MIKADO
Mobile Calculi based on Domains

MIKADO Global Computing Project
IST-2001-32222

# Virtual Machine Technologies: Core Software Framework v2

MIKADO Deliverable D3.1.3

**Editor :** R. De Nicola (U. of Florence)

**Authors :** L. Bettini, R. De Nicola (U. of Florence),
M. Lacoste (FTR&D), M. Loreti (U. of Florence)

**Classification :** Public
**Deliverable no. :** D3.1.3
**Reference :** RR/WP3/3
**Date :** February 2005

ist

Project funded by the European Community under the "Information Society Technologies" Programme (1998–2002)

## Abstract

This document discusses the progress in the implementation of the MIKADO Core Software Framework, that we call IMC (*Implementing Mobile Calculi*). The framework has been developed to build run-time support for languages oriented aiming at programming global computing applications. It enables platform designers to customize communication protocols and network architectures and guarantees transparency of name management and code mobility in distributed environments. A first version of the IMC framework was delivered last year [BFN$^+$04]. The version described here has been completely re-designed and implemented with the aims of guaranteeing better usability and additional features. The actual changes have been prompted by the actual use of the framework by two units of the project. The IMC framework has been used to re-engineer the KLAVA package (the runtime support for KLAIM) and to implement two variants of $D\pi$, one of the reference calculi for the mikado domain based model. Investigation have started to consider the embedding the concept of membrane within IMC that have lead to guidelines that will be used for further implementations. The framework presented here will be released as open source software. The actual code of the different IMC components can be inspected at the following address: `http://music.dsi.unifi.it/software/`

## Contents

## 1 Introduction

Technological advances of both computers and telecommunication networks, and development of more efficient communication protocols are leading to an ever increasing integration of computing systems and to diffusion of so called Global Computers [Car99]. This is a massive networked and dynamically reconfigurable infrastructure interconnecting heterogeneous, typically autonomous and mobile components, that can operate on the basis of incomplete information.

Global Computers are fostering a new style of distributed programming that has to take into account variable guarantees for communication, cooperation and mobility, resource usage, security policies and mechanisms. This has stimulated the proposal of new theories, computational paradigms, linguistic mechanisms and implementation techniques for the design, realization, deployment and management of global computational environments and their applications.

We have thus witnessed the birth of many calculi and kernel languages intended to support programming according to the new style and to provide tools for formal reasoning over the modelled systems. We have also seen that, to assess the impact and quality of the proposals, many implementations of these formalisms have been proposed. Very often, the language used for implementation is Java [AGH00]. Indeed, Java provides many useful features for building network applications with mobile code: *object serialization*, to encode/decode object structure into/from a stream; *dynamic class loading*, to insert a new class dynamically into a running application, a wide *network class library* and language-based *synchronization* functionalities.

Indeed, Java [AGH00] provides many useful features that are helpful in building general network applications and in particular with network applications mobile code: *object serialization*, to encode/decode object structure into/from a stream; *dynamic class loading*, to insert a new class dynamically into a running application, a wide network class library and language based synchronization functionalities.

However, these above mentioned Java mechanisms still require a big programming effort, and so they can be thought of as “low-level” mechanisms. Because of this, many existing Java-based distributed systems (see, e.g., [LO98, ARS97, PS97, CLZ98, BDFP98, PMR99] and the references therein) tend to re-implement from scratch many components that are typical and recurrent in distributed and mobile applications.

To support the implementation of languages for global computing, we have been working on a generic framework called IMC (*Implementing Mobile Calculi*) that can be used as a kind of middleware for the implementation of different distributed mobile systems. Such a framework aims at being as general as possible and at providing the necessary tools for implementing new language run-time systems directly derived from calculi for mobility. The basic idea and motivation of this framework is that the implementer of a new language would need concentrating on the parts that are really specific of his system, while relying on the framework for the recurrent standard mechanisms. The development of prototype implementations should then be quicker and the programmers should be relieved from dealing with low-level details. The proposed framework aims at providing all the required functionalities and abstractions for arbitrary components to communicate and move in a distributed setting.

After analysis of different kernel languages for mobility, we singled out four components described below as a foundation for IMC:

**Communication Protocols** provides abstractions and reference implementations to build customized communication protocols.

**Code Mobility** provides the basic functionalities for making code mobility transparent to the programmer. It deals with object marshalling, code migration, and dynamic loading of code.

**Node Topology** manages the topological structure of the network and its components. It deals with primitives for connection and disconnection, node creation and deletion and node-based decentralized topology.

**Naming and Binding** defines a uniform way to designate and interconnect the set of objects involved in the communication paths between computational nodes. It deals with primitives for name creation and deletion, typing and policies for name resolution.

IMC already provides concrete implementations for the standard and most used functionalities that should fit most Java mobile framework requirements (e.g., Java byte-code mobility and standard network communication mechanisms). The user of the IMC package can then customize parts of the framework by providing its own implementations for the interfaces used in the package. In this respect, the IMC framework will be straightforward to use if there is no need of specific advanced features. Nevertheless, the framework is open to customizations if these are required by the specific mobility system one is willing to implement. Customization of the framework can be achieved seamlessly thanks to design patterns such as *factory method*, *abstract factory*, *template method* and *strategy* [GHJV95] that are widely used throughout the package.

The framework was designed to achieve both *transparency* and *adaptability*. For instance, concerning code mobility, the framework provides all the basic functionalities for making code mobility transparent to the programmer: all issues related to code marshalling and code dispatch are handled automatically by the classes of the framework. Its components are designed to deal with object marshalling, code migration, and dynamic loading of code. The framework can also be adapted to deal with many network topologies (flat, hierarchical, peer-to-peer networks, etc.) and with message dispatching and forwarding. Furthermore, the implementer can build his own communication protocols by specializing the protocol base classes provided by the framework. Thus, the developer will only have to implement the parts that are relevant to the system he wants to build: typically, he will develop the communication protocol which best matches application-specific requirements. Connections and network topology are dealt with directly from within the framework. However, the developer can access the current state of its application at any time by using listeners to events that the classes of the framework generate.

The main intent of the IMC framework is not to be "yet another" distributed mobile system. It should rather be seen at a meta-level, as a framework/toolbox for building "yet another" distributed mobile system. A first version of the IMC framework was delivered last year [BFN$^+$04]. The version described here has been completely re-designed and implemented with the aims of guaranteeing better usability and additional features. The actual changes have been prompted by the actual use of the framework by two units of the project. Indeed, we have used the IMC framework to re-engineer the KLAVA package (the runtime support for KLAIM) and to implement two variants of $D\pi$, one of the reference calculi for the mikado domain based model.

In the rest of this document we shall describe the different components of the revised IMC and shall present a couple of small examples aiming at showing the flexibility and potentiality of the described framework.

The framework presented here will be released as open source software.The actual code of the different components can be inspected at the following address: `http://music.dsi.unifi.it/software/`

## 2 Overview of the IMC Framework

In this section we sketch the main parts of the framework, their interfaces and functionalities. For the sake of simplicity we will not detail all the method signatures, e.g., we will not show the exceptions.

### 2.1 Communication Protocols

When implementing a distributed system, one of the system-specific issues is the choice of the communication protocol, which may range from high-level protocols such as Java RMI, well integrated with the Java Virtual Machine environment and taking advantage of the architectural independence provided by Java, to protocols closer to hardware resources such as TCP/IP. Marshalling strategies may range from dedicated byte-code structures to Java serialization. A generic communication framework [HHD98, Exo02, OKS$^+$00, KSO02, DHDS98] should strive to be minimal, and allow to introduce support for new protocols with little effort, without need to re-implement a new communications library.

Thus, IMC provides tools to define customized protocol stacks, which are viewed as a flexible composition of micro-protocols. The IMC design, inspired from the *x*-kernel [HP91] communication framework, allows to define *bindings* with various semantics, and to combine them in flexible ways. In other words, with simple architectural principles such as separating marshalling from protocol implementation, IMC allows to create adaptable access and communication paths between components of a distributed system with a wide variety of semantics: mobile, persistent, with QoS guarantees, etc. Thus, IMC enables to achieve adaptable forms of communication transparency, which are needed when implementing an infrastructure for global computing.

In the IMC framework, a *network protocol* like, e.g., TCP, UDP, or GIOP is viewed as an aggregation of *protocol states*: a high-level communication protocol can indeed be described as a state automaton. The programmer implements a protocol state by extending the `ProtocolState` abstract class and by providing the implementation for the method `enter`, which returns the identifier of the next state to execute. The `Protocol` class aggregates the protocol states and provides the *template method* [GHJV95] `start` that will execute each state at a time, starting from the first protocol state up to the final one. Thus, the programmer must simply provide the implementation of each state, put them in the correct order in a protocol instance, and then start the protocol.

```
public class Protocol {
 public void start() { /* executes the states */ }
}

public abstract class ProtocolState {
 public abstract String enter();
}
```

The protocol states abstract from the specific communication layer. This enables re-using of a protocol implementation independently from the underlying communication means: the same protocol can then be executed on a TCP socket, on UDP packets or even on streams attached to a file (e.g., to simulate a protocol execution). This abstraction is implemented by specialized streams: `Marshaller` (to write) and `UnMarshaller` (to read). These streams provide high-level and encoding-independent representations of messages that are about to be sent or received, i.e., they are basically an extension of standard `DataOutput` and `DataInput` Java streams, with the addition of means to send and receive migrating code (explained later) and serialize and deserialize objects. The interface of `UnMarshaller` is the following (the interface of `Marshaller` contains the corresponding write instead of read methods):

```
public interface UnMarshaller extends DataInput, Closeable, MigratingCodeHandler {
 public Object readReference();
 public MigratingCode readMigratingCode();
 public MigratingPacket readMigratingPacket();
}
```

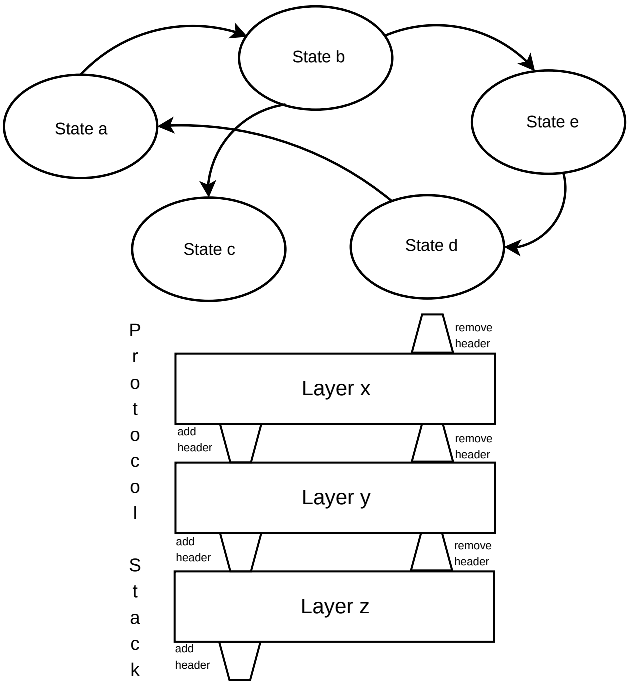

Figure 1: Protocol States and Protocol Stack Abstractions.

The data in these streams can be "pre-processed" by some customized *protocol layers* that can remove some information from the input and can add some information to the output: typically this information are protocol specific headers removed from the input and added to the output. A protocol layer is an abstract representation of a communication channel which uses a given protocol. It lets messages be sent and received through the communication channel it stands for using that protocol. The base class `ProtocolLayer` deals with these functionalities, and can be specialized by the programmer to provide his own protocol layer. These layers are then composed into a `ProtocolStack` object that ensures the order of preprocessing passing through all the layers in the stack. The structure of a protocol instance can then be depicted as in Figure 1.

For instance, the programmer can add a layer that removes a sequence number from an incoming packet and adds the incremented sequence number into an outgoing packet. The framework also provides functionalities to easily implement *tunnels*, so that it can be possible, e.g., to implement a tunneling layer to tunnel an existing protocol into HTTP (see Section 3).

Before reading something from a stack, a protocol state must obtain an `UnMarshaller` instance from the stack by calling the method `up`: this allows the stack layers to retrieve their own headers. In the same way, before starting to write information to the network, the state must obtain a `Marshaller` instance from the stack by calling the method `prepare`, so that the stack layers can add their own headers into the output. When the state has finished to write, it must notify the stack by calling the method `down`, passing the marshaller instance it had used to write the information, in order to flush the output buffer.

The methods `up`, `prepare` and `down` are declared **public** and **final** in the base class `ProtocolLayer`: the subclasses should instead provide their own implementations for these functionalities in the methods `doUp`, `doPrepare` and `doDown`, respectively. The public methods in the base class will ensure that the methods implemented in the derived classes will be called in the right order so to implement the stack of layers.

```
public class ProtocolLayer {
  public final UnMarshaller up() { /* implementation of the framework */ }
  protected UnMarshaller doUp(UnMarshaller um) { /* implementation of the programmer */ }
  public final Marshaller prepare() { /* implementation of the framework */ }
```

```
  protected Marshaller doPrepare(Marshaller m) { /∗ implementation of the programmer ∗/ }
  /∗ similar for down() ∗/
}
```

The `UnMarshaller` returned by the lower layer in the stack is passed to the implementation method `doUp`; thus, a layer can use the passed `UnMarshaller` to retrieve its own header and pass the `UnMarshaller` to the next layer, or it can create a new `UnMarshaller` to pass to the next layer. The latter scenario is typical of tunneling layers (as briefly shown in Section 3). Similarly, the `Marshaller` returned by the lower layer is passed to `doPrepare`. Typically, the first `UnMarshaller` and `Marshaller` objects will be created by the lowest layer, e.g., in case of a TCP socket, it will be a stream attached to the socket itself, while, in case of UDP packets, it will be a buffered stream attached to the datagram contents. Low layers for TCP and UDP are already provided by the framework.

### 2.2 Code Mobility

When code (e.g., a process or an object) is moved to a remote computer, its 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, i.e., the classes needed by the moved process are collected and delivered together with the process; *on-demand* approach, i.e., the class needed by the remote computer that received a process for execution is requested to the server that did send the process. We follow the automatic approach because it complies better with the mobile agent paradigm: when migrating, an agent takes with it all the information that it may need for later executions. This makes the code migration completely transparent to the programmer, so that he will not have to worry about classes movement. Our choice has also the advantage of simplifying the handling of *disconnected operations* (the agent can execute even if the owner is not connected) [PR98]. 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. The framework also provides means to support a fully on-demand approach.

With the automatic approach, an object will be sent along with the byte-code of its class, and with the byte-code of all the classes of the objects it uses (i.e., all the byte-code it needs for execution). Obviously, only the code of user-defined classes must be sent, as other code (e.g., Java class libraries and the classes of the IMC packages) must be common to every application. This guarantees that classes belonging to Java standard class libraries are not loaded from other sources (especially, the network); this would be very dangerous, since, in general, such classes have many more access privileges with respect to other classes. The framework also allows the programmer to manually exclude other classes (or entire packages) from mobility.

The framework defines the empty interface `MigratingCode` that must be implemented by the classes representing a code that has to be exchanged among distributed sites. This code is intended to be transmitted in a `MigratingPacket`, stored in the shape of a `byte` array. How a `MigratingCode` object is stored in and retrieved from a `MigratingPacket` is taken care of by the these two interfaces:

```
public interface MigratingCodeMarshaller {
  public MigratingPacket marshal(MigratingCode code);
}

public interface MigratingCodeUnMarshaller {
  public MigratingCode unmarshal(MigratingPacket p);
}
```

Starting from these interfaces, the framework provides concrete classes that automatically deal with migration of Java objects together with their byte-code, and for transparently deserializing such objects by dynamically loading their transmitted byte-code. In particular, the framework provides the base class `JavaMigratingCode`, implementing the above mentioned interface, `MigratingCode`, that provides all the procedures for collecting the Java classes that the migrating object has to bring to the remote site:

```
public class JavaMigratingCode extends Thread implements MigratingCode {
  public JavaMigratingPacket make_packet() {...}
}
```

The method `make_packet` will be used directly by the other classes of the framework or, possibly, directly by the programmer, to build a packet containing the serialized (marshalled) version of the object that has to migrate together with all its needed byte-code. Thus, this method will actually take care of all the code collection operations. 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. Once these class names are collected, their byte code is gathered and packed along with the object in a `JavaMigratingPacket` object (a subclass of `MigratingPacket` storing the byte-code of all the classes used by the migrating object, besides the serialized object itself).

Finally, two classes, implementing the above mentioned interfaces `MigratingCodeMarshaller` and `MigratingCodeUnMarshaller`, will take care of actually marshalling and unmarshalling a `JavaMigratingPacket` containing a migrating object and its code:

```
public class JavaByteCodeMarshaller implements MigratingCodeMarshaller {...}
public class JavaByteCodeUnMarshaller implements MigratingCodeUnMarshaller {...}
```

In particular, the first one will basically rely on the method `make_packet` of `JavaMigratingCode`, while the second one will rely on a customized *class loader* provided by the framework (a `NodeClassLoader`) to load the classes stored in the `JavaMigratingPacket` and then on Java serialization to actually deserialize the migrating code contained in the packet.

The `readMigratingCode` method of the `UnMarshaller`, shown in Section 2.1, will rely on an a `MigratingCodeUnMarshaller` to retrieve a migrating object and the corresponding method in `Marshaller` will rely on a `MigratingCodeMarshaller` to send a migrating object, so that all the code mobility issues will be dealt with internally by the framework. Even in this case, the programmer can provide his own implementations of `MigratingCodeUnMarshaller` and `MigratingCodeMarshaller` so that the framework will transparently adapt to the customized code mobility.

### 2.3 Node Topology

The framework already provides some implemented protocols to deal with connections and disconnections (these protocols can be specialized or overridden by the programmer). With this respect, the concept of connection is logical, since it can then rely on a physical connection (e.g., TCP sockets) or on a connectionless communication layer (e.g., UDP packets). In the latter case, a keep-alive mechanism can be implemented. A `ConnectionManager` instance will keep track of all the connections.

This can be used to implement several network topology structures: a *flat* network where only one server manages connections and all the clients are at the same level; a *hierarchical* network where a client can be in turn a server and where the structure of the network can be a tree or, in general, an acyclic graph of nodes; or, a *peer-to-peer* network.

A participant of a network is an instance of the class `Node` contained in the framework. A node is also a container of running processes that should be thought of as the computational units. The framework provides all the means for a process to access the resources contained in a node and to migrate to other nodes. Thus, a developer of a distributed and mobile code system has all the means to start to implement its own infrastructure or the run-time system for a mobile code language. A process is a subclass of the class `NodeProcess` that implements the `JavaMigratingCode` base class (this allows to easily migrate a process to a remote site), and can be added to a node for execution with the method `addProcess` of the class `Node`.

A `NodeProcess` has the following interface:

```
public abstract class NodeProcess extends JavaMigratingCode {
```

```
  public abstract void execute();
  public final void run() {
    // framework initialization operations; then call execute()
  }
}
```

Thus, a node keeps track of all the processes that are currently in execution. A concurrent process is started by calling `start` on the `NodeProcess` thread; the final implementation of `run` will initialize the process structure (not detailed here) and then invoke *execute* that must be provided by the programmer.

A different kind of process, called *node coordinator*, is allowed to execute privileged actions, such as establishing a connection, accepting connections from other nodes, closing a connection, etc. Standard processes are not given these privileges, and this allows to separate processes that deal with node configurations from standard programs executing on nodes. For these processes a specialized class is provided called `NodeCoordinator`.

The programmer can provide its implementation of the concept of `NodeLocation` to address in a unique way a node in the net (e.g., the standard IP address:port representation). If there is a (logical) connection with a node, then a location is mapped by the connection manager into a protocol stack. Thus a process can retrieve a stack to run its own protocols with a remote node.

The framework also provides means to dynamically "manipulate" a protocol: it permits extending a protocol automaton by adding new states and extending the protocol stack by inserting new layers. With respect to the manipulation of the protocol automaton, it is possible to add a new starting state and a new final state, so that the original protocol is embedded in an extended protocol. When a new start and a new end state are added to an existing protocol, the framework will also take care of re-naming the previous start and end state and update all the references to the original start and end state with the re-named version. This will guarantee that the original protocol will transparently work as before internally, while from the outside, the new start state will be executed before the original start state and the new end state will be executed after the original end state.

The manipulation of a protocol is used internally by the classes of the framework, for instance in connection and disconnection management. The `Node` class provides a `connect` method to establish a connection and a method `accept` to accept a connection (these connections are both logical and physical). These methods, apart from the connection details (e.g., host and port) also take a protocol instance. These methods will take care of establishing (accepting, resp.) a physical connection, add a logical connect protocol state as the new start state and a logical disconnect state as the end state to the passed protocol. They also take care of setting the low layer in the protocol stack (e.g., TCP socket or UDP datagrams). Then, the protocol can be started. This manipulation is depicted in Figure 2.

### 2.4 Naming and Binding

The framework also supports logical name management, inspired by the Jonathan ORB [DHDS98]. The aim 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.

In the IMC framework, an *identifier* is a generic notion of name that uniquely designates an object in a given naming context. Identifier semantics are naming context-specific: distributed, persistent, etc. A *naming context* provides name creation and management facilities. It guarantees that each of the names it controls designates some object unambiguously. It 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. Finally, a *binder* is a a special kind of naming context that, for a given managed name, is able to create an access path, also called *binding*, towards the object designated by that name.

These definitions offer a generic and uniform view of bindings, and clearly separate object identification from object access:

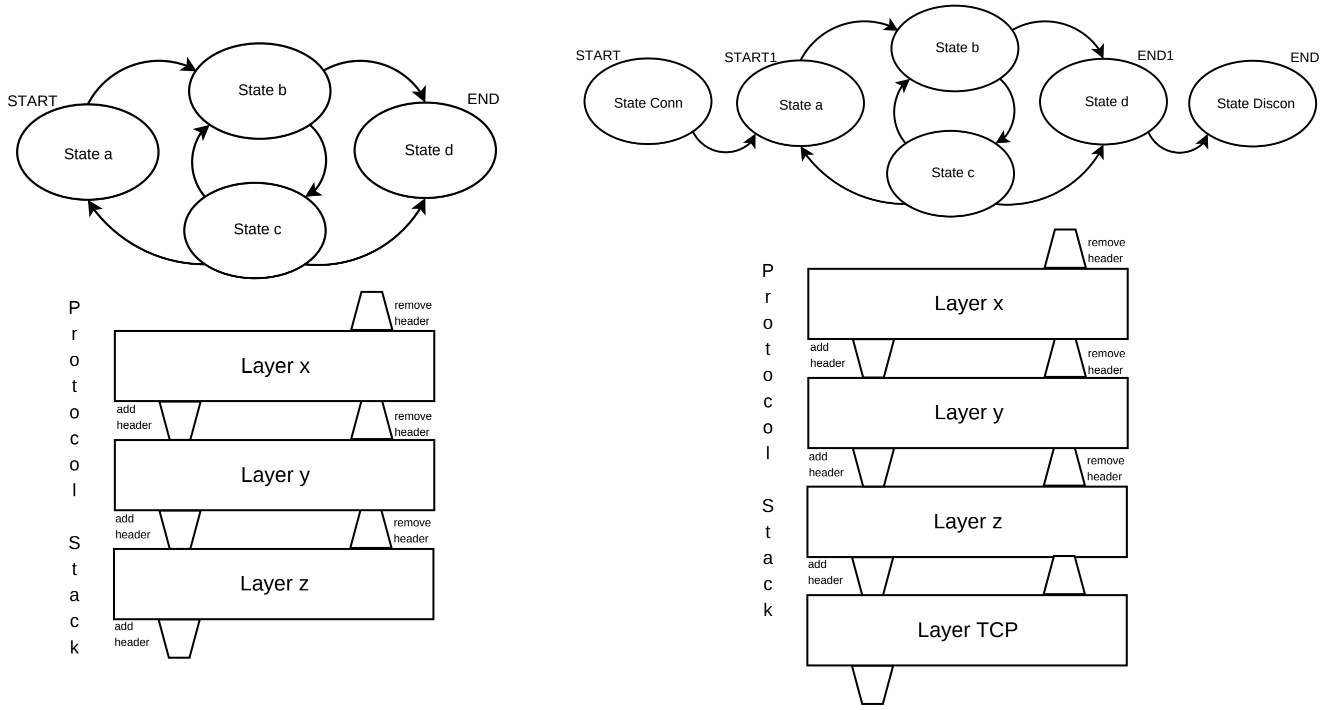

Figure 2: The original protocol (left) and the new protocol extended with a new start and end state and the TCP layer (right).

- 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 abstractions are reflected in the following Java interfaces:

```
public interface Identifier {
 public NamingContext getContext();
 public Object bind();
 public Object resolve();
}

public interface NamingContext {
 public Identifier export(Object obj);
}
```

The `Identifier` interface represents the generic notion of identifier described above. It contains a reference to its naming context, and bears the fundamental `bind` operation to set up a binding between two (possibly remote) objects. The interface, using the `resolve` method, also permits returning 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.

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.

Other methods, not represented here, deal with identifier transmission over the network, using encoding-

```
public class IncrementProtocolLayer extends ProtocolLayer {
  private int sequence;
  protected UnMarshaller doUp(UnMarshaller um) {
    sequence = um.readInt();
    return um;
  }
  protected Marshaller doPrepare(Marshaller m) {
    m.writeInt(sequence + 1);
    return m;
  }
}
```

**Listing 1:** A protocol layer that deals with sequence numbers.

```
public class EchoProtocolState extends ProtocolState {
  public String enter() throws {
    UnMarshaller um = up();   // start reading
    String line = um.readStringLine();
    Marshaller m = prepare();   // stop reading, start writing
    m.writeStringLine(line);
    down(m);   // finish writing
    return "END";
  }
}
```

**Listing 2:** An echo protocol state.

independent representations, namely involving the `Marshaller` and `UnMarshaller` interfaces already described.

This export-bind pattern is closely related to the communication part of the IMC framework: a `Protocol` object can be viewed as a binder which exports (i.e., builds an access path to) a communication end-point, a `ProtocolLayer` designated through a specific type of identifier, namely a protocol layer identifier. Typically, the `export` operation will be called by a server object to advertise its presence on the network. This will be translated into a call to the `accept` method of a `Node` object, to accept incoming network connections. The `bind` operation will be called by a client-side object to bind to the interface designated by a given identifier. This will be translated into a call to the `connect` method of the `Node` object, to establish the communication path to the remote server-side object.

## 3 Some Examples

In this section we will present some simple examples that show how the framework can be used to program a customized protocol. We will not show all the details of the code, but we concentrate on how the single objects developed by the programmer can be composed together and used from within the framework itself.

First of all, in Listing 1 we show a protocol layer that removes a sequence number from the input stream and writes the incremented sequence number in the output stream. Thus, when a protocol state starts reading this layer will remove this header and when a state starts writing this layer will add the incremented sequence number. Now we can create our protocol stack with this layer:

```
ProtocolStack mystack = new ProtocolStack();
mystack.insertLayer(new IncrementProtocolLayer());
```

Then we can implement our own protocol; for simplicity our protocol will consist of only one state, that does nothing but read a line and send that line back (an echo server); after that the protocol ends. In order to implement such a state we only have to extend the `ProtocolState` base class and provide the implementation for the method `enter` and return the state `END` as the next state in the protocol (Listing 2). We can then create our protocol instance, set the protocol stack, and add the start state:

```
Protocol myprotocol = new Protocol();
myprotocol.setStack(mystack);
myprotocol.setState("START", new EchoProtocolState());
```

The protocol is now built, but no communication layer has been set yet. In order to do so, we can use the `Node` class functionalities:

```
Node mynode = new Node();
mynode.accept(9999, myprotocol);
myprotocol.start();
```

These instructions wait for an incoming connection on port 9999, update the protocol with a starting connection state and a final disconnection state, and update the protocol stack with the low communication layer. At this point, the protocol can start on the established physical connection.

As we hinted in Section 2, the framework provides a specialized protocol layer base class, `TunnelProtocolLayer`, that permits implementing a tunneling layer, in order to envelop a protocol inside another one. A typical example is the one of an *http tunnel* that wraps a protocol in HTTP requests and responses. Notice that a tunnel layer does not simply remove a header when reading and add a header when writing: typically it will need to read an entire message, strip the tunneling protocol information, and pass to the upper layer the information that was wrapped; in the same way, it will need to intercept the information written by the upper layer and wrap it into a message according to the tunneling protocol. For this reason the framework provides this specialized base class with the features to implement these more complex functionalities.

In particular, `TunnelProtocolLayer` provides two piped stream pairs to allow the tunnel layer to communicate with the tunneled layer: the field `tunneledMarshaller` is piped with the field `newUnMarshaller` (i.e., everything that is written into `tunneledMarshaller` can be read from `newUnMarshaller`). So, the tunnel layer can implement the `doUp` this way:

```
public class HTTPTunnelLayer extends TunnelProtocolLayer {
 protected UnMarshaller doUp(UnMarshaller um) {
  String data = strip(readHTTPRequest(um));
  tunneledMarshaller.writeStringLine(data);
  return newUnMarshaller;
 }
```

Similarly the implementation of `doPrepare` will return to the tunneled layer a piped `UnMarshaller` and `doDown` will read the data written by the tunneled layer from the other end of the pipe, envelop the data in the tunnel protocol structure and pass everything to the lower layer by using the `Marshaller` originally returned by the lower layer's `prepare` method.

Since a tunneling layer is still a layer, it can be inserted smoothly in an existing protocol stack:

```
ProtocolStack mystack = new ProtocolStack();
mystack.insertLayer(new IncrementProtocolLayer());
mystack.insertLayer(new HTTPTunnelLayer());
```

The representation of the protocol after the call to `accept` is depicted in Figure 3. Let us stress that the insertion of the tunnel layer did not require any change to the existing protocol states and layers.

## 4 Conclusions

We have presented a Java software framework for building infrastructures to support the development of applications over global computers where mobility and network awareness are key issues. The framework enables platform designers to customize communication protocols and network architectures and is particularly useful to develop run-time supports for languages oriented towards global computing.

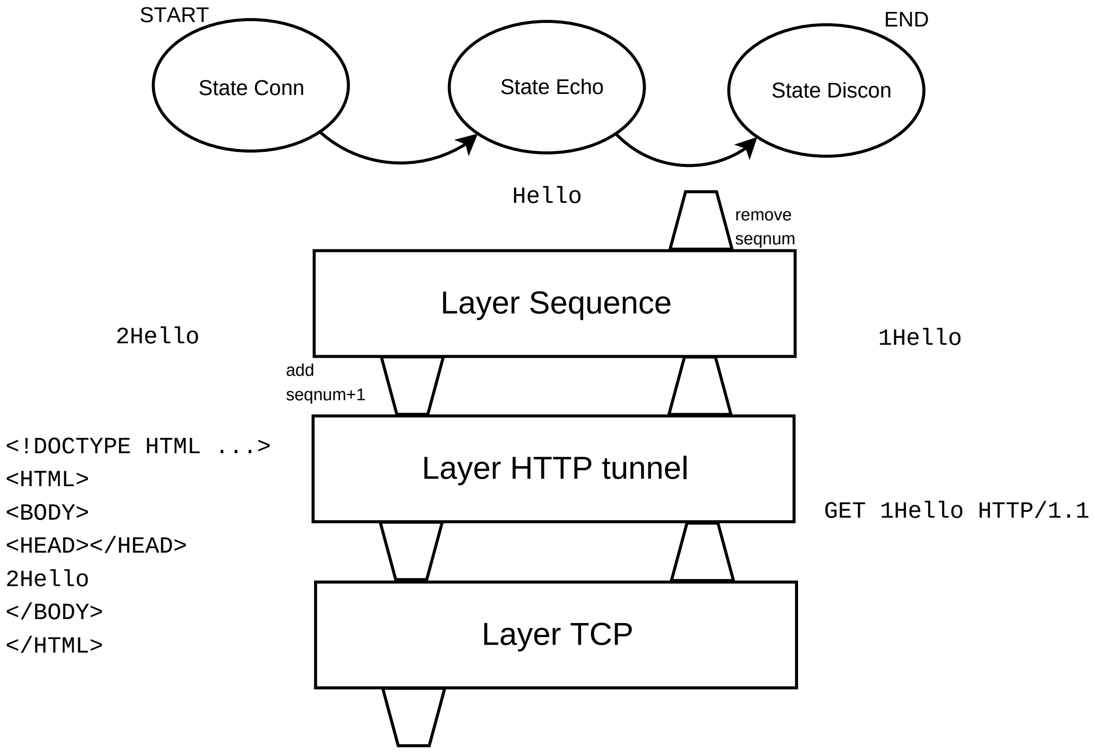

Figure 3: The graphical representation of protocol with the HTTP tunneling.

The components have been designed after a close analysis of proposed models for mobile computing [BCGL02, BBD+02]. We have tried to single out the most recurrent notions of network aware programming and packed them together. Developers can then concentrate on those parts that are really specific of their system, while relying on the framework for the recurrent standard mechanisms (node topology, communication and mobility of code).

The main aim of the framework is making the development of prototype implementations faster and relieving programmers from low level details. Of course, if applications require a specific functionality that is not in the framework (e.g., a customized communication protocol built on top of TCP/IP, or a more sophisticated mobile code management), programmers can still customize the behaviors that concern these mechanisms in the framework.

In a companion document, we report on experiments with the framework. IMC has been used by using it as the basis for implementing $D\pi$ [HR98] and to re-engineer KLAVA [BDP02], the run time support for KLAIM [BBD+03]. We have already started investigating on the one hand the use of the framework to implement richer languages for mobility and on the other hand how its components can be enriched or needs to be modified to take security issues into account. Indeed, in the companion document we will also report on a preliminary results and findings about the impact of the introduction of the concept of membrane within IMC and its components.

## References

[AGH00] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

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

[BBD$^+$02] L. Bettini, 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.

[BBD$^+$03] L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The KLAIM Project: Theory and Practice. In C. Priami, editor, *Global Computing. IST/FET International Workshop, GC 2003, Revised Papers*, volume 2874 of *LNCS*, pages 88–150. Springer, 2003.

[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.

[BDFP98] L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. WETICE*, pages 110–115. IEEE, 1998.

[BDP02] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. KLAVA: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.

[BFN$^+$04] L. Bettini, D. Falassi, R. De Nicola, M. Lacoste, L. Lopes, M. Loreti, L. Oliveira, H. Paulino, and V. Vasconcelos. Language experiments v1: Simple calculi as programming language. MIKADO Global Computing Project Deliverable D3.2.1, 2004.

[Car99] Luca Cardelli. Abstractions for Mobile Computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 51–94. Springer-Verlag, 1999.

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

[CLZ98] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In K. Rothermel and F. Hohl, editors, *Proc. of the 2nd Int. Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer, 1998.

[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.

[Exo02] ExoLab Group. The OpenORB project, 2002. Software available for download at `http://openorb.exolab.org/`.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HCK94] C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research Report 19887, IBM Research Division, 1994.

[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.

[HR98] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3, pages 3–17. Elsevier Science Publishers, 1998.

[Kna96] F. Knabe. An overview of mobile agent programming. In *Proc. LOMAPS*, number 1192 in LNCS. Springer, 1996.

[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.

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

[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.

[PMR99] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. ICSE'99*, pages 368–377. ACM Press, 1999.

[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 *Proc. MA*, number 1219 in LNCS, pages 50–61. Springer, 1997.

[Tho97] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997. Also Technical Report 1083, University of Rennes IRISA.

[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.