**MIKADO Global Computing Project IST-2001-32222**

*Mobile Calculi Based on Domains*

# Virtual Machine Technologies: Core Software Framework v0

**MIKADO Deliverable D3.1.0 :**
**Towards MIKADO Abstract Machines**

| | |
|---|---|
| **Title** : | Virtual Machine Technologies:<br>Core Software Framework v0 |
| **Editor** : | R. De Nicola (U. of Florence) |
| **Authors** : | L. Bettini, R. De Nicola (U. of Florence), M. Lacoste (FTR&D),<br>Luís Lopes (U. of Porto), V. Vasconcelos (U. of Lisbon) |
| **Classification** : | Deliverable D3.1.0, Public |
| **Reference** : | RR/WP3/2 |
| **Version** : | 1.0 |
| **Date** : | January 2003 |

**Abstract**

This document discusses the key issues of the MIKADO Core Software Framework. In particular, four important components of abstract machines for mobile calculi are discussed: node topology, naming and binding, communication protocols and mobility. The document also describes the future activities within work package 3.

# Contents

# 1 Introduction

In the first year of the project, activity has continued at the different sites on the development of prototypes of abstract machines built to support the calculi for mobility there developed. More specifically, the partners of the MIKADO consortium have worked on X-KLAIM and KLAVA (Firenze); TYCO and DITYCO (Lisbon); M-calculus implementation CLAM and JONATHAN middleware (France Telecom); Safe Ambient Abstract Machines (INRIA).

For the second year of the project, the development of software tools will take advantage of the know-how accumulated by the project partners. In the first meeting at Firenze, the participants decided that, rather than developing a machine to support a specific programming model, they would develop a generic framework that could be used as a kind of middleware for the implementation of mobile programming systems. The main constraint in the design of such a framework is that it has to be as general as possible, and has to permit permit the implementation of existent systems (Klava, Safe Ambients, DJoin, M-calculus and DiTyCO) on top of it. The framework should, for the sake of dissemination and portability, be developed as a set of Java interfaces and classes. Thus, the Virtual Machine technology to be used will essentially be based on the Java Virtual Machine.

The proposed middleware framework should implement or specify all the required functionalities for arbitrary components to communicate and move in a distributed setting. Four abstractions were isolated as being fundamental to this goal:

- *node topology* - encapsulating the heterogeneity of the different abstract machines available and the topological structure of the net;
- *naming and binding* - providing a uniform way to designate and interconnect the set of objects involved in the communication paths between computational nodes;
- *communication protocols* - defining the communication strategies and (physical) connectivity primitives for the programmer;
- *mobility* - providing the abstractions for mobile units and primitive movement operations.

The workgroup agreed that the framework should be provided as a set of Java APIs and LIBs. The APIs are provided for those issues that are machine specific for the various existent systems. The LIBs are composed of actual class implementations for those services that can be implemented in a generic way or are otherwise of widespread use (e.g., communication protocols).

Part of the work carried on during the first year is also described in [BDP02, GLS02], the corresponding papers are attached to this document and should be considered as integral part of it. The four items outlined above will be considered in more details in the next sections.

# 2 Node Topology Management

The purpose of this part of the framework is to describe the encoding of the topological structure of the network and to take into account the effect of distributed computations performing changes in its overall structure.

## 2.1 Existing Work

Despite the diversity of existing distributed process calculi to model distributed mobile computation [BCGL02], in their implementations, only a few alternatives are observed concerning the organization of computational nodes which is either flat or hierarchical, usually using a tree-based structure. We review below some existing solutions by describing node structures and topological node patterns [BDLV02]. Additional details concerning node identification and node creation/deletion are also given:

- In X-KLAIM and KLAVA, tuple spaces are distributed over *nodes* which are elements of a *net*. Each node contains a single tuple space, and a number of running processes, and can be accessed through identifiers called *localities*, which are either physical or logical: physical localities uniquely identify a node within a net; logical localities are symbolic names for nodes, valid only within the context of a physical locality. Logical localities are mapped to physical localities using *allocation environments*. In the original version of X-KLAIM the organization of nodes, both logical and physical, was flat, more recent versions permit hierarchical structuring [BLP01, BLP02]. New nodes can be created using the **newloc**($l$) operation.

- The DITYCO implementation is organized into a three-layered architecture: *sites* are sequential computational units (essentially TYCO VMs containing located channels, definitions, objects and messages), hosted by *nodes* which provide support for communications. A flat organization of nodes is called a *network*. Nodes are uniquely identified by their IP address, and may host any number of sites running concurrently. A network-level naming service provides IP address information given some site name as input, or network-level names (IP address+site name+unique identifier) of given exported channels. At a finer level of granularity, definitions, objects, and messages can move from site to site.

- The SAM implementation of Safe Ambients separates logical and physical distribution of *ambients*: if the `in` and `out` capabilities are implemented by a modification of the logical ambient hierarchy without any physical movement, only the `open` capability calls for a physical change of location. More precisely, the implementation separates *nodes* which are uniquely identified by their IP address, and *agents* which are clustered on a given node, and stand for located ambients. Each agent maintains a link to its parent agent which can reside on a different node. Thus, the organization of nodes is flat, while that of agents is tree-structured, the hierarchy of agents possibly spanning several nodes. An agent is uniquely identified by the name of the node on which it resides, and by a unique suffix which is different for each agent on the same node. The `open` primitive can cause the removal of a given agent, and modify physical distribution.

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

location in the location tree. At system level, locations are designated using some notion of *reference*, which stands for a name, and can be created using the JCL `loc` (`let loc` in JOCAML) primitive. The name space for references is global. Naming facilities include a service on each physical site to create fresh names, and a mechanism to share names between sites (naming service).

- In the CLAM, as in JOCAML, *locations* also serve as a uniform way to describe logical and physical distribution: each physical site called a *machine* hosts a location tree. Distributed interaction occurs by asynchronous message passing between top-level locations, and is formalized using a global *ether*. A location hosts a number of flat (i.e. non-cell) processes living together at the same level of the M-calculus cell hierarchy and holds pointers to children locations. Thus, locations are organized into a forest, each machine containing a single location tree. In each location tree, leaves (*ether-locations*) contain the flat processes found at the top level of a cell content, while other nodes (*membrane-locations*) implement a cell membrane. To support routing and guarantee the unicity of cell names, each machine contains a *lookup service* to locate both communication resources (in which location is a resource defined?) and locations (on which machine does a location reside?). In the implementation, locations are uniquely identified using *internal names*, which are rich names which hold the name of the machine they were created on. Each machine maintains a list of locations which reside locally on that machine, and contains a *name factory* to create fresh internal names. Operations such as cell creation and cell passivation/activation change the structure of the forest of locations by adding or deleting nodes.

## 2.2 Requirements

This quick tour of some existing implementations in turn suggests the following remarks from which can be derived some design requirements for the Core Software Framework, in order to choose the right abstractions and primitives.

### 2.2.1 Abstraction and Encapsulation

- **Observation:** The notion of node appears in all existing implementations. However, the internal structure of the node itself is programming-model specific.
- **Discussion:** Computational nodes can include data structures ranging from definitions (DITYCO, JCL/JOCAML, CLAM), processes (X-KLAIM/KLAVA, DITYCO, SAM, JCL/JOCAML, CLAM) and channels (DITYCO, SAM, JCL/JOCAML, CLAM), to objects (DITYCO) and tuple spaces (X-KLAIM/KLAVA).
- **Requirement 1 (Abstraction and Encapsulation):** *The primitives of the Core Framework should not aim at describing precisely the structure of nodes, but instead provide some abstraction to exhibit computational nodes explicitly.*
- **Remark:** This requirement is similar to the need for some notion of component offering both abstraction and encapsulation found in standadardized component models such as CCM from OMG, EJB from Sun, or COM from Microsoft. See also [BCS02] for a more detailled discussion of abstraction and encapsulation.

### 2.2.2 Identity

- **Observation:** In all existing implementations, nodes are designated using some form of identifier.
- **Requirement 2 (Identity):** *Nodes should have a well-defined identity exhibited in the Core Framework by some notion of node identifier. However, the structure of the identifier itself is implementation-specific.*

### 2.2.3 Naming

- **Observation:** Most implementations provide services to create fresh node identifiers.
- **Discussion:** Thus, it sounds reasonnable for this kind of service to appear in the Core Framework.
- **Requirement 3 (Naming):** *The Core Framework should provide a service to create fresh node identifiers.*

### 2.2.4 Composition

- **Observation:** In existing implementations, the topological organization of nodes is either hierarchical, i.e., tree-structured, or flat.
- **Discussion:** The SAM, JCL/JOCAML, and CLAM implementations use a tree-structured topology of nodes, while in X-KLAIM/KLAVA and DITYCO, nodes are organized into a flat structure pattern. Primitives allowing to express a topological hierarchy of nodes can easily be used to reflect a flat organization of sites by adding a root (virtual) location whose children are the given sites. The converse, i.e., to express a hierarchy of nodes starting from a framework only able to express flat node organizations, appears more difficult. Thus, a sound requirement for the framework should be to be able to support hierarchical node composition patterns.
- **Requirement 4 (Composition):** *The Core Framework should support hierarchical node composition patterns. In particular, this implies being able to query the node hierarchy, e.g., to retrieve the set of subnodes of a given node.*

### 2.2.5 Reconfiguration

- **Observation:** In some implementations, reduction rules imply adding new nodes to the hierarchy or removing existing nodes.
- **Discussion:** This can be due to the strict implementation of the reduction rules, or to new components dynamically being introduced into a running system. This requirement should also be part of the Core Framework.
- **Requirement 5 (Reconfiguration):** *The Core Framework should provide primitives to add new nodes to the node hierarchy or to remove existing nodes.*

## 2.3 Proposed Interfaces

These requirements lead to the following tentative design, inspired by the FRACTAL component model [BCS02, Obj02], for the part of the Core Framework dealing with network topology management. The framework includes the following set of basic Java interfaces, which provide an answer to the abstraction and encapsulation, identity, and naming requirements:

```
interface NodeIdentifier {}

interface Node {
    public NodeIdentifier getNodeIdentifier();
    public Object getImplementation();
}

interface NodeIdentifierFactory {
    public NodeIdentifier newNodeIdentifier();
}
```

The `NodeIdentifier` marker interface defines a generic way to identify a computational node component. Implementation-specific nodes identifiers may be obtained by designing specific classes implementing that interface. The `Node` interface describes the basic computational nodes which may also be called location, site, agent, ..., according to the underlying programming model. It contains a reference to its node identity interface, as well as a reference to an object describing in greater detail the internal structure of the node. That particular structure is implementation specific, and is not part of the Core Framework. Naming services conforming to the `NodeIdentifierFactory` interface are to create fresh node identifiers.

The composition and reconfiguration requirements lead to the introduction of the following Java interface, specifically dedicated to the management of the node topology:

```
interface TopologyManager {
    public NodeIdentifier [] getSubNodes(NodeIdentifier nodeId);
    public void addSubNode(NodeIdentifier nodeId,
                           NodeIdentifier subnodeId);
    public void removeSubNode(NodeIdentifier nodeId,
                              NodeIdentifier subnodeId);
    public Node getSubNode(NodeIdentifier nodeId,
                           NodeIdentifier subnodeId);

}
```

The `TopologyManager` interface supports node hierarchy management mechanisms. It provides the ability to reconfigure the network by adding a new subnode to a node designated by a given node identifier, or to remove an existing subnode from a given node. The interface also includes methods allowing to query the structure of the node hierarchy.

# 3 Naming and Binding

The purpose of this part of the framework is to define a uniform manner to designate and interconnect the set of objects involved in the communication paths between computational nodes. We shall call *binding* such a set of objects.

## 3.1 Existing Work

Many common distributed process calculi [BCGL02] belong to the family of name-passing calculi, where the primitive data elements are *names* exchanged on communication channels. Though the notion of name or identifier be implemented in various ways in existing platforms, it is present in all implementations of distribted process calculi we are aware of, and should thus be part of the Core Framework. In implementations, a notion of name is only valid within a certain *context*, sometimes called *environment*, which captures at the implementation-level the scoping discipline of the calculus, and generally maps a name to an object or entity designated by that name, or can also map names to other contexts, if the resolution of names needs to be refined. Thus, there is a also need for some generic notion of *context* within the Framework. Finally, once an entity knows a name viewed here as a capability to communicate, there is a need to be able to create an access path towards the entity designated by that name.

## 3.2 Requirements

The above discussion leads to the following requirements:

- **Requirement 5 (Identity):** *The Core Framework should include a notion of* ***identifier*** *viewed as a generic notion of name that uniquely designates an object in a given naming context. Identifier semantics should be naming context-specific: distributed, persistent... More generally, a* ***reference*** *should encapsulate a number of identifiers.*
- **Requirement 6 (Naming):** *The Framework should also include a notion of* ***naming context*** *to provide name creation and management facilities. A naming context should guarantee that each of the names it controls unambiguously designates some object.*
- **Requirement 7 (Binding):** *Finally, the Framework should include a special kind of naming context that, for a given managed name, is able to create an access path towards the object designated by that name. We shall call* ***binder*** *such a naming context.*

## 3.3 Proposed Interfaces

We now describe a minimal set of interfaces for dealing with naming and binding, inspired from the JONATHAN Binding Framework [Kra02]. More details are given in appendix.

The definitions given in section 3.2 offer a generic and uniform view of bindings, and clearly separate object identification from object access:

- In a given naming context `nc`, a new name for an object `o` is obtained by the `nc.export(o)` invocation. Chains of identifiers can then be created by exporting that name to other naming contexts.

- The creation of an access path to object `o` designated by identifier `id` is performed by the `id.bind()` invocation which returns a ready-to-use surrogate to communicate with `o`.

These notions concerning naming and binding can be formulated using the following basic Java interfaces:

```
interface Identifier {
    public NamingContext getContext();
    public Object bind();
    public void unexport();
    public Object resolve();
    public byte [] encode();
    public void encode(Marshaller m);
}

interface Reference {
    public Identifier [] getIdentifiers();
    public void setIdentifiers(Identifier [] ids);
}

interface NamingContext {
    public Identifier export(Object obj);
    public Identifier decode (byte [] data);
    public Identifier decode(UnMarshaller u);
}
```

The `Identifier` interface represents the generic notion of identifier described above, used to designate some object relatively to a given naming context, such as an object reference within a JVM, or an integer within an array of objects. The interface contains a reference to its naming context, and bears the fundamental `bind` operation to set up a binding between two (possibly remote) objects. The other methods deal with other aspects of the naming model and respectively:

- cancel an `export` operation by making an identifier no longer valid within a naming context (`unexport` method), i.e. the mapping identifier-object designated by that identifier is broken.

- return the next element in a chain of identifiers, where each identifier was obtained as the result of exporting the next one to some naming context (`resolve` method).

The last two methods deal with identifier marshalling for transmission to a remote host (`encode` method), either returning an array of bytes, or a high-level encoding-independent representation of a message about to be sent (`Marshaller` interface). More generally, the `Reference` interface is simply a wrapper for a set of identifiers.

An object implementing the `NamingContext` interface stands for the most generic notion of a naming context which manages names of type `Identifier`. The interface includes the

`export` operation to create a new name in a given context – which can also, if used repeatedly, create chains of identifiers of arbitrary length – and the `decode` method to unmarshall an identifier, either from an on-the-wire representation (array of bytes) or from a high-level encoding-independent representation of a message about to be received (`UnMarshaller` interface).

# 4 Communication Protocols

The purpose of this part of the framework is to identify both the abstractions and the primitives for logical and physical node connectivity, as well as the strategies which can be used to capture and perform communications between computational nodes. One should keep in mind the general aim of the framework which is to assist both the designer and the programmer of a virtual machine implementing a domain-based programming model.

## 4.1 Existing Work

When looking at the communication protocols found in existing implementations [BDLV02] of some common models of distributed programming using process calculi [BCGL02], one is struck by the diversity of protocols and programming languages used. In first approximation, the communication protocols can be split into two families: high-level protocols such as Java RMI are well integrated with the Java Virtual Machine environment and take advantage of the architectural independence provided by Java (SAM implementation of Safe Ambients) ; protocols closer to hardware resources, such as socket-based implementations of TCP/IP, are accessible, either directly in Java (KLAIM/KLAVA) or in other programming languages allowing easier manipulation of system resources such as OCaml (JCL/JOCAML) or C (DITYCO). Marshalling strategies range from dedicated byte-code structures (JCL/JOCAML, DITYCO) to Java serialization (SAM, KLAIM/KLAVA).

## 4.2 Requirements

From the above discussion, it seems that an important design requirement for the communication framework should be not to concentrate on a fixed set of communication primitives or marshalling strategies but instead to be flexible enough to support multiple marshalling strategies and communication protocols by possibly introducing (statically or dynamically) new strategies and customized protocols.

- **Requirement 8 (Flexibility):** *The communication framework should support multiple marshalling strategies and communication protocols with possible introduction of new strategies and customized protocols.*

Another important requirement for the framework is to aim at minimality so that customized communication protocols or marshalling strategies can be implemented with little effort by either programming special-purpose implementations of the framework interfaces or by defining framework increments which will complement the core framework interfaces and libraries. In any case, a whole communications library should not be reimplemented.

- **Requirement 9 (Minimality):** *The communication framework should be minimal.*

To fill this need for a minimal framework for flexible communications, a number of adaptable middleware aiming at minimality have already been specified and implemented. Examples include FLEXINET [HHD98], OPENORB [Exo] or the $\mu$-ORBs TAO[OKS+00], ZEN [KSO02], and JONATHAN [DHDS98, Kra02]. Some of these middleware provide support for building highly-flexible ORBs where objects interact transparently through remote method invocation on well-defined interfaces. Their originality compared to traditional object platforms such as CORBA or Java RMI is to provide a core framework for building different types of middleware – called *personalities* in the case of JONATHAN – based on the notions of *binding* and of *flexible communications*:

- Creating a new binding should be understood as setting up access and communication paths between components of a distributed system. Bindings may come in a wide variety of semantics: mobile, persistent, with a QoS guarantee ... Thus, an adaptable communication framework should provide means to define bindings with various semantics, and to combine them in flexible ways.

- With some simple architectural principles such as separating marshalling from protocol implementation, or threading from resource management, some of those middleware allow, for instance, to dynamically introduce new protocols, or to control the level of multiplexing of resources.

Such frameworks could provide useful inspiration to define the main abstractions and primitives of the MIKADO Core Communication Framework. One of their prominent features is that the real "communication" part of the framework (wiring, protocols, sessions...) appears as completely separate from a naming and/or binding model, often drawing its inspiration from the ODP Reference Model [ISO95], which "provides tools for managing names (identifiers) and developing binding factories (or extending available ones). Different inter-object binding models may be managed, allowing for example the use of different qualities of service" [Kra02]. The communication framework itself builds on this naming and binding model and "defines the interfaces of the components implied in inter-object communications, such as protocols and sessions. It provides tools for composing these pieces to construct new protocols" [Kra02].

A communication framework inspired by such guiding principles can easily provide support for the network communication transparency often needed in the implementation of a particular process calculus, by defining a new personality directly above the communication abstractions provided by the framework implementation (for instance TCP/IP for lightweight communications, or Java RMI for better integration with the Java environment).

### 4.3 Proposed Interfaces

A modular framework for implementing flexible communication protocols can be constructed using only the following abstractions, inspired from the $x$-kernel [HP91], and from the JONATHAN Communication Framework [Kra02]:

- A *protocol* object is a binder representing some network protocol. It ensures the management of session identifiers, and the creation of bindings towards session objects.

- A *session* is a logical communication channel dynamically created by a protocol object. It has higher and lower interfaces to send messages down and up a protocol stack, which may be viewed as a stack of sessions. Sessions are identified using *session identifiers* which designates exported session objects. Their internal structure is protocol-specific. A session identifier is typically created on the server side when an object is exported to a protocol and may be used on the client side to establish a communication channel.

The main protocol-related communication interfaces are the following:

```
interface Protocol {}

interface ProtocolGraph {
    public SessionIdentifier export(Session_Low hls);
}
```

The `Protocol` marker interface represents communication protocols like TCP or GIOP. "A *protocol* provides a naming context that only deals with a special family of interfaces – called *sessions* – and manages names – called *session identifiers* to designate these interfaces. The actual naming context associated with a protocol is called a *protocol graph*: a protocol graph describes a rooted acyclic structure composed of one or several communication protocols, e.g. a protocol stack" [Kra02]. Its structure reflects the path the messages should follow when sent or received over the network. The `export` method is used to inform the communication layers that the exported session object (which should be of type `Session_Low`) is willing to accept messages. The object will be exported, starting at the root of the protocol graph, and calls to the appropriate `export` methods will be made recursively on each node of the protocol graph. A session identifier is then returned: it constitutes a new name for the exported server-side session object. To communicate with the exported session, the client just needs to call the `bind` method on the returned session identifier. This will create a ready to use surrogate (a `Session_High` object) the client can use to send messages to the exported server session.

The session-related communication interfaces are the following:

```
interface SessionIdentifier {
    public Protocol getProtocol();
    public Session_High bind(Session_Low hls);
}

interface Session_High {
    public void send(Marshaller m);
    public void close();
}

interface Session_Low {
    public void send(UnMarshaller message,
                     Session_High session);
}
```

A `SessionIdentifier` designates an exported session. Its internal structure is protocol specific: each protocol may define its own `SessionIdentifier` type so that it can contain protocol specific information: host name, port, .... In the case of TCP/IP, a session identifier encapsulates a host name and port number. Session identifiers are created when server objects are exported, and can typically be encoded in identifiers. On the client side, they can be decoded and used to establish communication channels using the `bind` operation, possibly using an additional session object as a parameter to receive the messages sent by the remote server-side session.

The two other interfaces are the main session interfaces. Their purpose is to send messages down (`Session_High`) or up (`Session_Low`) a protocol stack: they just include a single method for doing the message send operation, and with possibly another for releasing resources associated with the session.

- A `Session_High` object is a session used to send messages down to the network. It will usually be some kind of surrogate for a session of type `Session_Low`, which has been exported to a protocol object and is designated by a `SessionIdentifier` object. A `Session_High` instance may be obtained using the `bind` operation on a session identifier representing a `Session_Low` interface: it is thus a surrogate, or a proxy, for that interface. "If `lower` is a variable that designates a TCP/IP session in the application session, `lower.send(message)` sends a message down to the network (eventually to the remote application session for which the TCP/IP session is a surrogate)" [Kra02].

- A `Session_Low` object is a session used to forward messages coming from the network to their actual recipient. `Session_Low` is also the type of interfaces exported to protocols, and designated by session identifiers. "If `hls` (standing for "higher level session") is the variable that designates an application session in the TCP/IP session, `hls.send(message)` sends a (presumably incoming) message up to the application session" [Kra02]. The additional parameter in the `send` method represents the sender, and may be used to send a reply, if necessary.

In the previous interfaces, marshallers and unmarshallers are used as a high-level encoding-independent representation of messages that are about to be sent or received. Their typical interfaces would look like:[1]

```
interface Marshaller {
    public public void writeByte(byte b);
    public void writeBoolean(boolean b);
    public void writeChar(char i);
    public void writeShort(short i);
    public void writeInt(int i);
    ...
    public void writeReference(Object obj);
    public boolean isLittleEndian();
    public void close();
```

[1] Only the `Marshaller` interface is presented. The `UnMarshaller` interface is similar but with read instead of write operations.

```
}
```

Each session also contains a lower-level abstraction of a communication channel, called a *connection*, which is typically an encapsulation of a regular socket, and provides operations like `emit` or `receive` to read and write to the socket.

# 5 Code Mobility Management

The purpose of this part of the framework is to provide the basic functionalities for making code mobility transparent to the programmer: all the issues related to code marshaling and code dispatch will be handled automatically by the classes of the framework. Thus, this part will also provide a base class that all the objects/processes that have to migrate must inherit from.

## 5.1 Existing Work

Two main forms of agent mobility have been identified in the literature [CGPV97, HY98]:

- *strong mobility*: the movement of the code and of the execution state of a thread to a different site and the resumption of its execution on arrival (i.e. mobile agents automatically resume execution from the point following migration);
- *weak mobility*: the dynamic linking of code arriving from a different site (i.e. mobile agents after migration start execution from the beginning).

Some existing mobile agent systems implementing strong mobility are *Telescript* [Whi96], *Agent Tcl* [LO95], *Sumatra* [ARS97] and *ARA* [PS97]. Systems such as *TACOMA* [JSR98], *Mole* [SBH96] and *Aglets* [LO98], and, 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 the MIKADO framework can only supply *weak mobility* of agents.

In the following, we briefly review some existing frameworks for mobile code, highlighting their features related to code mobility management:

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

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

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

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

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

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

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

- In SAM, an agent can be a *located ambient* or a *forwarder*. Located ambients are the basic unit of SAM, and represent SA ambients running at a specific physical address, and their (internal) local processes; a located ambient becomes a forwarder when opened.

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

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

  SAM provides strong mobility; this is possible since SAM code is not written directly in Java: it is compiled into Java. In SAM, an agent starts and ends its life on the same physical location. Only opening an agent (i.e., destroying it) its contents can migrate to another host. Since processes are expressed as Java object trees, the SAM interpreter simply stop executing a process, serialize it on another node, and resume it, merging it to the contents of an agent, to be executed.

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

- In the two main implementations of the JOIN-calculus, JCL (Join Calculus Language) and JOCAML, the VM is fairly standard and allows the execution of bytecode threads. The VM memory can be divided into three main segments:

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

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

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

  JCL and JOCAML allow three types of code mobility:

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

### 5.2 Requirements

When code (e.g., a process or an object) is moved to a remote computer, its Java classes may be unknown at the destination site. 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* [PR98]: 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, an object must be sent along with its class binary code, and with the class code of all the objects it uses. Obviously, only the code of user defined classes has to be sent, as the other code (e.g. Java and the classes of the MIKADO framework) has to be common to every 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.

### 5.3 Proposed Interface

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 MIKADO package. When a remote object or a migrating process is received from the network, before using it, the node must add the class binary data (received along with the object) to its class loader's table. Then, during the 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.

We can define a base class for all objects/process that can migrate to a remote site, `MigratingObject`[2], that implements all the procedures for collecting the Java classes that the migrating object has to bring to the remote site.

When extending `MigratingObject`, there is an important detail to know in order to avoid run-time 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` of class `MigratingObject`; all the details of code marshaling are still handled by the framework and are transparent to the programmer.

Once these class names are collected, their byte code is gathered in the first server from which the process/object was sent, and packed along with the process in a `MigratingPacket`

[2]Unfortunately, Java only provides single inheritance, thus providing a base class could restrict its usability; however, we can adopt the same idiom used, for instance, by the class `Thread` and the interface `Runnable` in the standard Java class library.

object. Notice that the migrating object (namely, its variables) is written in an array of bytes and not in a field of type `MigratingObject`. This is necessary because otherwise, when the packet is received at the remote site and read from the stream, the remote object would be deserialized and an error would be risen 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 object is read from the byte array; when its 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 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 or object is retrieved by the method `getClassBytes` of the class loader: at the server 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.

## References


[ARS97] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In Vitek and Tschudin [VT97], pages 111–130.

[BCGL02] G. Boudol, I. Castellani, F. Germain, and M. Lacoste. Models of Distribution and Mobility: State of the Art. MIKADO Global Computing Project Deliverable D1.1.1, 2002.

[BCS02] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings WCOP'02*, 2002.

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

[BDLV02] M. Boreale, R. De Nicola, M. Lacoste, and V. Vasconcelos. Analysis of Distribution Structures: State of the Art. MIKADO Global Computing Project Deliverable D3.1.1, 2002.

[BDP02] L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java Framework for Distributed and Mobile Applications. *Software – Practice and Experience*, 2002. To appear. Available at `http://music.dsi.unifi.it/papers.html`.

[BLP01] L. Bettini, M. Loreti, and R. Pugliese. Modelling Node Connectivity in Dynamically Evolving Networks. In *Proc. of CONCOORD, Int. Workshop on Concurrency and Coordination*, volume 54 of *ENTCS*, 2001.

[BLP02] L. Bettini, M. Loreti, and R. Pugliese. An Infrastructure Language for Open Nets. In *Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications*, pages 373–377. ACM, 2002.

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

[DHDS98] B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stefani. Jonathan: an Open Distributed Processing Environment in Java. In *Proceedings MIDDLEWARE'98*, 1998.

[Exo] ExoLab Group. The OpenORB project. Software available for download at `http://openorb.exolab.org/`.

[GLS02] Florence Germain, Marc Lacoste, and Jean-Bernard Stefani. An Abstract Machine for a Higher-Order Distributed Process Calculus. In *Proc. of F-WAN 2002 (Foundations of Wide Area Network Computing)*, volume 66 of *Electronic Notes in Theoretical Computer Science*, 2002.

[HHD98] R. Hayton, A. Herbert, and D. Donaldson. Flexinet: a Flexible Component Oriented Middleware System. In *Proceedings ACM SIGOPS European Workshop*, 1998.

[HP91] N. Huntchinson and L. Peterson. The $x$-kernel: an Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[HY98] M. Hohlfeld and B.S. Yee. How to Migrate Agents. Available at `http://www.cs.ucsd.edu/~bsy`, 1998.

[ISO95] ISO (International Standards Organization). *ODP Reference Model: Overview, Foundations, Architecture*, ISO/IEC JTC1/SC21/WG7 10746-1, 10746-2 and 10746-3 edition, 1995.

[JSR98] D. Johansen, F.B. Schneider, and R. van Renesse. What TACOMA Taught Us. In D. Milojicic, F. Douglis, and R. Wheeler, editors, *Mobility, Mobile Agents and Process Migration - An edited Collection*. Addison-Wesley, 1998.

[Kra02] S. Krakowiak. *The Jonathan Tutorial: Overview, Binding, Communication, Configuration and Resource Frameworks*. ObjectWeb Consortium, 2002. Available electronically at `http://www.objectweb.org/jonathan/doc/tutorial/index.html`.

[KSO02] R. Klefstad, D. Schmidt, and C. O'Ryan. The Design of a Real-time CORBA ORB using Real-time Java. In *Proceedings ISORC'02*, 2002.

[LO95] J.Y. Levy and J.K. Ousterhout. A Safe Tcl Toolkit for Electronic Meeting Places. In USENIX Association, editor, *Proc. of the 1st USENIX Workshop of Electronic Commerce*, pages 133–135, Berkeley, 1995. USENIX.

[LO98] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[Obj02] ObjectWeb Consortium. *The Fractal Component Model*, 2002. Specification, software, and documentation available at `http://www.objectweb.org/architecture/component/index.html`.

[OKS+00] C. O'Ryan, F. Kuhns, D. Schmidt, O. Othman, and J. Parsons. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. In *Proceedings MIDDLEWARE'00*, 2000.

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

[PS97] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proc. of the 1st International Workshop on Mobile Agents (MA '97)*, number 1219 in LNCS, pages 50–61. Springer-Verlag, 1997.

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

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

[Whi96] J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.

# A Binding and Communication

To clarify the notion of binding and communication, we reproduce the following excerpts, taken from the JONATHAN tutorial [Kra02] developped by the ObjectWeb consortium, of which France Telecom R&D and INRIA are founding members.

## A.1 Binding and Binding Objects

"*Binding* is the process of interconnecting a set of objects in a computing system. The result of this process, i.e. the association, or link, created between the bound objects, is also called a binding. The purpose of binding is to create an access path through which an object may be reached from another object. Thus a binding associates one or several sources with one or several targets.

The simplest form of binding is the association of a name with the object that it designates (a naming context is a set of bindings). Actual access from an identifier `id` to the object that it designates is carried out through the operation `id.bind()`, which returns the bound object. The object needs to have been previously made available in a naming context `nc` through the operation `nc.export(obj)`.

More elaborate bindings may take place, e.g. between a client and a remote server, or between several parties linked by a group communication protocol. The export-bind pattern

is also used in these cases. A complex binding is made up of a set of parts (e.g. a client stub, a communication session and a server stub), which make up a *binding object*."

Consider two objects `O1` and `O2`. "[...] If `O1` and `O2` belong to the same address space, `O1` can directly invoke `O2` through its reference (an address in memory). If `O1` and `O2` belong to different address spaces, on possibly heterogeneous platforms, the invocation uses a chain of intermediate objects. The ODP Reference Model of Open Distributed Processing (RM-ODP) [...] defines the notion of binding object to represent this situation. The simplest form of a binding object is a direct reference to the object (e.g. an address). The general form is a composite, distributed object that encapsulates the chain of intermediate entities through which the target object may be reached. For instance, in the case of a remote method call, the binding object encapsulates the client stub, the bidirectional communication channel used to transmit messages between the client and the server and the server stub (or skeleton). A binding object may take different forms, to represent different kinds of bindings (e.g. one to one, one to many, group communication, etc.). A binding object has a local representative in the address space of each of the objects that it binds; thus interaction between a client or server and the binding object is purely local. All remote communication takes place within the binding object. A binding object has a type, which represents the set of properties associated with the binding: protocols used, quality of service constraints, etc.

[...] A *binding factory*, also called *binder* is a special form of a naming context, which can create binding objects. Thus a binder implements the `bind` operation. [...] A binder also implements the `export` operation, which returns a handle (usually an identifier) in a specified naming context for an object managed by the binder. If there exists a chain of contexts between that naming context and the binder, the `export` operation is recursively called, and walks through this context chain." For instance, "a communication protocol (more precisely, a protocol graph) is a binder that exports (gives access to sessions) through session identifiers."

## A.2 The `export-bind` Pattern

"A *name* is an information that designates an object, i.e. allows the object to be unambiguously referred to by the name. A name has no absolute meaning; it is only valid in a particular environment called a *naming context*. A *naming context* is a set of pairs ($name_i$,$object_i$) where $name_i$ designates $object_i$. A name may take different forms, depending on the context (object reference, integer...)."

In the Core Framework, examples of identifiers are the following:

- "`Identifier` is used to designate applicative object interfaces that need to have specific properties (e.g. distributed or persistent). These interfaces are associated with naming contexts of type `NamingContext`.
- `SessionIdentifier` is used to designate interfaces (sessions) used in communication protocols. These interfaces are associated with naming contexts of type `Protocol`.

A naming context is itself an object, which may be designated by an identifier in another naming context. This allows chains of names to be built, in which the path from an identifier to an object may go through several successive naming contexts. [...] A common organization is a tree-structured context, in which names are identifiers built of strings sperated by separators (e.g. `a/b/c`)."

The binding model may be viewed as follows:

- "The object to be bound (the target of the binding) "advertizes" its availability for access in a naming context. The corresponding operation, `export`, is not borne by the object itself, but by the target naming context and may take different forms. Typically, if `target_nc` is a naming context and `obj` an object, then `handle = target_nc.export(obj, parameters)` delivers an object handle that gives access to object `obj` and is known in `target_nc`. [...] Others parameters may be used to specify additional properties such as quality of service. The effect of `export` may be cancelled by an `unexport` operation, which is usually borne by the exported name [...]

- An object that holds an identifier `cid` for object `obj` in a naming context `client_nc` may create a binding to `obj` through the method call `s = cid.bind()`. Provided `obj` has been exported, this call delivers an object `s` through which `obj` can be reached. `s` may be the object `obj` itself, but usually is some kind of surrogate (or substitute) for `obj`, e.g., a stub.

[...] A typical `bind-export` situation is that of a client-server system. The server exports the interface of a service that it provides. A client binds to this interface, using a name that represents it.

[...] In addition to `export`, `unexport`, and `bind`, the main operations regarding naming are related to the management of identifiers. Since identifiers may need to be transmitted on a network, they may need to be encoded into a byte array, and decoded upon reception. The operations `encode` and `decode` respectively perform the encoding and decoding. While encoding is borne by the `Identifier` interface, decoding must be borne by each destination naming context.

- `byte_array = id.encode()` encodes `id` into a byte array `byte_array`.

- `id = nc.decode(byte_array)` decodes `byte_array` into an identifier `id` in the target naming context `nc`.

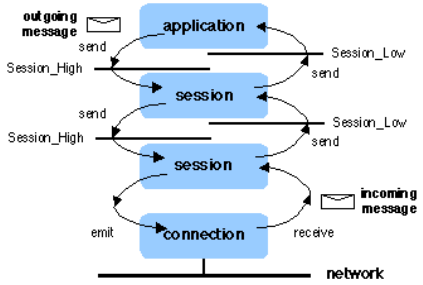
The last operation on identifiers is `resolve`. Recall that a name may be a referencing chain, consisting of a sequence of names valid in intermediate naming contexts. The operation `id.resolve()` returns the next name in the referencing chain, if such a name exists (`null` otherwise). It may be considered as the dual of `export` (`export` adds a name in front of a referencing chain, `resolve` removes the first name)."

### A.3 Sessions and Protocols

"A *protocol* (a basic abstraction for communication) provides a communication service for exchanging messages through a network. A protocol typically works by using the services provided by a lower level protocol, down to the hardwired communication interface. This defines a layered organization such as a protocol stack or acyclic graph. [...]

A *session* is an object that represents a communication channel. It provides an interface for sending and receiving messages; actually two different interfaces (`Session_Low` and `Session_High`) are respectively provided for incoming and outgoing messages. A protocol

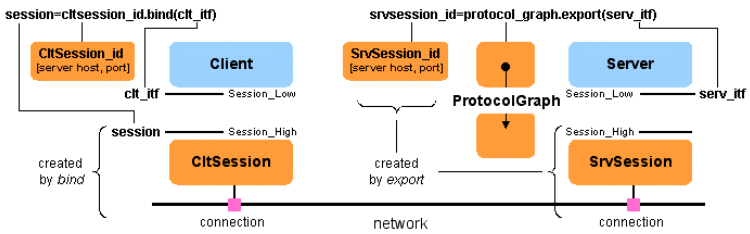
is essentially a session manager: it creates sessions, acts as a naming and binding context for these sessions, and provides them with communication resources. Like protocols, sessions are organized in a hierarchy. At the lowest level, a session relies on a basic communication mechanism called a *connection*, which provides an interface to send and to receive elementary messages (sequences of bytes). [...]




The main communication primitives are message sending and receiving. They operate in different ways, due to the asynchronous nature of receiving. A read operation (implemented by a `receive()` method on a connection) blocks the executing thread until data is available on the input channel associated with the connection. When data becomes available (a message has arrived), the thread is unblocked, causing the message to be passed up the protocol stack by calling the "lower" interfaces of the sessions, in ascending order. On the other hand, an application process sends an outgoing message by calling the "higher" interface provided by a session. The message is then sent down the protocol stack by calling "higher" interfaces in descending order, down to the call of an `emit` method on the connection."

## A.4 Building a Stack of Sessions

"On the server side, a protocol graph is first constructed by assembling elementary protocols. The protocol graph is a naming context, which provides the `export` method. The exported interface (`srv_itf`) is the "lower" interface of a session (of type `Session_Low`), which provides the functionality of the server. The `export` method returns a session identifier (a name for the exported interface), which contains all the information needed to set up a communication with the server (e.g., for TCP/IP, the IP address of the server and a port number). This information may be transmitted over the network and decoded by a client.




In order to be able to access the interface exported by a server, a client must call the `bind` method provided by a session identifier that designates the server, passing the client application's "lower" interface (`clt_itf`) as a parameter. The session identifier may be obtained

from the network (e.g. through a name service), or it may be constructed locally using the server address and port number if these are known. The `bind` method returns an interface session of type `Session_High`, which may be used by the client to call the server. Messages from the server are directed to the client application, through the interface `clt_itf` provided as a parameter of the call to `bind`."