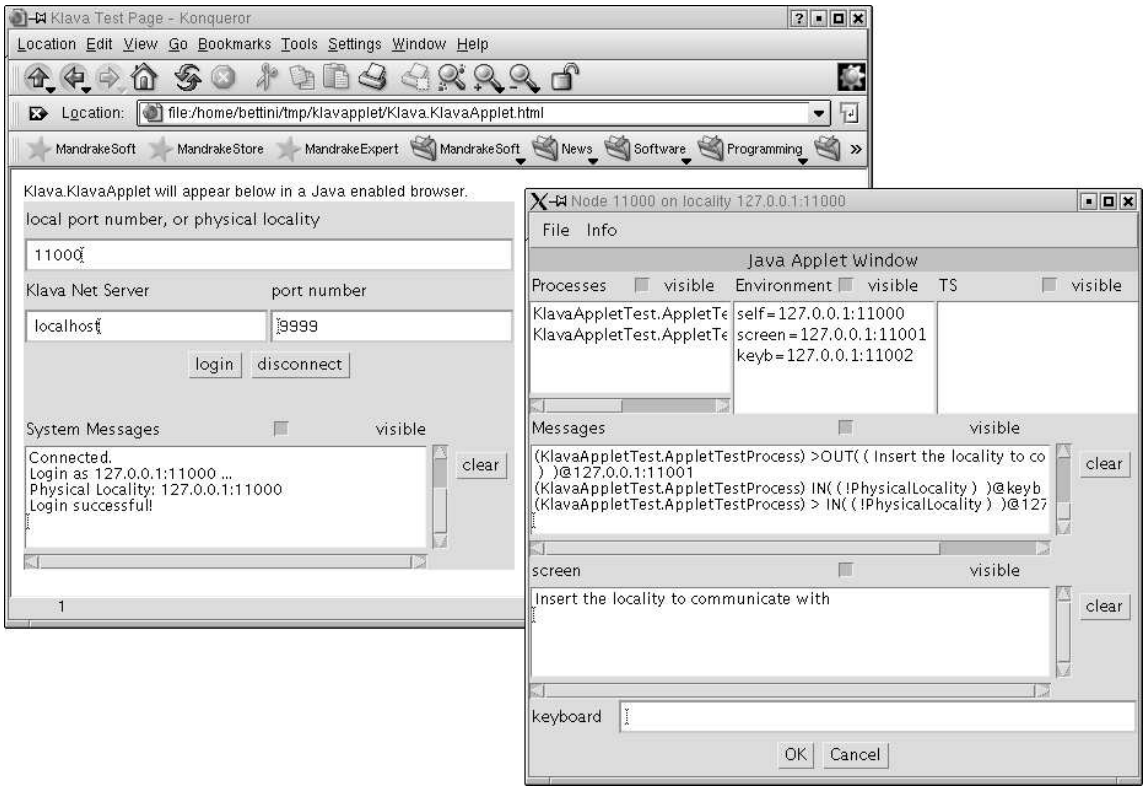
No logical locality is involved in this kind of connection. These are instead used when the following (complementary) methods are invoked:

boolean subscribe(Locality ploc, LogicalLocality lloc) **throws** KlavaException;
boolean register(PhysicalLocality ploc, LogicalLocality lloc);

Once again, the former has to be executed by the client node and the latter by the server. With **subscribe** the client specifies also the logical locality with which it wants to become part of the server's net. Of course the connection is refused in case this logical locality is already used by another already connected client.

A client node can disconnect from a server by using the following methods (depending on the method used for the connection):

Screenshot 2.1 A KLAVA applet in a browser: the applet allows to launch a node that logs in a net server (typically executing at the web server computer).



void logout(Locality loc);
void unsubscribe(Locality loc, LogicalLocality myloc);

In turn, the server can use the methods:

void disconnected(PhysicalLocality loc);
void disconnected(PhysicalLocality loc, LogicalLocality lloc);

for catching the disconnection events above. These methods can also be used for detecting connection failures; both methods return the physical locality of the node that has disconnected, the second method also returns the logical locality the node had subscribed with.

In this scenario communications among nodes belonging to the same subnet take place, through the gateway node. In case of firewalls or network restrictions the access to a remote node may be permitted only through a server. For instance, an applet can only open a network connection towards the computer it has been downloaded from. If on this computer there is a `NetNode` running that is willing to act as a gateway, the applet is still able to indirectly communicate with all the nodes and, possibly, with applets that are part of that net managed by that gateway; An example of a KLAVA applet is available at `http://music.dsi.unifi.it/klava_applet` (see Screenshot 2.1). In this sense, a `NetNode` gateway allows nodes to communicate even if they belong to different restricted domains.

However, when there are no network restrictions, direct connections can still be established in order to use a direct (probably faster) communication between nodes of the same, or different, subnet. In the previous version of KLAVA, this was enabled by a flag in the class `Node`: in this case, the first time a node, say n_1, has to communicate with a node, say n_2, the IP address of n_2

was obtained through the net server and a direct connection between n_1 and n_2 is established (all these operations are performed automatically by KLAVA). Further messages between these two nodes will be exchanged directly through this connection.

In the new version of KLAVA also *direct connections* can be dealt with explicitly, through the following complementary methods:

```
boolean dirconnect(Locality loc) throws KlavaException;
void acceptconn(PhysicalLocality loc);
```

that allow to create a unidirectional direct communication channel. Thus if a node n_1 establishes a direct connection with the node n_2 whenever n_1 sends a message to n_2 it will do this directly, i.e., without passing through a common server. This situation is not symmetric because the direct connection is unidirectional. Should one want a bidirectional *peer to peer* communication, he has to program it explicitly so that, upon accepting a direct connection from a node, also the other way direct connection is established (an example in X-KLAIM is shown in [Bettini, 2003b]). If this control is not needed for direct connections, one can use the method `setDirectConnections(boolean)` to automatically enable direct connections: these will be established automatically the first time a message has to be delivered to a specific physical locality.

The environment of a node can be dynamically modified using the following two methods:

```
void bindloc(LogicalLocality ll, PhysicalLocality pl) throws KlavaException;
void unbindloc(LogicalLocality ll) throws KlavaException;
```

Notice that `NetNode` also provides the method `void addToEnv(String logLoc, String phyLoc)` for adding a mapping to the environment (described in Section 2.4.1), but this method should be used only when initializing the node and can be called only by the subclasses of `NetNode`. On the contrary, **bind** and **unbind** can be called dynamically when the `NetNode` is up and running.

In this version of KLAVA **newloc** is a privileged action and has three variants:

```
PhysicalLocality newloc() throws KlavaException;
PhysicalLocality newloc(NodeCoordinator P) throws KlavaException;
PhysicalLocality newloc(NodeCoordinator P, int port, String classname) throws KlavaException;
```

The first variant simply returns the physical locality of the newly created node. The second variant also takes as argument a node coordinator to be installed in the newly created node: since **newloc** does not automatically logs the new created node in the net of the creating node, this second variant allows to install a node coordinator in the new node that can perform this action (or other privileged actions). As explained in Section 2.5, this is the only way of installing a node coordinator on another node, since node coordinators, due to security reasons cannot be sent for evaluation to another node. The third variant of **newloc** takes two additional arguments: the port number where the new node is going to be listening, and the name of the Java class to be used for instantiating the new node. Notice that the port number also determines the physical locality of the newly created node, since the IP address will be the same of the creator node.

As hinted before, the physical locality of a node is made of the IP address and a port number. This port number identifies the port where the node installs a Java `ServerSocket` listening for incoming connection requests. This is managed by a concurrent thread in the node, a `ConnReqHandler`. The connection requests have to follow a protocol that is quite simple: once `ConnReqHandler` receives a connection request, it reads a string from the connecting client that identifies the kind of connection that is being established (e.g., login, subscribe, etc.); if there is a node coordinator registered in the node for that specific connection type, `ConnReqHandler` delegates the management of that request directly to that node coordinator, and keeps on listening for other incoming connection requests. Otherwise the connection is simply refused. A snippet of

```
public void run() {
 while (true) {
   Socket socket = serversocket.accept();

   ObjectInputStream objIStream = new ObjectInputStream(socket.getInputStream()) ;
   ObjectOutputStream objOStream = new ObjectOutputStream(socket.getOutputStream()) ;

   // read protocol identification string
   Object next_token = objIStream.readObject();
   boolean handled_protocol = false;
   if (next_token instanceof String) {
    TupleSpace ts = proto_table.handle_protocol((String) next_token);
    if (ts != null) {
     Tuple tuple = new Tuple(socket, objIStream, objOStream);
     ts.out(tuple);
     handled_protocol = true;
    }
   }

   if (! handled_protocol)
    socket.close();
 }
}
```

Listing 2.1: The method `run` of `ConnReqHandler`.

such code is shown in Listing 2.1. Notice that internal tuple spaces are used for synchronizing the `ConnReqHandler` and the node coordinators listening for specific requests. We used this technique often, because a tuple space implements the basic functionalities for interprocess communication.

The rest of the connection protocol is left to the node coordinator in charge of that request. Each protocol will require a different exchange of messages, e.g., a client that uses **subscribe** will have to provide the logical locality with which it wants to become part of the net, while for a **login** no logical locality is needed, but only the port number. The IP address has not to be communicated by the client: the server can automatically retrieve it from the socket, but the port has to be explicitly provided, because it is the one where the main `ServerSocket` of the client is opened, and it is not the same with which it opened the socket to the server.

2.4.3 Locality Resolution

The hierarchical KLAIM model is also characterized by the way logical localities are resolved: in order to evaluate locality names, whenever s_1 is logged in s_2, if a locality cannot be resolved by just using the allocation environment of s_1, then the allocation environment of s_2 (and possibly that of nodes to which s_2 is logged in) is also inspected. Thus, in order to find the mapping for a locality, allocation environments of nodes in this hierarchy are inspected from the bottom upwards. In particular, when a node is not able to solve a logical locality, it basically sends a message to the gateway it is connected to and waits for the answer; the gateway in turns may have to rely on its own gateway if it cannot solve it itself. The whole operation fails when a node in the hierarchy is not able to solve a logical locality and it is not connected to any gateway.

In particular, in this implementation of KLAVA, we only considered nets structured as *trees*; the model described in [Bettini *et al.*, 2002a] allows also *graphs* (actually, directed acyclic graphs). The choice of considering only tree structures makes the overall system simpler and does not raise issues such as "which way to chose for resolving a name, when a node is connected to more than one server?". A future enhancement of KLAVA could be that of dealing also with these more complex hierarchical structures, possibly by providing the programmer with customizable policies for choosing the resolution path.

Gateways are essential for communication: apart from direct connections, two nodes are guar-

anteed to interact only if there exists a node that acts as gateway for both. In order to implement this in an efficient way, a gateway has to keep track of all the nodes that are connected to it, and recursively of all the nodes for which its connected nodes act, in turn, as gateway. Thus when a server node accepts another node in its own subnet, it propagates the physical locality of the new client to the gateway it is connected to (if there is one). This propagation mechanism allows the hierarchy to be up-to-date; of course disconnections have to be propagated as well. Notice however that propagations always take place from the client to the server, and never the other way round.

2.5 Processes & Node Coordinators

Processes are the basic computational units. The class `KlavaProcess` is an abstract class that must be specialized to create processes. The derived classes must implement the method `execute` declared as follows:

abstract public void execute() **throws** KlavaException

This method will be invoked when a process is executed (just like `run` for threads). A process must be executed within a node, which will be its *execution environment*. `KlavaProcess` also offers all the methods to access tuple spaces; these methods transparently call the corresponding methods of the class `NetNode`.

In the new hierarchical model of KLAIM, a new category of processes is introduced that, apart from the standard operations, can also perform the privileged actions dealing with node connectivity implemented by the methods presented in Section 2.4.2. For these privileged processes KLAVA provides a subclass of the class `KlavaProcess`, namely `NodeCoordinator`. As required by the model, due to security reasons, node coordinators cannot migrate, and cannot be part of a tuple. Of course, in order to guarantee better programmability, this rule is slightly relaxed: a node coordinator can perform the **eval** of a node coordinator, provided that the destination is `self`.

Since processes and node coordinators have to forward the KLAIM actions to the node they are currently executing on, they need to store a reference such node. In order to force the distinction between standard processes (that can only perform a limited number of actions) and node coordinators (that can also execute privileged actions), the reference is indeed a reference to a *proxy* that has a reduced interface when it is stored inside a standard process: a `NodeProxy` is stored inside a standard process, while a "more powerful" `NetNodeProxy` is stored in a node coordinator. Notice that since a proxy, and not a direct reference to the actual node, is stored, the process cannot call a method that is not allowed to call, since the proxy itself does not provide it: misuses, thanks to the different interfaces of proxies, are caught at compile time.

A number of standard node coordinators is supplied by KLAVA, that simply perform the connection actions, such as, `LoginNodeCoordinator`, `SubscribeNodeCoordinator`, etc., and that performs the corresponding complementary actions, such as, `AcceptNodeCoordinator`, `RegisterNodeCoordinator`, etc. In particular the old KLAVA classes `Node` and `Net` are still supplied in the framework, implemented as subclasses of `NetNode` that simply execute a `LoginNodeCoordinator` and an `AcceptNodeCoordinator`, respectively. In the class `Net` such coordinator executes an infinite loop, always accepting new login requests. This also enables the old KLAVA applications to seamlessly scale to the new hierarchical model.

3 Code mobility in KLAVA

In KLAVA, processes can be sent as part of a message and executed at the destination site, where however their Java classes, i.e., their code, may be unknown. It might then be necessary to make such code available for execution at remote hosts; this can be done basically in two different ways:

- *automatic* approach: the classes needed by a process are collected and delivered together with the process;

- *on-demand* approach: when a Java class is needed by the remote computer that received a process for execution, it is requested to the server that did send the process.

We follow the automatic approach because it complies better with the mobile agent paradigm: during a migration, an agent takes with it all the information that it may need for later executions. The drawback of this approach is that code that may never be used by the mobile agent or that is already provided by the remote site is also shipped. However, our choice has the advantage of simplifying the handling of *disconnected operations* [Park & Reichl, 1998]: the agent owner does not have to stay connected after sending the agent and can connect later just to check whether his agent has terminated. This may not be possible with the on-demand approach: the server that sent the process must always be on-line in order to provide the classes needed by remote hosts.

Therefore, a process must be sent along with its class binary code, and with the class code of all the objects the process uses. Obviously, only the code of user defined classes has to be sent, as the other code (e.g., Java and `Klava` classes) is common to every KLAVA application. This guarantees that classes belonging to `java` sub-packages are not loaded from other sources (especially, the network); this would be very dangerous, since, in general, such classes have many more access privileges.

All the nodes that are willing to accept remote processes (due to security issues, a node may refuse accepting remote processes for execution) must have a custom *class loader*: a `NodeClassLoader` supplied by the KLAVA package. When a process is received from the network, before using it, the node must add the class binary data (received along with the process) to its class loader's table. During process execution, whenever a class code is needed, if the class loader does not find the code in the local packages, then it can find it in its own local table of class binary data.

The names of user defined classes can be retrieved by means of class introspection (*Java Reflection API*). Just before dispatching a process to a remote site, a recursive procedure is called for collecting all classes that are used by the process when declaring: data members, objects returned by or passed to a method/constructor, exceptions thrown by methods, inner classes, the interfaces implemented by its class, the base class of its class.

This procedure is implemented by some methods of the class `KlavaProcess` presented in Listing 3.1 and 3.2 in a simplified form. Notice that this procedure recursively calls itself. Mutual dependencies are handled by checking whether a class has already been collected; thus a class is never inspected more than once.

When extending `KlavaProcess`, there is an important detail to know in order to avoid runtime errors that would take place at remote sites and would be very hard to discover: Java Reflection API is unable to inspect local variables of methods. This implies that if a process uses a class only to declare a variable in a method, this class will not be collected and thus, when the process executes that method on a remote site, a `ClassNotFoundException` may be thrown. This limitation is due to the specific implementation of Java Reflection API, but it can be easily dealt with, once the programmer is aware of the problem. The programmer can explicitly add a class to the collection by calling the method `addUsedClass` in Listing 3.2; all the details of code marshaling are still handled by KLAVA and are transparent to the programmer. The X-KLAIM compiler will generate code for processes in such a way that all the classes used by the process will successfully be collected by KLAVA.

Once these class names are collected, their byte code is gathered in the first node from which the process was sent, and packed along with the process in a `KlavaProcessPacket` object. Notice that the process (its variables) is written in an array of bytes and not in a field of type `KlavaProcess`. This is necessary because otherwise, when the packet is received at the remote site and read from the stream, the process would be deserialized and an error would be raised when any of its specific classes is needed (indeed, the class is in the packet but has not yet been read). Instead, by using our representation, we have that, first, the byte code of process classes is read from the packet and stored in the class loader table of the receiving node; then, the process is read from the byte array; when process classes are needed, the class loader finds them in its own table. Thus, when a node receives a process, after filling in the class loader's table, it can simply

```
protected void getUsedClasses( Class c ) {
  if (c == null || ! addUsedClass( c ))
    return ;

  Field[] fields = c.getDeclaredFields() ;
  Constructor[] constructors = c.getDeclaredConstructors() ;
  Method[] methods = c.getDeclaredMethods() ;
  int i ;

  for( i = 0 ; i < fields.length ; i++ )
    getUsedClasses( fields[i].getType() ) ;

  for ( i = 0 ; i < constructors.length ; i++ ) {
    getUsedClasses( constructors[i].getParameterTypes() ) ;
    getUsedClasses( constructors[i].getExceptionTypes() ) ;
  }

  for ( i = 0 ; i < methods.length; i++ ) {
    getUsedClasses( methods[i].getReturnType() ) ;
    getUsedClasses( methods[i].getParameterTypes() ) ;
    getUsedClasses( methods[i].getExceptionTypes() ) ;
  }

  getUsedClasses( c.getDeclaredClasses() ) ;
  getUsedClasses( c.getSuperclass() ) ;
  getUsedClasses( c.getInterfaces() ) ;
}

protected void getUsedClasses( Class[] classes ) {
  for (int i = 0 ; i < classes.length ; ++i)
    getUsedClasses (classes[i]) ;
}
```

Listing 3.1: The methods to collect the classes used by a process

```
protected boolean addUsedClass( Class classVar ) {
  String className = classVar.getName() ;
  if ( isUsefulClass( className ) && ! usedClasses.containsKey( className ) ) {
    usedClasses.put( className, getClassBytes( className ) ) ;
    return true ;
  }
  return false ;
}

final protected boolean isUsefulClass( String className ) {
  boolean result ;
  result =
    (className.equals( "void" ) || className.equals( "int" ) || className.equals( "char" ) ||
     // ... the same for every base type ...
     className.startsWith("java.") || className.startsWith("Klava.")
    ) ;
  return (! result) ;
}
```

Listing 3.2: The methods to collect the classes used by a process (continued)

deserialize the process, without any need of explicit instantiation. The point here is that classes are always stored in the class loader's table, but they are linked (i.e., actually loaded) on-demand.

The byte code of the classes used by a migrating process is retrieved by the method `getClassBytes` of the class loader (as shown in Listing 3.2): at the node from where the process is first sent, the byte code is retrieved from the local file system, but when a process at a remote site has to be sent to another remote site, the byte code for its classes is obtained from the class loader's table of the node. This strategy, for sending and receiving code, is similar to a *page on-demand* mechanism in an operating system: first the class loader table is filled in with the code for all the classes that a remote process may need, then, when a class is needed, the class loader can load the class by taking the code from its table.

According to the requirements made on the run-time support, code mobility may also be classified as follows [Cugola *et al.*, 1997; Hohlfeld & Yee, 1998]:

- *weak mobility*: code coming from a different site can be dynamically linked;
- *strong mobility*: a thread can move its code and execution state to a different site and resume its execution on arrival;
- *full mobility*: in addition to strong mobility, the whole state of the running program is moved, and this includes all threads' stacks, namespaces (e.g., I/O descriptors, file-system names) and other resources, so that migration is completely transparent.

Full mobility can be considered orthogonal to mobile agents and requires a strong support from the operating system layer. Strong mobility is the notion of mobility that best fits in with the classical concept of mobile agent: the execution state of a migrating agent is suspended, and its stack and program counter are sent to the destination site, together with the relevant data; at the destination site, the stack of the agent is reconstructed and the program counter is set appropriately, i.e., to the first instruction after the migration action. Instead, weak mobility does not meet the intuitive idea of mobile agent, because automatic resumption of execution thread is one of the main features of mobile agents (it exalts their autonomy).

As for the kind of mobility supplied by our framework, KLAVA only provides *weak mobility*. In general, all those systems based on Java, implement only weak mobility; this is due to the fact that Java does not permit dynamic inspection of the byte code stack and this makes impossible to save the execution state for later use. For this reason, also KLAVA can only supply *weak mobility* of agents. X-KLAIM supports strong mobility via a preprocessing performed by the compiler [Bettini & De Nicola, 2001].

In addition to agent mobility, KLAVA also permits moving single parts of code: a single class can be sent to a remote site by building a generic wrapper for it, which implements the `TupleItem` interface, as in the following example:

```
class KCode implements TupleItem {
 protected byte[] code;
 public KCode(String classname) {...}
 public Class get_class() {...}
 // implementation of methods of TupleItem interface
 ...
}
```

The constructor takes the name of the class and stores its byte code in the field `code` (e.g., by using the method `getClassBytes` as in Listing 3.2). In this way a class can be sent to a remote site as follows:

```
out( new KCode("mypackage.myclass"), l );
```

At the remote site this class can be retrieved with the following instructions (`get_class` will interact with the local class loader and create a `Class` object):

```
KCode kcode = new KCode(); // formal
in( kcode, self );
Class c = kcode.get_class();
```

The on-demand approach to agent mobility can be implemented by exploiting single class mobility: agents are sent without their classes; then, at the remote site, when a class for an agent is needed it can be obtained by using a communication protocol exploiting the previous class `KCode`.

Downloading code from the net exposes the executing machine to security risks: the downloaded code could execute dangerous operations (maliciously or due to programming errors) that may tamper other processes or the overall system. To cope with this problem, we implemented a `KlavaSecurityManager` that, if activated by the node, does not allow processes downloaded from the net or sent by remote nodes to execute operations on critical system resources. Implementations of further security mechanisms that rely on the new Java security model [Gong, 1999].

In Section 6 we present an extension of KLAVA with cryptographic primitives for encrypting and decrypting tuple fields and the original Linda operations have been extended for handling encrypted data. The extended framework allows mobile agents, which are not supposed to carry private keys with them when migrating, to collect encrypted data while executing on remote sites, and decrypt them safely when back at the home site.

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

4 Three Example Applications

In this section we present three programming examples that rely on mobility and distribution. The first example concerns a *news gatherer* that exploits mobile agents for retrieving information on remote sites; the second example implements a *load balancing system* that dynamically redistributes mobile code among several processors; the last example is a simplified *chat system*. The main purposes of these examples, whose core implementation parts are reported in some code snippets throughout the following sections, is showing that, by using KLAVA, dealing with mobility and communications among distributed processes boils down to a few method calls, since KLAVA takes care of all the low level details of code mobility and distributed synchronization. In the first two examples we do not show the login phase, since it is not relevant in those contexts. Instead, in the chat system example, we show the subscription phase.

4.1 A news gatherer

In this section we show how to program in KLAVA a *news gatherer* that relies on mobile agents for retrieving information on remote sites. We assume that some data is distributed over the nodes of a KLAVA net and that each node either contains the information we are searching for, or the locality of the next node to visit in the net (Figure 3 depicts the overall architecture of the system). A slightly different version of this scenario is implemented in X-KLAIM in [Bettini, 2003b].

The implementation in KLAVA is reported in Listing 4.1. The agent *NewsGatherer* uses a timeout to test for the presence of the tuple containing the information: if this is not found within two seconds, the locality of the next node to visit is retrieved and a new instance of the agent is remotely spawned there by means of an **eval**. If the information is found, the agent communicates the result to its owner and terminates.

By using KLAVA, to spawn a new process to a remote site, it suffices to just invoke **eval** with the appropriate arguments: the underlying system will take care of serializing the process through

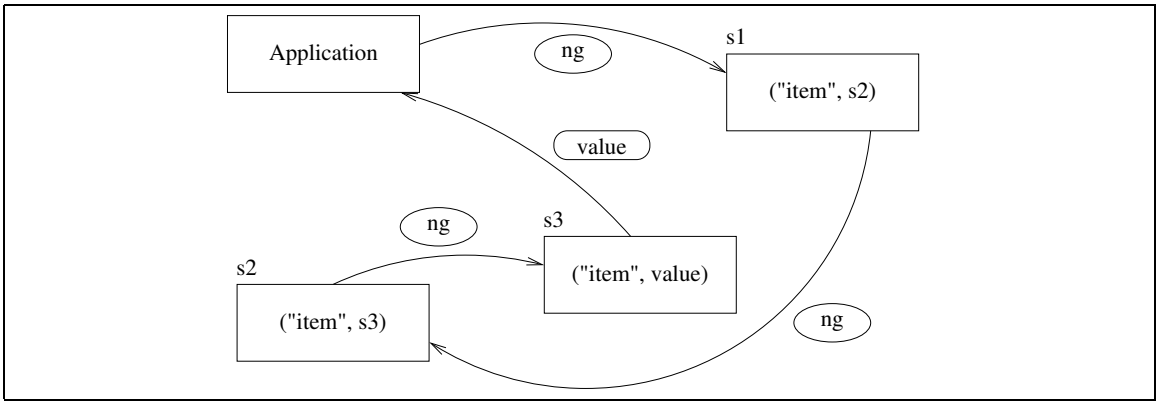
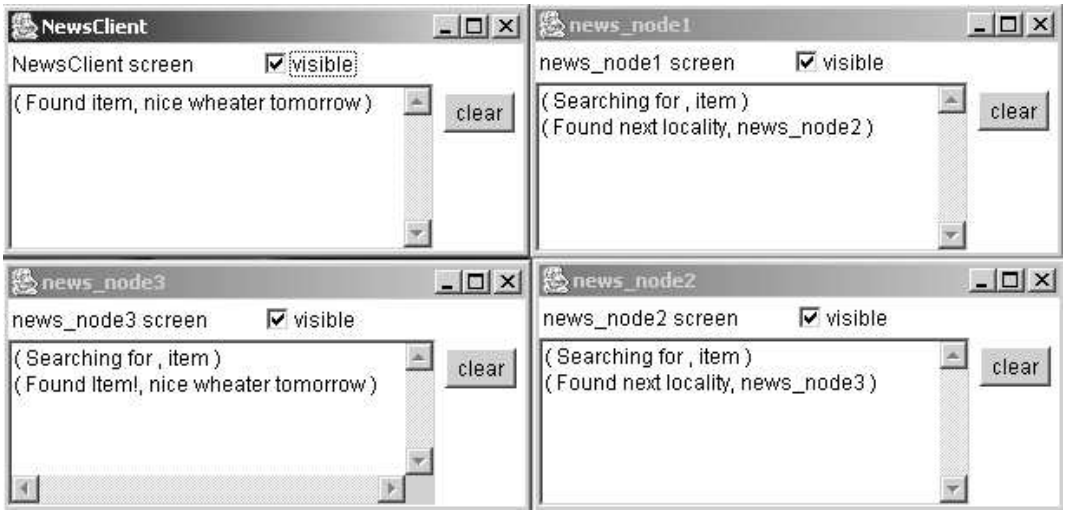


Figure 3: News gatherer (ng is the news gatherer agent)

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

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

  public void execute() throws KlavaException {
    itemVal = new KString() ;
    Print( "Searching for ", item ) ;
    try {
      read( item, itemVal, self, 2000 ) ;
      Print( "Found Item!", itemVal ) ;
      out( itemVal, retLoc ) ;
    } catch (KlavaTimeOutException e) {
      Locality nextLoc = new PhysicalLocality() ;
      read( item, nextLoc, self ) ;
      Print( "Found next locality", nextLoc ) ;
      eval( new NewsGatherer( item, retLoc ), nextLoc ) ;
    }
  }
}
```

Listing 4.1: KLAVA implementation of the news gatherer

Screenshot 4.1 A news gatherer agent visiting three nodes



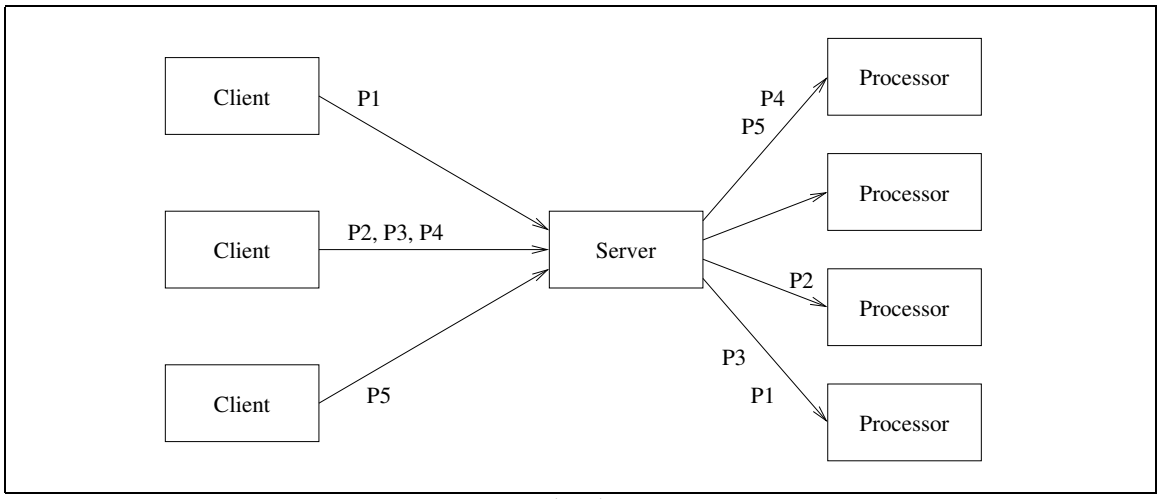


Figure 4: Load Balancing System

the network together with the code of all its classes and the values of its fields. Screenshot 4.1 shows an agent that visits three nodes before finding the wanted information.

4.2 Load balancing

In this second scenario, 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 (Figure 4). This is determined by using the *Leaky Bucket of Credits* pattern [Adams *et al.*, 1996]: when entering the net managed by the load balancing server, 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. The server may exhaust all credits; in that case it waits until it is notified that new credits have arrived.

When a processor receives a process, it immediately starts executing the process (in a parallel thread) and sends a credit back to the server (represented by the locality `processorServer`). Indeed, the Leaky Bucket Of Credits pattern 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 behavior is implemented by the code fragment in Listing 4.2 that shows the process in the processor node

```
class ExecutorProcess extends KlavaProcess {
 ...
 public void execute() throws KlavaException {
  KString execute = new KString("EXECUTE") ;
  KlavaProcessVar P ;
  KString credit = new KString("CREDIT") ;

  while (true) {
   P = new KlavaProcessVar() ;
   in( execute, P, self ) ;
   eval( P, self ) ;
   out( credit, processorServer ) ;
  }
 }
}
```

Listing 4.2: The `ExecutorProcess`

```
class ProcScheduler extends KlavaProcess {
 protected Hashtable processors ;

 public ProcScheduler( Hashtable tab ) {
  processors = tab ;
 }

 public void execute() throws KlavaException {
  KString execute = new KString( "EXECUTE" ) ;
  KlavaProcessVar P ;
  Locality loc ;
  while ( true ) {
   Tuple t = new Tuple( execute, P ) ;
   in( t, self ) ;
   loc = getHighestCredit() ;
   out( t, loc ) ;
  }
 }
}
```

Listing 4.3: The `ProcScheduler`

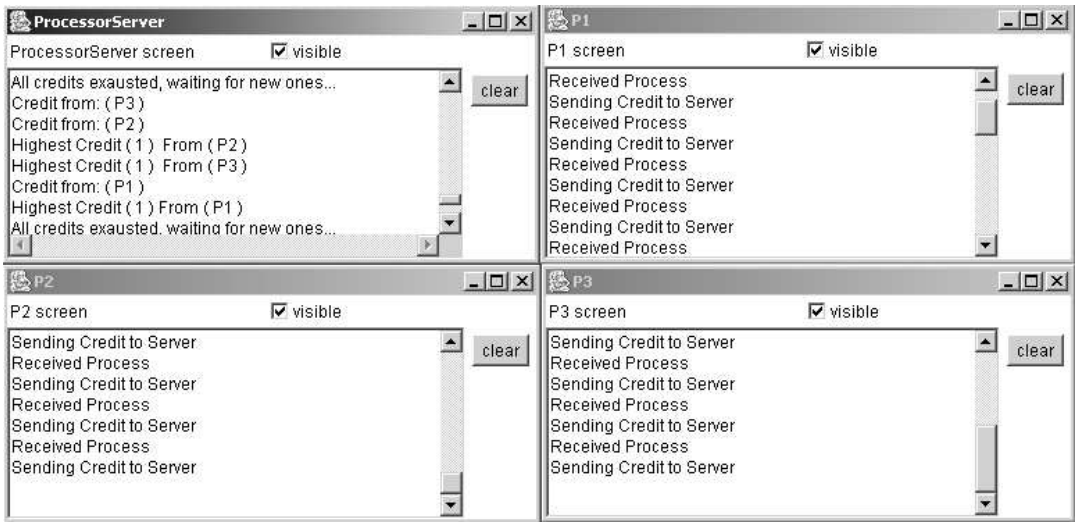
taking care of receiving the processes to be executed. This process simply waits for a tuple of the form `("EXECUTE", P)`, where `P` is a formal obtained by creating a brand new `KlavaProcessVar`, and, by means of **eval**, spawns the received process for execution at the current site. A credit is sent to the server by means of **out**.

The server executes a `ProcScheduler` process that, as shown in Listing 4.3, whenever it receives a process sent by a client, delivers the received process to the idlest processor. The table `processors` is passed to the constructor of the scheduler process and it is updated by another concurrent process (not shown here) that receives tuples of the form `("CREDIT")` from the processor nodes. The locality to which the process can be sent is computed by the method `getHighestCredit` (also not shown here) that, in case all credits for all processors are exhausted, will block the execution of the `ProcScheduler` until new credits arrive.

We would like to point out that the main structure of the application is represented by these few operations that, nonetheless, permit process sharing in a distributed environment. Screenshot 4.2 shows a `ProcessorServer` and three `ProcessorNodes`.

In KLAVA, the main computational unit is represented by a `KlavaProcess`, so a process is assigned to a processor as a whole and once for all: no further load balancing is performed after this assignment. Further decompositions and assignments would require that the framework supplies

Screenshot 4.2 A `ProcessorServer` and three `ProcessorNodes`



means for strong mobility (see Section 3) and for *objective mobility* [Cardelli & Gordon, 1998] (the process does not migrate autonomously, but it is moved by the execution environment), which Java does not supply. However, even with this limitation, this architecture can still be employed in client-server based distributed systems, where the server has to balance the number of connections and communications among several nodes it manages: the client issues a connection request to the server that then picks a node and charges it with the task of interacting with that client.

Let us observe that a process has to be delivered to the server by means of an **out**, since **eval** would automatically starts the execution of the process at the destination node, and thus there would be no easy way of redirecting such process to a specific processor. In the previous versions of KLAVA the closure of such process would have been actually delivered, due to the automatic evaluation mechanism, and thus, the process would have not been able to access the actual executing site by means of `self` (that would be bounded to the starting client site). This shows that the removal of this automatic evaluation mechanism in the new version of KLAVA makes programming these kinds of application much easier.

The overall architecture of this load balancing system is based on a *push* model, in that the server delivers the processes to be executed to a chosen processor node. An alternative implementation could be based on a *pull* model: a processor node, when idle, asks the server for a process to be executed. This architecture can be employed to develop systems similar to *SETI@home* [Korpela *et al.*, 2001] that uses Internet-connected computers in the *Search for Extraterrestrial Intelligence* (*SETI*): users that want to help the project can install this software that downloads data to be analyzed from the server when the computer is idle (for instance when the screen saver starts).

4.3 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. Though this example does not deal with mobile code, it shows how to use KLAVA to implement distributed applications that can communicate through distributed and located tuple spaces. An X-KLAIM version of this chat system is presented in [Bettini, 2003b] and in Section 6.2 a version of the KLAVA chat system is implemented by exploiting cryptography primitives.

The system consists of a `ChatServer` and many `ChatClients`. A client that wants to enter the chat must subscribe at the chat 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 class `ChatServer` derives from the class `NetNode` and executes two node coordinators,

```
public void handle_subscriptions() {
  while (true) {
    LogicalLocality nickname = new LogicalLocality();
    PhysicalLocality clientloc = new PhysicalLocality();
    register(clientloc, nickname);
    addClient(nickname, clientloc);
  }
}
```

```
public void handle_unsubscriptions() {
  while (true) {
    LogicalLocality nickname = new LogicalLocality();
    PhysicalLocality clientloc = new PhysicalLocality();
    disconnected(clientloc, nickname);
    removeClient(nickname, clientloc);
  }
}
```

Listing 4.4: Implementations of two node coordinators in the chat server that manage, respectively, clients entering and exiting from the chat.

```
public void handleOUT( NodeMessage message ) {
  if ( message.Content instanceof Tuple ) {
    Tuple t = (Tuple) message.Content ;
    if ( t.getItem(0) instanceof KString ) {
      KString command = (KString)t.getItem(0) ;
      if ( command.equals("MESSAGE") ) {
        broadcast(messageCode,(KString)t.getItem(1), message.Source) ;
      } else if ( command.equals("PERSONAL") ) {
        broadcast(personalCode, (KVector)t.getItem(1), (KString)t.getItem(2), message.Source) ;
      }}}
}
```

Listing 4.5: Redefinition of method `handleOUT`

whose main code is shown in Listing 4.4, that manage the new subscriptions to the chat and the corresponding clients leaving the chat, by using the methods `register` and `disconnected` explained in Section 2.4.2. The internal methods `addClient` and `removeClient`, not shown here, also take care of notifying all the connected clients that a new client entered the chat or that an existing client left the chat.

`ChatServer` redefines `handleOUT`, a protected method in class `NetNode`, that is called when a node performs an **out** to this node, as shown in Listing 4.5. The first field of the tuples that are handled by the server is the command that the client wishes to perform, i.e., in this simple version, sending a message either to everyone currently in the chat, or only to a list of people. The remaining fields depend on the content of the first field. Tuples having a different format are not handled and, hence, are implicitly discarded. The advantage of our approach with respect to an alternative one based on dynamic creation of handler processes for tuples, is that we can implement *reactions* [Cabri *et al.*, 1998] against the presence of some specific tuples.

The `ChatClient` node enters the chat by using the method **subscribe** (Section 2.4.2). The logical locality with which it issues the request for entering the chat will be used by the chat server as the client's *nickname*; the client will be visible to the other clients with this nickname. After succeeding in entering the chat, the `ChatClient`, continuously gets tuples containing messages delivered by the server and displays the messages on the user graphical interface. All these operations are performed by a node coordinator executing on the `ChatClient` whose main code is shown in Listing 4.6. Messages issued by the server aim at keeping a client up-to-date about new clients entering the chat or existing clients leaving the chat. Notice that upon subscription the chat server provides the new client with all the nicknames of the existing clients in the chat system.

When a user enters a message that has to be sent, the graphical interface will send the message to the chat server, as shown in Listing 4.7. Screenshot 4.3 shows three `ChatClients`. We remark that both the server and the clients are relieved from the details of sending and retrieving messages to and from the network: they use tuples and the operations supplied by KLAVA.

5 Performance Assessment

In this section we touch upon performance issues of the `Klava` package. In particular, we will concentrate on those issues related to accessing (local/remote) tuple spaces and those related to

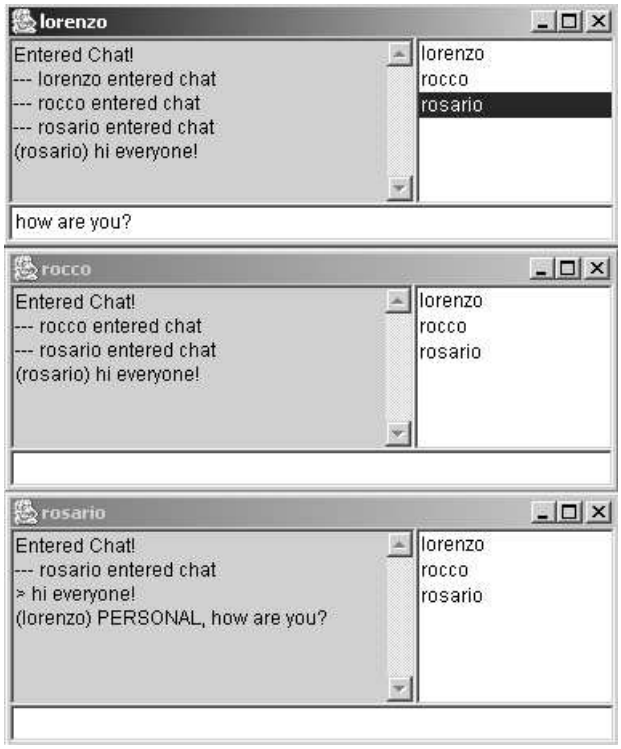
```
public void execute() throws KlavaException {
 if (subscribe(chatServer, name)) {
  KVector allClients = new KVector() ;
  in( allClients, self ) ;
  frame.FillList( allClients ) ;
  frame.AddText("Entered Chat!") ;
 } else {
  Print("Registration Failed!") ;
  return ;
 }

 while ( true ) {
  KString messageCode = new KString() ;
  KString message = new KString() ;
  LogicalLocality from = new LogicalLocality() ;
  in( messageCode, message, from, self ) ;
  if ( messageCode.equals("MESSAGE") ) {
   frame.AddText( ... ) ; // show the message
  } else if ( messageCode.equals("PERSONAL") ) {
   frame.AddText( ... ) ; // show the personal message
  } else if ( messageCode.equals("ENTERED") ) {
   frame.AddText("--- " + message + " entered chat") ;
   frame.AddName( message.toString() ) ;
  } else if ( messageCode.equals("LEFT") ) {
   frame.AddText("--- " + message + " left") ;
   frame.RemoveName( message.toString() ) ;
  }
 }
}
```

Listing 4.6: The method `execute` of the node coordinator executing on the `ChatClient`. The logical locality *name* and the locality of the server, *chatServer*, are provided by the end user.

```
public void sendString( String s ) {
 String[] clients = frame.GetSelectedNames() ;
 if ( clients.length > 0 ) {
  KVector names = new KVector() ;
  for ( int i = 0 ; i < clients.length ; i++ )
   names.addElement( new KString( clients[i] ) ) ;
  out(new KString("PERSONAL"), names, new KString(s), chatServer) ;
 } else
  out(new KString("MESSAGE"), new KString(s), chatServer) ;
}
```

Listing 4.7: Sending a message to the server

Screenshot 4.3 `ChatClient` nodes



code mobility.

In the current implementation, a tuple space is implemented by storing its tuples in a `Vector` object. The search for matching tuples is performed sequentially, thus the average cost for retrieving a matching tuple is linear in the number of stored tuples. However, the class `TupleSpace` abstracts from the specific internal representation of the tuple space, thus alternative implementations can be plugged in by simply implementing specialized subclasses of `TupleSpace`. For instance, the following alternative implementations could be adopted that are based on:

- *Hash tables*: the hash code of a tuple can be computed by smartly composing the hash code of single tuple items;
- *Trees*: tuples are stored in a search tree, and are indexed according to types and values of their items;
- *DBMS*: the tuple space acts as a wrapper for a database where tuples are stored in a specific format. The matching algorithm can retrieve tuples by means of SQL queries or of a richer query language. Indeed, in [Ianett, 2002], an extension of KLAVA has been developed for storing tuples in a database that can be queried through the *TQL* query language [Conforti *et al.*, 2002].

More elaborated techniques, such as those based on fingerprints and summaries presented in [Vitek *et al.*, 2001], can also be employed.

When performing remote retrieval operations, such as, e.g., a read request to a remote site, the remote tuple space is inspected by a thread executing at the remote site. Thus, apart from the communication cost of the remote request and response, the cost of inspection of a remote tuple space is the same as that of a local retrieval.

Inserting a tuple in the local tuple space consists in storing a reference to the tuple in a `Vector`. Obviously, this operation takes place in a *synchronized* way; this is necessary because many processes may try to concurrently access the same tuple space. When inserting a tuple in a remote tuple space, the tuple is sent in a serialized form. Buffering is applied during this operation, so

that the serialization does not interleave heavily with the network communications. Thus, the overhead of interacting with a tuple space is kept to the minimum, and basically does not influence the overall performance of the system.

The cost in time of process migration depends on:

- the time for collecting, through Java Reflection, the classes used by the process (recall that KLAVA relies on the "automatic approach", thus the migrating process carries also the byte code of the classes it uses),
- the time for delivering the serialized state of the process and the contents of the class files.

As explained in Section 3, the (default) choice of sending a process along with its code renders mobile processes more autonomous and independent of the originating site. Moreover, it makes process migration an atomic operation, differently from the on-demand approach that always requires a connection with the home site. However, one can disable the automatic collection and transmission of code for a mobile process by calling the method `setDeliverCode(boolean)` of class `KlavaProcess`. This disabling is reasonable when it is known that (the same version of) the needed code is already installed at every site visited by the mobile process. The automatic transmission of code can also be disabled when the connections are known to be solid and permanent: in this case, one can implement the on-demand approach either by using a low-level remote class loading (e.g., by using the Java `URLClassLoader` functionalities or the *NetworkClassLoader* presented in [Bettini & Cappetta, 2001]), or by implementing a protocol using the KLAVA communication model for exchanging code (see also Section 3).

Notice, however, that a process usually does not use many classes (apart from the ones of the standard packages, which are not moved anyway), hence, for most of the processes exchanged among distributed nodes, there is basically no difference whether the code is shipped or not (in these tests every node had all the classes needed by the processes). Thus, the way KLAVA deals with code mobility does not lead to a great decrease of performance.

An overhead imposed by the KLAVA framework includes the login (or subscription) phase (Section 2.4.2). However, this is not a frequent operation and it is common to distributed systems where resources are addressable by names (see, e.g., Java RMI Name Registry [Sun Microsystems, 2002]). After this first step, distributed nodes basically rely on sockets and the results of the tests we executed show that there is no noticeable difference with respect to a standard client-server socket based application. Obviously, KLAVA nodes that rely on direct communications perform better than those that communicate using a gateway, but, as explained in Section 2.4.2, direct communications cannot be used in some circumstances.

Finally, we would like to remark that issues concerning performance optimization are orthogonal to our prototype implementation. We refer the interested reader to, e.g., [Jagannathan, 1991; Czajkowski & Zieliński, 1993; Kaxiras & Schoinas, 1993; Rowstron & Wood, 1996; James B. Fenwick, 1998; De Nicola *et al.*, 2000] that investigate the problem of optimizing various aspects of the Linda coordination model. We believe that most of these optimization techniques can be exploited also to improve KLAVA.

6 Privacy in Distributed Tuple Spaces

Sharing data over a wide area network such as Internet, calls for very strong security mechanisms. Computers and data are exposed to eavesdropping and manipulations. Issues such as *authentication*, *authorization* and *data integrity* [Gollmann, 1999] are amplified. Dealing with these issues is even more important in the context of code mobility, where code or agents can be moved over the different sites of a net. Malicious agents could seriously damage hosts and compromise their integrity, and may tamper and brainwash other agents. On the other hand, malicious hosts may extract sensible data from agents, change their execution or modify their text [Yee, 1999; Sander & Tschudin, 1998].

The flexibility of the shared tuple space model opens possible security holes; it basically provides no access protection to the shared data. Indeed there is no way to determine the issuer of an operation to the tuple space and there is no way to protect data:

1. a process may (possibly intentionally) retrieve/erase data that do not belong to it;
2. shared data can be easily modified and corrupted.

In spite of this, within the Linda based approaches, very little attention has been devoted to protection and access control.

We have extended KLAVA [Bettini & De Nicola, 2003] with cryptography: classical Linda operations are extended for handling encrypted data. Primitives are also supplied for encrypting and decrypting tuple contents. This finer granularity allows mobile agents (that are not supposed to carry private keys with them when migrating) to collect encrypted data, while executing on remote sites, and decrypt them safely when back at the home site. The proposed extension, while targeted to KLAVA, is still general enough to be applied to similar frameworks using multiple distributed tuples spaces possibly dealing with mobility, such, e.g., [Picco *et al.*, 1999; Arnold *et al.*, 1999; Ciancarini & Rossi, 1997]. Indeed, this extension represents a compromise between the flexibility and open nature of Linda and of mobile code, and the privacy of data in a distributed context.

The basic idea is that a tuple may contain both clear text fields and encrypted fields. All the encrypted fields of a specific tuple are encrypted with a single key. This choice simplifies the overall design and does not harm usability of the system; it would be unusual that different fields of the same tuple are encrypted with different keys. Encrypted fields completely hide the encrypted contents that they embody: they even hide the type of the contents. This strengthens the secrecy of data (it is not even possible to know the type of sensible information).

In line with the open nature of the Linda model, our main intention is not to prohibit processes to retrieve data belonging to other processes, but to guarantee that these data be read and modified only by entitled processes. A shared tuple space is basically a shared communication channel: in such a channel information can be freely read and modified. Should a tuple space be isolated and accessible only by a limited number of local processes, security issues would not be crucial; Indeed, also cryptography is not necessary when working with protected communication channels.

At the same time one of our aims is avoiding that wrong data be retrieved by mistake. Clear text fields of a tuple can be used as identifiers for filtering tuples (as in the Linda philosophy), but if a matching tuple contains encrypted fields, which a process is not able to decrypt, it is also sensible that the tuple is put back in the tuple space if it was withdrawn with an **in**. Moreover, in such cases, a process may want to try to retrieve another matching tuple, possibly until the right one is retrieved (i.e., a tuple for which it has the appropriate decryption key), and to be blocked until one is available, in case no such tuple is found.

Within our framework it is possible to

- use tuple fields with encrypted data;
- encrypt tuple fields with specific keys;
- decrypt a tuple with encrypted fields;
- use variants of the operations **in** and **read** (**ink** and **readk**) to atomically retrieve a tuple and decrypt its contents.

The modified versions of the retrieving operations, **ink** and **readk**, are based on the following procedure:

1. look for and possibly retrieve a matching tuple,
2. attempt a decryption of the encrypted fields of the retrieved tuple

3. if the decryption fails:

   (a) if the operation was an **ink** then put the retrieved tuple back in the tuple space,

   (b) look for alternative matching tuples,

4. if all these attempts fail, then block until another matching tuple is available.

Thus the programmer is relieved from the burden of executing all these internal tasks, and when a **readk** or an **ink** operation succeeds it is guaranteed that the retrieved tuple has been correctly decrypted. Basically the original Linda pattern matching mechanism is not modified: encrypted fields are seen as ordinary fields that have the same type. It can be seen as an extended pattern matching mechanism that, after the structural matching, also attempts to decrypt encrypted fields.

In case mobile code is used, the above approach may be unsafe. Indeed, symmetric and asymmetric key encryption techniques rely on the secrecy of the key (in asymmetric encryption the private key must be kept secret). Thus, a fundamental requirement is that *mobile code and mobile agents must not carry private keys when migrating to a remote site*[2]. This implies that the above introduced operations **ink** and **readk** cannot be used by a mobile agent executing on a remote site, because they would require carrying over a key for decryption.

For mobile agents it is then necessary to supply a finer grain retrieval mechanism. For this reason we introduced also operations for the explicit decryption of tuples: a tuple, containing encrypted fields, will be retrieved by a mobile agent by means of standard **in** and **read** operations and no automatic decryption will be attempted. The actual decryption of the retrieved tuples can take place when the agent is executing at the home site, where the key for decryption is available and can be safely used. Typically a mobile agent system consists of stationary agents, that do not migrate, and mobile agents that visit other sites in the network, and, upon arrival at the home site, can communicate with the stationary agents.

Thus the basic idea is that mobile agents collect encrypted data at remote sites and communicate these data to the stationary agents, which can safely decrypt their contents. Obviously, if some data are retrieved by mistake, it is up to the agents to put it back on the site from where they were withdrawn. This restriction of the protocol for fetching tuples is necessary if one wants to avoid running the risk of leaking private keys. Obviously, public keys can be safely transported and communicated. By using public keys mobile agents are able to encrypt the data collected along their itinerary. An example of a scenario with mobile agents dealing with encrypted tuples is shown in Section 6.2.

Notice that there is no guarantee that a "wrong" tuple is put back: our framework addresses privacy, not security, i.e., even if data can be stolen, still it cannot be read. Should this be not acceptable, one should resort to a secure channel-based communication model, and give up the Linda shared tuple space model. Indeed the functionalities of our framework are similar to the one provided, e.g., by *PGP* [Zimmermann, 1995] that does not avoid e-mails be eavesdropped and stolen, but their contents are still private since they are unreadable for those that do not own the right decryption key.

An alternative approach could be that of physically removing an encrypted tuple, retrieved with an **in**, only when the home site of the agent that performed the **in**, notifies that the decryption has taken place successfully. Such a "ghost" tuple would be restored if the decryption is acknowledged to have failed or after a specific timeout expired. However, this approach makes a tuple's life time dependent on that of a mobile agent, which, by its own nature, is independent and autonomous: agents would be expected to accomplish their task within a specific amount of time. Moreover, inconsistencies could arise in case successful decryption acknowledgments arrive after the timeout has expired.

[2] "Software agents have no hopes of keeping cryptographic keys secret in a realistic, efficient setting" [Yee, 1999].

6.1 Implementation

The extension of this package, CRYPTOKLAVA, provides the cryptography features described in the previous section. We have used the *Java Cryptography Extension* (*JCE*) [Sun Microsystems, 2001], a set of packages that provide a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. JCE defines a set of standard API, so that different cryptography algorithms can be plugged into a system or an application, without modifying the existing code. Keys and certificates can be safely stored in a *Keystore*, an encrypted archive.

CRYPTOKLAVA is implemented as a subpackage of the package `Klava`, namely `Klava.crypto`, so that it is self-contained and does not affect the main package. In the rest of this section we will describe the main classes of the package `Klava.crypto`, implementing cryptographic features.

The class `KCipher` is introduced in order to handle formal and actual fields containing encrypted data (it follows the KLAVA convention that wrapper classes for tuple items start with a `K`). Basically it can be seen as a wrapper for standard KLAVA tuple fields. This class includes the following fields:

```
protected byte[] encItem; // encrypted data
protected Object ref; // reference to the real tuple item
protected String alg; // enc−dec algorithm type
```

The reference `ref` will be `null` when the field is a formal field, or the field has not yet been decrypted. After retrieving a matching tuple, `encItem` will contain the encrypted data (that is always stored and manipulated as an array of bytes). After the decryption, `ref` will refer to the decrypted data. Conversely, upon creation of an actual field, `ref` will contain the data to be encrypted; after encryption, `encItem` will contain the encrypted data, while `ref` will be set to `null` (so that the garbage collector can eventually erase such clear data also from the memory). `alg` stores information about the algorithm used for encryption and decryption.

An actual encrypted tuple field can be created by firstly creating a standard KLAVA tuple field (in the example a string) and then by passing such field to an instance of class `KCipher`:

```
KString s = new KString("foo");
KCipher ks = new KCipher(s);
```

Similarly the following code creates an encrypted string formal tuple field[3]:

```
KString s = new KString();
KCipher ks = new KCipher(s);
```

`KCipher` supplies methods `enc` and `dec` for respectively encrypting and decrypting data represented by the tuple field. These methods receive, as parameter, the `Key` that has to be used for encryption and decryption, and `enc` also accepts the specification of the algorithm. These methods can be invoked only by the classes of the package.

The class `Tuplex` extends the standard KLAVA class `Tuple`, in order to contain fields of class `KCipher`, besides standard tuple fields; apart from providing methods for cryptographic primitives, it also serves as a first filter during matching: it will avoid that ordinary tuples (containing only clear text data) be matched with encrypted tuples. Once tuple fields are inserted into a `Tuplex` object, the `KCipher` fields can be encrypted by means of the method `encode`. For instance, the following code

```
KString ps = new KString("clear");
KCipher ks = new KCipher(new KString("secret"));
Tuplex t = new Tuplex();
t.add(ps); t.add(ks);
t.encode();
```

[3] We recall that, in KLAVA, a formal field is created by instantiating an object from a KLAVA class for tuple fields (such as `KString`, `KInteger`, etc.) through the default constructor.

creates a tuple where the first field is a clear text string, and the second is a field to be encrypted, and then actually encrypts the `KCipher` field by calling `encode`. Also `encode` can receive parameters specifying the key and the algorithm for the encryption; otherwise the default values are used. `encode` basically calls the previously described method `enc` on every `KCipher` tuple field, thus ensuring that all encrypted fields within a tuple rely on the same key and algorithm.

As for the retrieval operation, this can be performed either with the new introduced operations, **ink** and **readk**, if they are executed on the local site

```
KString s = new KString();
KString sec = new KString();
KCipher ks = new KCipher(sec);
Tuplex t = new Tuplex();
t.add(s); t.add(ks);
ink(t, l);
Print("encrypted data is: " + sec);
```

or by first retrieving the tuple and then manually decoding encrypted fields:

```
KString s = new KString();
KString sec = new KString();
KCipher ks = new KCipher(sec);
Tuplex t = new Tuplex();
t.add(s); t.add(ks);
in(t, l);
...
t.decode();
Print("encrypted data is: " + sec);
```

Notice that in both cases references contained in an encrypted field (such as `sec`) are automatically updated during the decryption. The **ink** in the former example is performed at a remote site but this does not mean that the key travels in the net: as explained in the previous section, the matching mechanism is implicitly split into a retrieve phase (which takes place remotely) and a decryption phase (which takes place locally).

Operations **ink** and **readk** are provided as methods in the class `KlavaProcessx`, which extends the class `KlavaProcess` for standard processes (Section 2.5). `KlavaProcessx` also keeps information about the `KeyStore` of the process and the default keys to be used for encryption and decryption. Obviously these fields are `transient` so that they are not delivered together with the process, should it migrate to a remote site. All these extended classes make the extension of KLAVA completely modular: no modification was made to the original KLAVA classes.

Finally, let us observe that, thanks to abstractions provided by the JCE, all the introduced operations are independent of the specific cryptography mechanism, so both symmetric and asymmetric encryption schemes can be employed.

6.2 Programming Examples

In this section we will present two examples implemented using CRYPTOKLAVA: the first one is a chat system where some messages travel encrypted, and the second one is a modification of the classical collecting-agent, where the mobile agent carries encrypted data that will be decrypted by the owner of the agent at its local site. The first example does not deal with mobile agents, but it presents the new primitives in action in a distributed context.

An Encrypted Chat System

The chat system we present in this section is a variant of the one implemented in KLAVA, presented in Section 4.3, with the addition of cryptography for private messages.

Messages are normally delivered through the network as clear text, so they can be read by everyone:

- an eavesdropper can intercept the messages and read their contents;
- a misbehaving chat server can examine clients' messages.

Moreover, the messages might also be modified so that a client believes he is receiving messages from another client, while it would be reading messages forged by a "man in the middle".

While this is normally acceptable, due to the open nature of a chat system, nonetheless there could be situations when the privacy and integrity of messages is a major concern; for instance if two clients want to engage a private communication. This is a typical scenario where cryptography can solve the problem of privacy (through encryption) and of integrity (through digital signatures).

In this example we implement a chat server and a chat client, capable of handling private encrypted messages:

- when the client wants to send a private message to a specific receiver, it encrypts the body of the message with a key;
- the server receives the message and simply forwards it to the receiver;
- the receiver will receive the message with the encrypted body and it can decrypt it with the appropriate key.

Notice that clients that want to communicate privately must have agreed about the specific key to be used during the private message exchange; this is definitely the case with symmetric keys. As for public and private key encryption the receiver can simply use its private key, to decrypt a message encrypted with its own public key.

A private message is represented by a tuple with the following format:

("PERSONAL", <*body*>, <*recipient*>, <*sender*>)

where <*recipient*> and <*sender*> are, respectively, the locality of the client the message is destined to and the locality of the issuer of the message. Basically, when a client wants to send a message with an encrypted body, it will have to perform the following steps:

```
Tuplex t = new Tuplex() ;
KCipher cryptMessage = new KCipher( message ) ;
t.add( new KString( "PERSONAL" ) );
t.add( cryptMessage ) ;
t.add( selectedUser ) ;
t.add( self ) ;
t.encode();
out( t, server ) ;
```

where `message` is the actual message body.

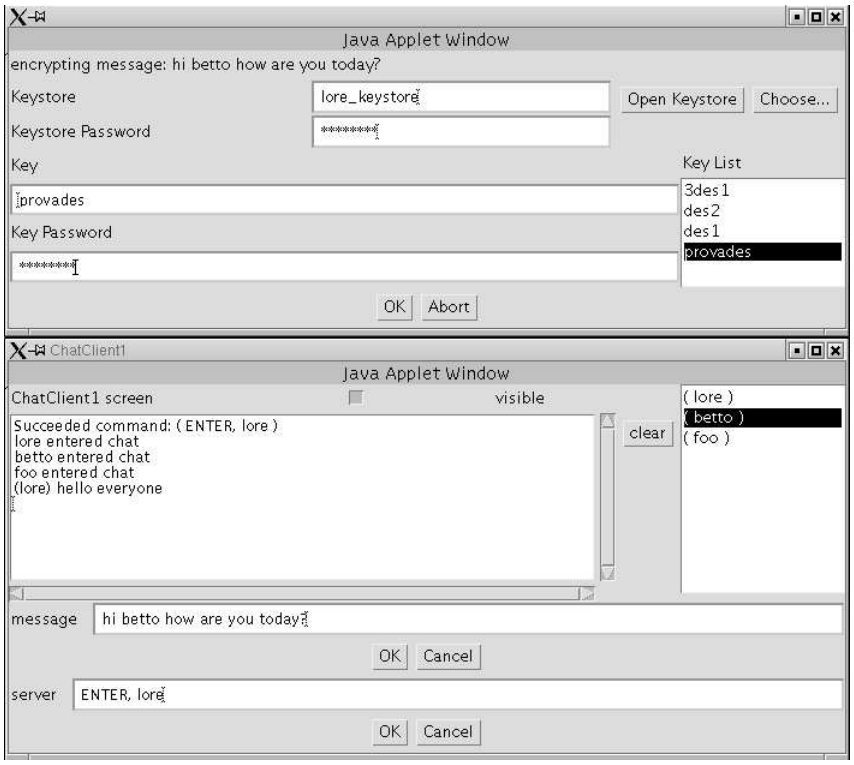
The server handles encrypted messages by retrieving them through the following actions (it will deliver the tuple without the field <*recipient*>, which is useless at this time):

```
KString message = new KString() ;
KCipher cryptMessage = new KCipher( message ) ;
Locality to = new PhysicalLocality() ;
Locality from = new PhysicalLocality() ;
Tuplex t = new Tuplex() ;
t.add( new KString( "PERSONAL" ) );
t.add( cryptMessage ) ;
t.add( to ) ;
t.add( from ) ;
in( t, self ) ;
```

and it delivers the message to the recipient as follows:

```
out( new Tuplex(new KString ("PERSONAL"), cryptMessage, from), to );
```

Screenshot 6.1 A chat client is sending an encrypted message to another chat user; the dialog allows to insert the passphrases for the keystore and the key to use for the encryption.



On the other hand, the receiver, which is always waiting for incoming messages, will read and decrypt a message (in one atomic step), by means of the operation **ink**:

```
KString message = new KString() ;
KCipher cryptMessage = new KCipher( message ) ;
KString from = new KString() ;
Tuplex t = new Tuplex() ;
t.add( new KString( "PERSONAL" ) ) ;
t.add( cryptMessage ) ;
t.add( from ) ;
ink( t, self ) ;
Print("Received message: " + message);
```

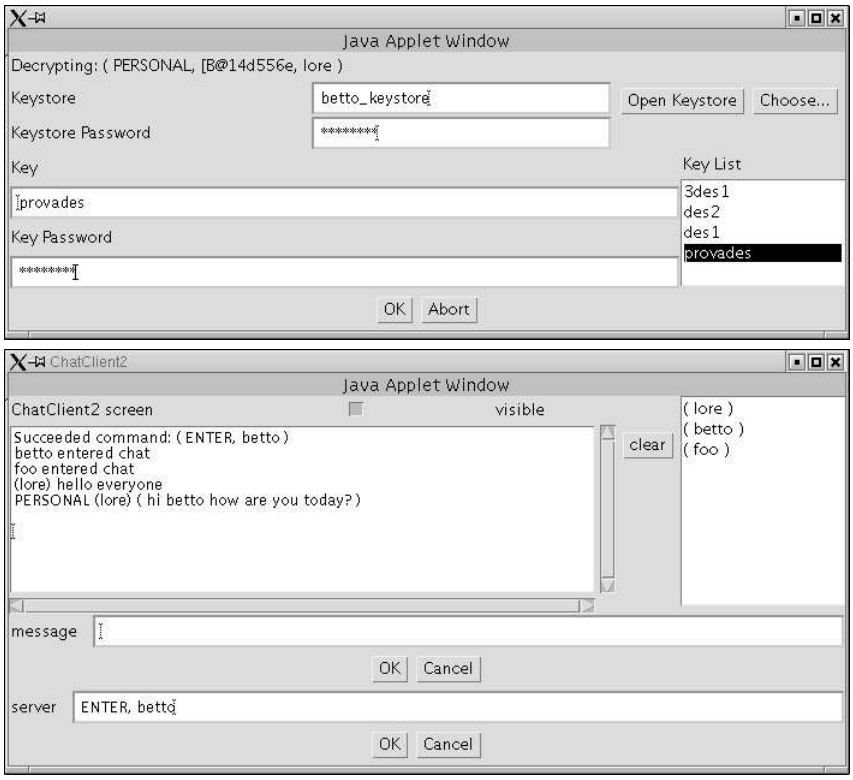
Both the server and the clients execute these operations within the loop for handling incoming messages.

In Screenshot 6.1 a chat client, `lore`, is shown that is sending an encrypted private message to another chat user, `betto`. A dialog pops up to let the send choose the key to use for encryption (and the keystore where the key is stored). Screenshot 6.2 shows the recipient client: a similar dialog pops up to communicate that he has just received an encrypted message; the key for decryption must be chosen by the client according to the sender of the message. Finally the clear text original message is shown on the screen of the recipient.

A mobile agent exploiting encryption

This example relies on the well known scenario of a migrating agent visiting some sites and collecting information on behalf of the owner. During its itinerary, the agent is exposed to attacks from the sites themselves or from possible eavesdropping processes on other sites. The information collected by the agent could be read and possibly modified by these intruders. Since this information could be sensible data, it is important that it is not accessible by no one but the owner of the agent and the agent itself.

Screenshot 6.2 The recipient chat client is receiving an encrypted message and decrypts it with the appropriate key.



For this reason the agent will encrypt, with the public key of its owner, the data collected during its itinerary, so that, even if eavesdropped, these cannot be read by intruders. The agent can safely travel with the public key, and the collected data, once the agent has safely come back home, can be decrypted by the owner by means of his private key. Unfortunately this does not come at no cost: it is a well-known problem (see, e.g., [Yee, 1999]) that the agent is not able to act according to the information collected during its itinerary since it cannot decrypt data (it does not hold the private key):

```
KString s1 = new KString() ;
KString fString1 = new KString() ; // retrieve clear text data...
in(s1, fString1, self, 1000);
Tuplex txf1 = new Tuplex();
txf1.add(s1);
txf1.add(new KCipher(fString1));
txf1.encode(my_public_key) ; // ... encrypt it
collectedData.add(txf1) ;

if ( ! done ) {
  // ... migrate to the next site
} else {
  out(collectedData, owner) ;
}
```

Once the owner receives these data he can try to decrypt them, once they are safely stored in its local tuple space:

```
// decrypt the collected data stored in the
// the local tuple space
...
Tuplex txf2 = new Tuplex() ;
```

```
while ( true ) {
  KString s1 = new KString() ;
  txf2.add( s1 ) ;
  KString fString1 = new KString() ;
  KCipher k = new KCipher( fString1 ) ;
  txf2.add( k ) ;
  if (readk_nb( txf2, self, 100 )) {
    out( "decoded", s1, fString1, self ) ;
    txf2.resetOriginalTemplate();
  } else {
    Print("All possible data decoded" ) ;
    break ;
  }
}
```

In the previous code excerpt, `readk_nb` is the non-blocking version of `readk` (the process is not blocked if no matching tuple is available); `resetOriginalTemplate`, which clears the tuple contents without erasing its identity, bringing back the original formal fields of the original template, allows to iterate through a tuple space, without retrieving the same tuple twice (see Section 2.1).

Let us now consider a slightly different scenario: the sites visited by mobile agents want to be sure that information destined to specific entities cannot be read by others. Even in this case asymmetric encryption helps in solving this problem: the site encrypts data for a person A with the public key of A. This way, even if a mobile agent is able to retrieve data that does not belong to its owner, these data will be useless since they cannot be decrypted. The example we show here implements this scenario: the mobile agent retrieves data according to a specific identifier (represented here by the string `s1`). The data related to this identifier are encrypted. Once the itinerary of the agent is over, the agent sends all the collected data to its owner.

```
Tuplex txf1 = new Tuplex() ;
KString s1 = new KString("item1") ;
txf1.add( s1 ) ;
KString fString1 = new KString() ;
KCipher k = new KCipher( fString1 ) ;
txf1.add( k ) ;
read(txf1, self) ;
collectedData.add(txf1) ;

if ( ! done ) {
  // ... migrate to the next site
} else {
  out(collectedData, owner) ;
}
```

Notice that the previous agent, instead of reading data through the tuple

("item1", !k)

could also read with the tuple

(!s, !k)

This way, it would be able to retrieve also data that is not pertinent with `"item1"` (possibly data that do not belong to its owner). However the agent owner will not be able to decrypt tuples that do not belong to him, since he does not own the keys for decrypting them.

Moreover the type of encrypted data, for the sake of security and privacy, is hidden, so the agent could also retrieve remotely encrypted tuples of the form (`"item1"`, !k) where k does not contain a string. In this case, at the owner site, such tuples would not be retrieved, since, during an **ink** or **readk**, once an encrypted field is decoded, its type is used for the actual matching.

References

ADAMS, M., COPLIEN, J., GAMOKE, R., HANMER, R., KEEVE, F., & NICODEMUS, K. 1996. Fault-tolerant telecommunication system patterns. *Pages 549–562 of:* VLISSIDES, J.M., & COPLIEN, J.O. (eds), *Pattern Languages of Program Design 2*. Addison-Wesley.

ARNOLD, K., FREEMAN, E., & HUPFER, S. 1999. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley.

BETTINI, L. 1998 (April). *Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile*. Master thesis, Dip. di Sistemi e Informatica, Univ. di Firenze.

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

BETTINI, L. 2003b. *X-KLAIM: a Programming Language for Object-Oriented Mobile Code. User's manual*. 1 edn. Dip. di Sistemi e Informatica, Univ. di Firenze. Available at `http://music.dsi.unifi.it/xklaim`.

BETTINI, L., & CAPPETTA, D. 2001. A Java 2 Network Class Loader. *Dr. Dobb's Journal of Software Tools*, **26**(2), 58–64.

BETTINI, L., & DE NICOLA, R. 2001. Translating Strong Mobility into Weak Mobility. *Pages 182–197 of:* PICCO, G. P. (ed), *Mobile Agents*. LNCS, no. 2240. Springer.

BETTINI, L., & DE NICOLA, R. 2003. A Java Middleware for Guaranteeing Privacy of Distributed Tuple Spaces. *Pages 175–184 of:* GUELFI, N., ASTESIANO, E., & REGGIO, G. (eds), *Proc. of FIDJI'02, Int. Workshop on scientific engineering of distributed Java applications*. LNCS, vol. 2604.

BETTINI, L., LORETI, M., & PUGLIESE, R. 2002a. An Infrastructure Language for Open Nets. *Pages 373–377 of: Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications*. ACM.

BETTINI, L., DE NICOLA, R., & PUGLIESE, R. 2002b. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, **32**(14), 1365–1394.

CABRI, G., LEONARDI, L., & ZAMBONELLI, F. 1998. Reactive Tuple Spaces for Mobile Agent Coordination. *Pages 237–248 of:* ROTHERMEL, K., & HOHL, F. (eds), *Proc. of the 2nd Int. Workshop on Mobile Agents*. LNCS, vol. 1477, no. 1477. Stuttgart, Germany: Springer-Verlag, Berlin.

CARDELLI, L., & GORDON, A. 1998. Mobile Ambients. *Pages 140–155 of: Foundations of Software Science and Computation Structures (FoSSaCS'98)*. LNCS, no. 1378. Springer.

CARRIERO, N., & GELERNTER, D. 1989a. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, **21**(3), 323–357.

CARRIERO, N., & GELERNTER, D. 1989b. Linda in Context. *Communications of the ACM*, **32**(4), 444–458.

CIANCARINI, P., & ROSSI, D. 1997. Jada - Coordination and Communication for Java Agents. *In:* [Vitek & Tschudin, 1997].

CONFORTI, G., FERRARA, O., & GHELLI, G. 2002. TQL Algebra and its Implementation. *In: Proc. of the 2nd IFIP International Conference on Theoretical Computer Science*. Kluwer. To appear.

CUGOLA, G., GHEZZI, C., PICCO, G.P., & VIGNA, G. 1997. Analyzing Mobile Code Languages. *In:* [Vitek & Tschudin, 1997].

CZAJKOWSKI, G., & ZIELIŃSKI, K. 1993. Extension of Strand with Linda-like Operations. Implementation and Performance Study. *In:* TICK, E. (ed), *Proc. Workshop on Practical Implementations and Systems Experience in Logic Programming*.

DE NICOLA, R., FERRARI, G., & PUGLIESE, R. 1998. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, **24**(5), 315–330.

DE NICOLA, R., PUGLIESE, R., & ROWSTRON, A. 2000. Proving the Correctness of Optimising Destructive and Non-destructive Reads over Tuple Spaces. *Pages 66–80 of:* PORTO, A., & ROMAN, G-C. (eds), *Proc. of Coordination 2000*.

DEUGO, D. 2001. Choosing a Mobile Agent Messaging Model. *Pages 278–286 of: Proc. of ISADS 2001*. IEEE.

GELERNTER, D. 1985. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, **7**(1), 80–112.

GELERNTER, D. 1989. Multiple Tuple Spaces in Linda. *Pages 20–27 of:* ODIJK, E., REM, M., & SYRE, J. (eds), *Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 89)*. LNCS, vol. 365. Springer.

GOLLMANN, D. 1999. *Computer Security*. John Wiley & Son.

GONG, L. 1999. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley.

HOHLFELD, M., & YEE, B.S. 1998. *How to Migrate Agents*. Available at `http://www.cs.ucsd.edu/~bsy`.

IANETT, R. 2002. *MOBILE_TQL: Interrogazione di Database Distribuiti XML con Agenti Mobili*. Master thesis, Dip. di Informatica, Univ. Pisa.

JAGANNATHAN, S. 1991. Optimizing Analysis for First-Class Tuple-Spaces. *In:* NICOLAU, A., GELERNTER, D., GROSS, T., & PADUA, D. (eds), *Advances in Languages and Compilers for Parallel Processing*. MIT Press.

JAMES B. FENWICK, JR. 1998. *Compiler Analysis and Optimization of Linda Programs for Distributed-memory Systems*. Ph.D. thesis, Univ. of Delaware.

KAXIRAS, S., & SCHOINAS, I. 1993. *Dynamic Optimizations in Linda Systems*. `http://www.cs.wisc.edu/~kaxiras`.

KORPELA, E., WERTHIMER, D., ANDERSON, D., COBB, J., & LEBOFSKY, M. 2001. SETI@home: Massively Distributed Computing for SETI. *IEEE Computing in Science and Engineering*, January.

PARK, A.S., & REICHL, P. 1998. Personal Disconnected Operations with Mobile Agents. *In: Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*.

PICCO, G.P., MURPHY, A.L., & ROMAN, G.-C. 1999. LIME: Linda Meets Mobility. *Pages 368–377 of:* GARLAN, D. (ed), *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*. ACM Press.

ROWSTRON, A., & WOOD, A. 1996. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. *Pages 511–513 of:* BOUGE, L., FRAIGNIAUD, P., MIGNOTTE, A., & ROBERT, Y. (eds), *EuroPar 96*, vol. 1123. Springer.

SANDER, T., & TSCHUDIN, C. 1998. Protecting Mobile Agents Against Malicious Hosts. *In:* VIGNA, G. (ed), *Mobile Agents and Security*. LNCS, vol. 1419. Springer.

SUN MICROSYSTEMS. 2001. *Java Cryptography Extension (JCE), Refence Guide*. available on line.

SUN MICROSYSTEMS. 2002. *RMI, Remote Method Invocation*. `http://java.sun.com/products/jdk/rmi`.

VITEK, J., & TSCHUDIN, C. (eds). 1997. *Mobile Object Systems - Towards the Programmable Internet*. LNCS, no. 1222. Springer.

VITEK, J., BRYCE, C., & ORIOL, M. 2001. Coordinating Processes with Secure Spaces. *Science of Computer Programming*. To appear.

YEE, B.S. 1999. A Sanctuary For Mobile Agents. *Pages 261–273 of:* VITEK, J., & JENSEN, C. (eds), *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*. LNCS, no. 1603. Springer-Verlag. Also Technical Report CS97-537, University of California at San Diego.

ZIMMERMANN, P.R. 1995. *The Official PGP User's Guide*. MIT Press.

Index

JCE, 30
PGP, 29
SETI, 23
ink, 28
readk, 28
CRYPTOKLAVA, 30
`ClassNotFoundException`, 16
`KBoolean`, 5
`KCipher`, 30
`KInteger`, 5
`KString`, 5
`KVector`, 5
`KlavaProcessPacket`, 16
`KlavaProcessVar`, 6, 9
`KlavaProcessx`, 31
`KlavaProcess`, 15
`KlavaSecurityManager`, 19
`Locality`, 6
`LogicalLocality`, 6
`NetNodeProxy`, 15
`NodeClassLoader`, 16
`NodeHandler`, 10
`NodeMessage`, 10
`NodeProxy`, 15
`PhysicalLocality`, 6
`ServerSocket`, 13
`TupleItem`, 5
`TupleSpace`, 7
`Tuplex`, 6
`Tuple`, 4
`closeProcess`, 10
`equals`, 5
`getPhysicalLocality`, 10
`here`, 9
`isFormal`, 5
`match`, 5
`self`, 9
`setValue`, 5

access protection, 28
actual field, 3, 4
allocation environment, 4, 6, 9
anonymous, 3
applet, 12
authentication, 27
authorization, 27
autonomy, 18

bandwidth, 7

certificate, 30
chat system, 23, 31
class loader, 16
closure, 10
code mobility, 15
communication
    anonymous, 3
    associative, 3
    asynchronous, 3
computational environment, 8
connection protocol, 14
content-addressable, 3
credit, 21
cryptographic, 19

data integrity, 27
decrypting, 28
destination uncoupling, 3
digital signature, 32
direct connection, 13
disconnected operation, 16
dynamic scoping, 10
dynamically linked, 18

eavesdropping, 27
encrypted field, 31
encrypted tuple, 30
encrypting, 28
execution engine, 8
execution environment, 15
execution state, 18

flat, 4
formal field, 3, 4
full mobility, 18

gateway, 8, 10
GUID, 6

heterogeneous, 3
hierarchical net, 4

integrity, 32
itinerary, 33

Java Reflection API, 16

key, 30
Keystore, 30

latency, 7
load balancing, 21
locality, 3
logical locality, 6

MAC, 30
malicious, 27
man in the middle, 32
manipulation, 27
migrating agent, 33

news gatherer, 19
node connectivity, 4
node coordinator, 11
NodeCoordinator, 4

objective mobility, 23
on-demand, 16

pattern
    leaky bucket of credits, 21
pattern-matching, 3, 5
peer to peer, 13
performance, 24
physical locality, 6
privacy, 29, 32
private communication, 32
private key, 19, 29, 32, 34
private message, 31
program counter, 18
propagation, 15
proxy, 10, 15
public key, 29, 32, 34
pull, 23
push, 23

reaction, 24
reboot, 19

safety, 19
security, 29
shared data, 28
shutdown, 19
space uncoupling, 3
stack, 18
static scoping, 10
strong mobility, 18, 23
subnet, 4, 8
subtyping, 6
super user, 11
symmetric key, 32

time uncoupling, 3
time-out, 8
topology, 4
tuple, 3, 4
tuple space, 3

weak mobility, 18