A Software Framework for Rapid Prototyping of Run-Time Systems for Mobile Calculi [⋆]

Lorenzo Bettini[1], Rocco De Nicola[1], Daniele Falassi[1], Marc Lacoste[2], Luís Lopes[3], Licínio Oliveira[3], Hervé Paulino[4], Vasco T. Vasconcelos[5]

[1]Dipartimento di Sistemi e Informatica, Università di Firenze.
[2]Distributed Systems Architecture Department, France Telecom R & D.
[3]Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto.
[4]Departamento de Informática, Faculdade de Ciências e Tecnologia, Univ. Nova de Lisboa.
[5]Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa.

Abstract. We describe the architecture and the implementation of the MIKADO software framework, that we call IMC (*Implementing Mobile Calculi*). The framework aims at providing the programmer with primitives to design and implement run-time systems for distributed process calculi. The paper describes the four main components of abstract machines for mobile calculi (node topology, naming and binding, communication protocols and mobility) that have been implemented as Java packages. The paper also contains the description of a prototype implementation of a run-time system for the Distributed Pi-Calculus relying on the presented framework.

1 Introduction

It has been widely argued that mobility will be an important technology for applications over a global network. The main breakthrough is that global applications may exchange *mobile code* [9, 33], not just data. A particular instance of mobile code is the concept of *mobile agents* [14, 19, 36]: software units that can suspend their execution and migrate to new hosts, where they can resume their execution.

The programming paradigm based on mobile agents is different from remote evaluation or code on-demand in that the code does not need to be sent and retrieved explicitly: the agent migrates autonomously somewhere else and continues executing there. An agent is self contained in that it possesses all the data it needs to execute and migrate, since this information is typically carried with the agent during migration. Decisions of executing and moving are taken according to information supplied by the programmer of the agent. Agents may also autonomously decide to take different choices due to contextual events such as, temporary unavailability of networks or not responding hosts.

Dealing with mobile code and mobile agents raises a number of issues such as: packing of moving agents, security, protocols, naming, network architecture. We are developing a generic framework called IMC (*Implementing Mobile Calculi*) that can

[⋆] This work has been funded by EU-FET on Global Computing, project MIKADO IST-2001-32222. The funding body is not responsible for any use that might be made of the results presented here

be used as a kind of middleware for the implementation of different mobile programming systems. Such a framework aims at being as general as possible. It can be, and indeed has been, used to implement existing systems (KLAVA [5], Safe Ambients [32], JCL [12] and DITYCO [26]) on top of it. But it also provides the necessary tools for implementing new languages directly derived from calculi for mobility.

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.

For the sake of dissemination and portability, the framework is being developed as Java packages. Thus, the used virtual machine technology is the one based on the Java Virtual Machine. This choice is also motivated by the fact that most existing mobile code systems are based on Java.

The proposed middleware framework aims at implementing (or, at least, specifying) all the required functionalities for arbitrary components to communicate and move in a distributed setting. Four abstractions have been isolated as being fundamental to this goal and each of them has been implemented as a sub-package of our IMC Java package:

- node topology
- naming and binding
- communication protocols
- mobility

The purpose of the sub-package for **Node Topology** 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. Its main components deal with primitives for connection and disconnection, node creation and deletion, domain specific node coordination (membrane, guardian, etc.), node-based decentralized topology and actual implementation of nodes and node naming.

The purpose of the sub-package for **Naming and Binding** is to define a uniform way to designate and interconnect the set of objects involved in the communication paths between computational nodes. We call such a set of objects a *binding*. Its main components are designed to deal with primitives for name creation and deletion, typing and compatibility checking, policies for name resolution (static, dynamic, mixed, bindings, ...) and name marshalling and un-marshalling.

The purpose of the sub-package for **Communication Protocols** is to identify both the abstractions and the primitives for logical and physical node connectivity, as well as the strategies that can be used to capture and perform communications between computational nodes. Its components are designed to deal with abstract, possibly secure, send and receive primitives, marshalling of messages at network level, session management, connection checking and multicasting.

The purpose of the sub-package for **Code Mobility** is to provide the basic functionalities for making code mobility transparent to the programmer; all issues related to code marshaling 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 general aim of the framework is to assist both the designer and the programmer of a virtual machine or run-time system implementing a domain-based programming model. The four components of the framework are connected and cooperate in order to implement the abstract representation of a distributed application with mobile code. Thus, for instance, the `topology` package relies on the `protocols` package to actually communicate over the network and the `protocols` package, in turn, relies on the `mobility` package to create packets containing migrating code, and so on. We observe that these cooperations are made through interfaces and abstract classes. Nonetheless, 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 TCP/IP sockets).

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 frameworks is open to customizations if these are required by the specific mobility system one is willing to implement. For example, the TyCO system makes use of its own code dispatch strategy and overrides the standard Java byte-code mobility. Customization of the framework can be achieved seamlessly thanks to design patterns such as *factory method* and *abstract factory* [13] that are widely used throughout the package.

The above mentioned sub-packages have been developed over the mikado sites by taking advantage of CVS server organized as a single project `org.mikado.imc` structured with four tasks:

– `org.mikado.imc.topology`
– `org.mikado.imc.naming`
– `org.mikado.imc.protocols`
– `org.mikado.imc.mobility`

The rest of the paper contains the detailed description of the four sub-packages and ends with an experimental implementation of *D*π, the only model considered within the Mikado project that has not yet been implemented.

2 Node Topology

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.

Motivation

The notion of node appears in most existing implementations of mobile calculi, such as, e.g., [4, 5, 12, 32, 34]. However, the internal structure of the node itself is programming-model specific. The computational nodes can include data structures ranging from definitions (TyCO, Jcl/JoCaml, CLAM), processes (X-Klaim/Klava, TyCO, SAM, JCL/JOCAML, CLAM) channels (TYCO, SAM, JCL/JOCAML, CLAM), objects (TYCO) and tuple spaces (X-KLAIM/KLAVA).

From this observation we designed the Node Topology package so that it would provide both a programming abstraction and a generic concrete implementation for a node whilst not providing any details of the specific implementation of the virtual machine that will run on it.

Another common property of the current mobile calculi implementations is that the nodes are designated to use some form of unique identifier. It is noteworthy that the structure of the node identifier is itself implementation specific. Also, node identifiers must be created dynamically when new nodes are added to a network.

Thus, we require the Node Topology framework to provide both an abstraction for a node identifier that encapsulates its implementation details and, some mechanism to create fresh node identifiers in accordance with some implementation-specific format.

In the existing implementations of mobile calculi, the topological organization of nodes is either hierarchical, i.e., tree-structured, or flat. For example, SAM, JCL/JOCAML, X-KLAIM/KLAVA and CLAM use a tree-structured topology of nodes, while TYCO nodes are organized according to a flat structure. Primitives for expressing a topological hierarchy of nodes can easily be used to reflect a flat organization of nodes by adding a root (virtual) location whose children are the given nodes. Thus, we feel that the framework should support hierarchical node composition patterns. This in turn implies providing tools to navigate through the network hierarchy and retrieve information about its structure.

Finally, some process calculi have reduction rules that imply adding new nodes to the hierarchy or removing existing nodes (or moving them). This can be due to the strict implementation of the reduction rules, or to new components dynamically being introduced into a running system. Thus, the Node Topology framework should provide primitives to add new nodes to the network hierarchy or to remove existing nodes.

Design

The above requirements lead to the following design for the Node Topology sub-package in the form of Java interfaces. Concrete and generic default implementations of these interfaces are also provided in the sub-package and named by prefixing the interface name with `Mikado` (e.g., a default implementation for interface `Node` is provided by the class `MikadoNode`).

```
interface NodeIdentifier {
  public Object getIdentifier();
}

interface Node {
  public String getNodeName();
  public NodeIdentifier getNodeIdentifier();
  public Object getImplementation();
  public NodeIdentifier getParentNode();
  public NodeIdentifier [] getChildNodes();
  public void connect(NodeIdentifier nodeId);
  public void disconnect();
  public Registry getRegistry();
}
```

The `NodeIdentifier` interface defines a generic way to identify a computational node component. Implementation-specific node 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 or agent, according to the underlying programming model. It contains a reference to its node identity interface, as well as a reference to an object holding the internal implementation of the node. That particular structure is implementation specific, and is not part of the Node Topology sub-package.

The requirements of a hierarchical network topology and support for editing such a hierarchy lead to the introduction of topology management functionality directly within the `Node` interface. Thus, the methods `getParentNode()` and `getChildNodes()` allow a node to inspect its network neighborhood. The methods `connect()` and `disconnect()` handle a node's connection to a specific point (as a sub-node) of the hierarchy and its disconnection when leaving the network or migrating to another point in the hierarchy.

The method `getRegistry()` provides access to a node's table of exported resources (e.g., channels) and will be further commented when describing the Naming and Binding sub-package.

A special node in a Mikado Network, the `NetworkServer`, acts as a portal where all nodes adhering to a computation must first register. The `NetworkServer` handles the mappings between nodes and their physical locations (e.g., IP addresses) in a Mikado Network. The network server accepts different implementations and network configurations that can be specified at startup by providing an implementation for the interface `NetworkServerImpl` and a class holding the network configuration called `Preferences`. These settings and the current server handle can be obtained through a set of methods (`getImpl()`, `getServer()` and `getConfig()`).

```
class NetworkServer {
  NetworkServer(String [] args);
  static NetworkServer getServer();
  static NetworkServerImpl getImpl();
  Preferences getConfi g();
  ...
}
```

Examples

IMC's `topology` revolves around the interface `Node`. This interface represents a running instance of a virtual machine and includes a set of operations that manipulate that same VM instance. To enforce interoperability between the different virtual machines that may end up subclassing IMC, a base implementation for the interface `Node` has already been provided, in the form of the `MikadoNode` class.

In TyCO, the abstraction for a node in a network is implemented by a class `Site`. Since `Site` cannot subclass any other class (it already subclasses TyCO's `TyCOVirtualMachine` class and Java does not allow multiple inheritance), we create a wrapper class, `TyCONode`, that holds an instance of the `TyCOVirtualMachine` class.

```
public class TyCONode extends MikadoNode implements ... {
  Object virtualMachine;
  TyCONode (String name, Object virtualMachine) {
    super(name);
    this.virtualMachine = virtualMachine;
  }
  public Object getImplementation() {
    return this.virtualMachine;
  }
  ...
}
```

To create a new node running a TyCO virtual machine and add it to a network of running nodes one has to create a new instance of the class `TyCONode` supplying the node's name and the virtual machine running in it. To attach/detach the node to/from the network we need only to call the superclass' (`MikadoNode`) methods: `connect()` and `disconnect()`.

```
public class NodeManager {
  ...
  void newNode(String name, Assemble assembly) {
    TyCOVirtualMachine virtualMachine = new TyCOVirtualMachine(name, assembly);
    TyCONode node = new TyCONode(name, virtualMachine);
    node.run();
    node.connect();
  }
  ...
}
```

We assume the existence of a `connect()` wrapper in the `TyCONode` that skips the `NodeIdentifier` argument and automatically connects the new TyCO node to the `NetworkServer` that serves as the root of the flat network topology of the TyCO network.

An additional, optional, step is the inclusion of a `TopologySecurityManager`, which controls access to `MikadoNode`'s functionality. It allows an operation to be blocked or allowed, based on any desired security policy. The default `TopologySecurityManager` for TyCO would enforce rules such as: a node can only connect to the `NetworkServer` (its parent) and that it cannot accept connections (since the topology is flat).

The IMC infrastructure already contains a `NetworkServer` that handles the mappings between the node names and their physical location in a network. This enables a straightforward implementation of the TyCO's Name Server by using it as the virtual root of the TyCO flat network topology and extending the class with functionality for registering and type-checking top level exported channels.

3 Naming and Binding

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

Motivation

The fundamental concept to be provided by this sub-package is that of a *referenceable* object. Such an object is an abstraction for the fundamental communication peers in process calculi such as channels or definitions. A referenceable object is always associated with a unique network-wide *identifier*. Each resource identifier is uniquely associated with a *naming context* in a network.

A feature common to current mobile calculi implementations is the use of the export/bind programming pattern to make objects available in a network and to get an access path for such objects. This pattern is so pervasive that the Naming and Binding sub-package provides a *registry* abstraction that, for a given managed name, should be able to make it available to the network by registering it and to create an access path towards *the* object designated by that name. Thus, the registry is responsible for keeping the mappings between identifiers and referenceables for a given node in a network.

The above considerations offer a generic and uniform view of bindings, clearly separating object identification from object access.

Design

We now describe a minimal set of interfaces for dealing with naming and binding based on the above requirements:

```
public interface UID {
  public NamingContext getContext();
  public String getName();
  public String getEncoded();
  public NodeIdentifi er getNodeIdentifi er();
  public String toString();
}

public interface NamingContext {
  public String getName();
}
```

The `UID` interface represents the generic notion of a network wide unique identifier used to designate some object relatively to a given naming context, such as a channel in process calculi. Identifiers are implementation dependent. The interface contains a reference to its naming context. The `Naming Context` interface represents a set of `Referenceable` (see below) objects in a Mikado Network that is identified by a string.

```
public interface Referenceable {
  public NamingContext getContext();
  public UID getUID();
  public void handleData(ProxyRequest request);
  public void marshall();
  public void unmarshall();
}
```

The `Referenceable` interface must be implemented by any object in a Mikado Network that is to be exported and bound during a computation. The `handleData()` method is used to receive requests from object proxies (`Proxy`) elsewhere in the network. The `marshall()` and `unmarshall()` methods can be used, respectively, to prepare a reference for network dispatch and to restore a reference after traveling through the network. These methods typically call a `Marshaller` implementation from the `protocols` sub-package to perform some level of packing/unpacking (see Section 4).

```
public abstract class Proxy {
  private UID uid;
  protected Proxy(UID uid);
  public UID getUID();
  public abstract void dispatch(Serializable request);
  public abstract void marshall();
  public abstract void unmarshall();
}

public interface ProxyRequest {
  public abstract NodeIdentifi er getPeer();
  public abstract UID getUID();
  public abstract Serializable getRequest();
}
```

The `Proxy` interface describes the functionality associated with a proxy for a referenceable object. The method `dispatch()` handles communication by redirecting it to the corresponding referenceable object. Methods `marshall()` and `unmarshall()` have similar functions to the referenceable side. The interface `ProxyRequest` allows a referenceable object to obtain basic topological and naming information about a proxy sending data from another node and the data itself.

```
public interface Registry {
  public void export(Referenceable ref);
  public void unexport(Referenceable ref);
  public Proxy bind(NodeIdentifi er id, String name, NamingContext context);
  public Referenceable getRef(UID uid);
  public Referenceable [] getAllRefs();
}
```

Object access is provided for by the interface `Registry` which must be implemented by any class that exports or binds objects in a Mikado Network. The interface includes the `export()` method to create a new name in a given context and make it bindable in a network. The `unexport()` method cancels an `export` operation by making an identifier no longer valid within a naming context. In other words, the mapping identifier-referenceable object designated by that identifier is broken. The `bind()`

method returns a local Proxy associated with a given Referenceable object at a node `id`, with a given `name` and naming context `context`. This Proxy allows direct communication with the proxy for the Referenceable object in the Mikado Network.

All the required communication between referenceable objects and their proxies is provided via the protocols sub-package of the framework.

Examples

A default implementation of some of the `Naming` and `Binding` package interfaces (`Proxy`, `ProxyRequest`, `NamingContext` and `Registry`) is already provided in IMC (`MikadoProxy`, `MikadoProxyRequest`, `MikadoNamingContext` and `MikadoRegistry`, respectively).

The fundamental step in implementing TYCO on the IMC framework is the realization that TYCO's referenceable objects are instances of the class `Channel`. Also, given the flat topology of TYCO networks, the implementation requires only a single naming context identified by the string `"tyco"`.

In this approach, we make each exported TYCO channel in a running virtual machine implement the Referenceable interface and allow other nodes in the network to communicate with it directly by using proxies through the `Proxy` interface. The most important method in this implementation is `handleData()`. This method handles proxy requests to the channel from proxies elsewhere in the TYCO network. The incoming requests, messages or objects are either enqueued in the channel queue or reduced immediately if an adequate message-object redex forms.

```
public class Channel extends ... implements Referenceable {
  ...
  public void handleData(ProxyRequest request) {
    // unpack the request and check whether it is an object or a message
    Frame frame = unpack(request.getRequest());
    // run code according to case
    if( status == 0 ) {   // channel is empty
      enqueue(frame);
      if ( frame.isObject() )
        status++;
      else
        status−−;
    } else if ( status < 0 ) { //channel has messages
      if ( frame.isObject() ) {
        Frame message = dequeue();
        reduce(frame, message);
        status++;
      } else {
        enqueue(frame);
        status−−;
      }
    } else if ( status > 0 ) { // channel has objects
      if ( frame.isObject() ) {
        enqueue(frame);
```

```
            status++;
          } else {
            Frame object = dequeue();
            reduce(object, frame);
            status−−;
          }
        }
    }
    ...
}
```

TYCO represents channels in a distributed computation in two distinct formats reflecting their current position relative to their lexical bindings. A local channel to a node is represented as a JVM heap reference. A remote channel is represented in a network format containing information about the node (its name) it originated from and its local reference there. When, say, a message is sent to a channel in a program running at some node of a TYCO network, the internal representation of the channel is first checked to see if the channel is local to the node or if it is a remote channel. If the channel is remote, a `bind` operation is required to get a proxy to handle remote interaction. Thus, a TYCO Virtual machine instruction to handle message delivery would look like this:

```
void sendMessage(Channel channel, Label label, Value[] args) {
  if (channel.isLocal()) {
    // code for local handling, similar to above code for handleData()
    ...
  } else {
    // get a proxy object for communication with the real channel
    UID uid = channel.getUID();
    NodeIdentifi er nodeId = uid.getNodeIdentifi er();
    String name = uid.getName();
    NamingContext context = uid.getContext();
    Proxy proxy = getMikadoNode().getRegistry().bind(nodeId, name, context);
    // pack message in a Serializable object and send it to the channel
    SerializablePacket packet = new SerializablePacket(channel, label, args);
    proxy.dispatch(packet);
  }
}
```

Exporting and importing top-level channels in TYCO involves two operations in the TYCO Virtual Machine that interacts with the NetworkServer. When we `export` one channel from a node we make its access information available to other nodes in the network. Such an operation might be implemented using the above abstractions as:

```
void export(Channel channel) {
  getMikadoNode().getRegistry().export(channel);
}
```

The complementary operation in which we `import` a top-level channel from some node in a TYCO network requires the name of the channel being requested and the node it resides in. The operation simply requests a proxy for the remote channel based on the name of the channel and of the node:

```
Proxy import(String node, String name) {
  NodeIdentifi er nodeId = getMikadoNode().resolve(node);
  NamingContext context = new TycoNamingContext();
  return getMikadoNode().getRegistry().bind(nodeId, name, context);
}
```

4 Communication Protocols

This part of the IMC framework intends to identify the primitives and the communication strategies for logical and physical node connectivity. The general aim is to assist the architect of a run-time system for a distributed process calculus in the implementation of new communication protocols between computational nodes.

Motivation

Existing implementations [3] of some common distributed process calculi [6] are characterized by a flurry of communication protocols and of programming languages. In first approximation, the 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 [32] implementation of Safe Ambients [25]); protocols closer to hardware resources such as TCP/IP are accessible, either directly in Java (X-KLAIM/KLAVA [5]) or in other programming languages allowing easier manipulation of system resources such as OCaml (JCL [12] and JOCAML [24]) or C (DITYCO [26]). Marshalling strategies range from dedicated byte-code structures (JCL, JOCAML, DITYCO) to Java serialization (SAM, X-KLAIM/KLAVA).

Thus, a generic communication framework to build prototype implementations of process calculi cannot restrain itself to a fixed set of interaction primitives or marshalling strategies. Instead, a middleware like IMC should be flexible enough to support multiple marshalling strategies and communication protocols. The framework should also aim at minimality to introduce new communication protocol support with little effort, in any case without need to re-implement a new communication library: either by realizing specialized implementations of the framework interfaces, or by defining framework increments which will complement the IMC core interfaces and libraries.

A number of minimal platforms for flexible communications have already been implemented [10, 11, 15, 18, 20, 28] where objects interact transparently through remote method invocations on well-defined interfaces. Their originality compared to CORBA-like or Java RMI-based infrastructures is to provide a core framework for building different types of middleware using the notion of flexible *bindings*. Creating a new binding should be understood as setting up access and communication paths between components of a distributed system with a wide variety of semantics: mobile, persistent, with QoS guarantees, etc. An adaptable communication framework should provide primitives to define bindings with various semantics, and to combine them in flexible ways. With simple architectural principles such as separating marshalling from protocol implementation, or threading from resource management, those middleware have shown how to dynamically introduce new protocols or control the level of resource multiplexing. In the IMC communication framework design, an important decision was to leverage the previously described naming and binding framework for network protocols in order to achieve adaptable forms of communication transparency needed when implementing a specific process calculus. The communication framework enables the definition of customized protocol stacks by a flexible composition of micro-protocols. In practice, the implementation of a new process calculus will most likely use TCP/IP as lowest layer of interface to the network. Thus, the IMC communication framework provides support for TCP/IP bindings, but can be easily extended to other protocols.

The IMC communication framework is composed of two main sub-packages:

– Sub-package `org.mikado.imc.protocols.apis` contains the interfaces describing the main abstractions for communication, e.g. sessions, protocols, marshallers, . . .
– Sub-package `org.mikado.imc.protocols.libs` contains the classes which implement those interfaces and offer support for flexible TCP/IP bindings.

In what follows, we describe the main abstractions and interfaces of the IMC communication framework. We then illustrate over a simple example – a small client/server authentication protocol called “knock-knock” – how protocol and session objects can be combined to implement new communication protocols taking advantage if IMC.

Design

The communication framework builds upon the IMC abstractions for naming and binding such as identifiers, references, and naming contexts. Protocol-specific abstractions are inspired from the *x*-kernel [17] and JONATHAN [20] communication frameworks and are represented by the interfaces below:

```
public interface Protocol {}

public interface ProtocolGraph {
  public SessionIdentifi er export(Session_Low session);
}

public interface SessionIdentifi er {
  public Protocol getProtocol();
  public Session_High bind(Session_Low session);
}

public interface Session_High {
  public void send(Marshaller message);
}

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

A *protocol* represents network protocols like TCP, IP, or GIOP, and provides a naming context for a particular kind of interfaces called *sessions*. It manages names called *session identifiers* to designate those interfaces.

The structure of a protocol stack is captured by a *protocol graph*. This directed acyclic graph composed of protocol nodes describes the path to be followed by messages when they are sent over the network, or received. A given session can be exported to inform the communication layers that it is willing to accept messages: a call on the `export` method at the root of a protocol graph will issue recursively the appropriate calls on each node of its sub-graphs. A session identifier is then returned to designate the exported session. To communicate with the exported session, a client just needs to call the `bind` method on the returned session identifier, which will provide a surrogate the client can use to send messages to the exported server session.

A *session* is an abstract representation of a communication channel. A session object is dynamically created by a protocol and lets messages be sent and received through the communication channel it stands for using that protocol. It has higher and lower interfaces to send messages down and up a protocol stack which may be viewed as a stack of sessions.

Exported session objects are designated using *session identifiers*. Their internal structure is protocol-specific. For instance, a TCP/IP session identifier encapsulates a host name and port number. Session identifiers are created when exporting a server-side session and then transmitted over the network. On the client side, they allow to establish communication channels by invoking the `bind` operation, with an optional session parameter to receive messages sent by the remote server-side session.

Sessions and Connections Messages can navigate through a protocol stack using the session interfaces (`Session_High` and `Session_Low`) with a single method to perform the message sending operation. A `Session_High` object is used to send messages down to the network. It will usually be a surrogate for a `Session_Low` type of session, which has been exported to a `Protocol` instance and is designated by a `SessionIdentifier` interface. A `Session_High` instance may be obtained by invoking the `bind` operation on a session identifier representing a `Session_Low` interface: it is thus a surrogate, or a proxy, for that interface. A `Session_Low` object is 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. The additional parameter in the `send` method represents the sender, and may be used to send a reply, if necessary.

Each session contains a lower-level abstraction of a communication channel called a *connection*. It typically encapsulates a regular socket, and provides operations to read and write to the socket. Client-side or server-side connections may be built on demand using *connection factories*, for instance on an incoming connection request from a client. A *connection manager* keeps track of idle and active connections, and delegates the creation of new connections to a connection factory.

To facilitate concurrent programming within sessions, the framework also offers basic primitives for activity management and their scheduling according to various criteria such as priorities, deadlines, etc.

Marshalling Marshallers and unmarshallers are used as high-level and encoding-independent representations of messages that are about to be sent or received. The `Marshaller` interface is described below:

```
public interface Marshaller {
  public void writeBoolean(boolean b);
  public void writeChar(char c);
  ...
  public void writeReference(Object obj);
  public void writeCode(MigratingCode code);
  public boolean isLittleEndian();
  public void close();
}
```

The `UnMarshaller` interface is similar but with read instead of write operations. The communication framework allows to customize the marshalling and unmarshalling of messages, by including interfaces for the management of *chunks*, or fragments of byte arrays which are chained together to form messages. The use of chunks helps avoiding unnecessary copying of memory blocks when messages move up and down a protocol stack. In particular, `writeCode` and `readCode` are used to implement code mobility and rely on the sub-package `mobility` described in Section 5.

Implementation The IMC communication framework provides TCP/IP-level binding management mechanisms: the main classes implement the TCP/IP protocol, standard marshallers, TCP/IP connection managers, as well as standard chunk managers, schedulers, and distributed naming contexts [22].

An Example

We now show how to use the IMC communication framework to implement new networking protocols. Consider the following simple protocol called "knock-knock", for authentication between a client C and a server S:

1	**Connect Request**	$C \rightarrow S$:	**Connect**
2	**Connect Reply**	$S \rightarrow C$:	**Knock-knock**
3	**Authentication Request**	$C \rightarrow S$:	**Who's there?**
4	**Authentication Reply**	$S \rightarrow C$:	*Challenge* e.g. `Will`
5	**Confirmation Request**	$C \rightarrow S$:	*Challenge* **who?** e.g. `Will` **who?**
6	**Confirmation Reply**	$S \rightarrow C$:	*Response* e.g. `Will you let me in?` `It's cold out here!`

This protocol can be easily implemented using the IMC communication framework over TCP/IP bindings by a set of session and protocol objects shown in Figure 1. The client first asks the user for the server host name and TCP port number. The TCP/IP stack is initialized by creating a new instance of the `TcpIpProtocol` class and a new client session identifier with the given parameters. The communication channel is established by a `bind` call on this identifier. The returned `Session_High` object can later be used to send messages over the network: the **Connect** message is first sent. The client then waits for replies from the server. The "knock-knock" client is given by:

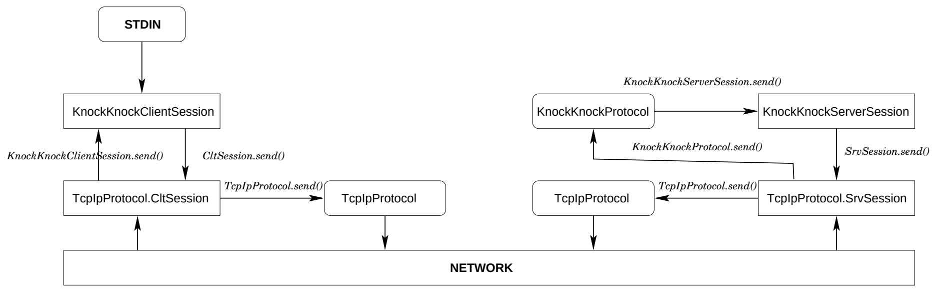

Fig. 1. Session and Protocol Objects in the "Knock-Knock" Protocol.

```
public class KnockKnockClient {
 public static void main(String[] args) {

   // Ask the user for the hostname and port number to connect to
   ...
   // Instantiate TCP/IP protocol stack
   TcpIpProtocol protocol = new TcpIpProtocol(...);
   IpSessionIdentifi er id = protocol.newSessionIdentifi er(hostname, port);

   // Bind to remote TCP/IP session
   Session_High session = id.bind(new KnockKnockClientSession(System.in));

   // Send "CONNECT" message
   Marshaller connectMsg = IMCMarshallerFactory.newMarshaller();
   connectMsg.writeString("Connect"); session.send(connectMsg);

   // Wait for replies from the server
   ...
 }
}
```

The top-level session object in the Knock-Knock/TCP/IP stack is a `KnockKnock-ClientSession` which directly reads data from standard input. When a message is received from the network, the "knock-knock" session `send` method is called. The message content is then displayed on the standard output device. Any message typed on the standard input device will be forwarded to the TCP/IP session which will send it over the network:

```
class KnockKnockClientSession implements Session_Low {
  ...
  public void send(UnMarshaller message, Session_High tcp_session) {

    // Read message from Knock−Knock server
    String fromServerMsg = message.readString();
    System.out.println("KKServer: " + fromServerMsg);

    // Get user input
```

```
      System.out.print("KKClient?"); fromUserMsg = System.in.readLine();

      // Send user input to Knock−Knock server
      Marshaller toServerMsg = IMCMarshallerFactory.newMarshaller();
      toServerMsg.writeString(fromUserMsg); tcp_session.send(toServerMsg);
    }
  }
```

The “knock-knock” server begins by creating a protocol graph composed of two nodes, instances of the `TcpIpProtocol` and `KnockKnockProtocol` classes. It then waits for client invocations:

```
  public class KnockKnockServer {
    public static void main(String[] args) {
      ...
      // Create Knock−Knock/TCP/IP protocol stack
      TcpIpProtocol protocol = new TcpIpProtocol(...);
      protocol.newProtocolGraph().export(new KnockKnockProtocol());

      // Wait for invocations
      ...
    }
  }
```

A “knock-knock” protocol object maintains a set of “knock-knock” sessions. Each session is associated with an underlying TCP/IP session in a hashtable, in order to record the path messages should follow in the protocol stack. When exporting the “knock-knock” protocol, a TCP/IP server-side session is created containing a new server socket to listen for connection requests from clients. When a client connects, the TCP/IP session reads all messages from the network and forwards them to the higher-level `KnockKnockProtocol` instance: the `send` method of that class is called, passing as a parameter the TCP/IP server session for sending back replies to the network. A new entry in the hashtable is created, associating the TCP/IP session with a new “knock-knock” session where message processing will be performed. Control is then transferred to the `send` method of the “knock-knock” session:

```
  class KnockKnockProtocol implements Session_Low {

    Hashtable kk_sessions; // A pool of session objects
    ...
    // Send back reply to the message coming from the network
    public void send(UnMarshaller message, Session_High tcp_session) {

      // Determine the TCP/IP session to use
      KnockKnockServerSession kk_session = null;
      synchronized (this) {
        kk_session = (KnockKnockServerSession) kk_sessions.get(tcp_session);
        if (kk_session == null) {
          kk_session = new KnockKnockServerSession(this);
          kk_sessions.put(tcp_session, kk_session);
        }
      }
      // Send the reply message on the network
      kk_session.send(message, tcp_session);
    }
  }
```

The “knock-knock” session simply determines the correct message to send back to the TCP/IP session, based on the message received from the client, and following the “knock-knock” protocol. The TCP/IP session then sends the message over the network:

```
class KnockKnockServerSession {
  ...
  int state = WAITING; // State of the protocol

  // Protocol messages : challenges and responses
  static String challenges[] = { "Will", ... };
  static String responses[] = { "Will you let me in, it's cold out here!", ... };
  int ChallengeNumber;

  // Perform ‘‘knock−knock’’ protocol message processing
  public void send(UnMarshaller message, Session_High tcp_session) {

    String fromClientMsg = message.readString();
    switch(state) {
      case WAITING:
        if (fromClientMsg.equals("Connect")) {
          toClientMsg = "Knock Knock!"; state = SENT_KNOCK_KNOCK;
        } else state = ERROR;
        break;
      case SENT_KNOCK_KNOCK:
        if (fromClientMsg.equals("Who's there?")) {
          challengerNumber = random(MAX_CHALLENGES);
          toClientMsg = challenges[challengeNumber]; state = SENT_CHALLENGE;
        } else state = ERROR;
        break;
      case SENT_CHALLENGE:
        if (fromClientMsg.equals(challenges[challengeNumber] + "who?")) {
          toClientMsg = responses[challengeNumber]; state = WAITING;
        } else state = ERROR;
        break;
      case ERROR:
        tcp_session.close(); state = WAITING;
        break;
    }

    // Send reply message on the network
    if (state == WAITING) {
      Marshaller reply = IMCMarshallerFactory.newMarshaller();
      reply.writeString(toClientMsg); tcp_session.send(reply);
    }
  }
}
```

5 Code Mobility Management

The purpose of this part of the framework is to provide the basic functionalities for code mobility. All these functionalities are implemented in the sub-package `org.mikado.imc.mobility`. This package defines the basic abstractions for code marshalling and unmarshalling and also implements the classes for handling Java byte-code mobility transparently.

Motivations

The base classes and the interfaces of this package abstract away from the low level details of the code that migrates. By redefining specific classes of the package, the framework can be adapted to deal with different code mobility frameworks. Nowadays, most of these frameworks are implemented in Java thanks to its great means and features that help in building mobile code systems. In many of these systems, the code that is actually exchanged among sites is Java byte-code itself. For this reason, the concrete classes of the framework deal with Java byte-code mobility, and provide functionalities that can be already used, without interventions, to build the code mobility part of a Java-based code mobility framework.

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: the classes needed by the moved process are collected and delivered together with the process;
- *on-demand* approach: 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 approach respects the main aim of this sub-package, i.e., it makes the code migration details 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* [29]: the agent owner does not have to stay connected after sending off an 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. 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; for this reason we also enable the programmer to choose whether this automatic code collection and dispatching should be enabled.

With the automatic approach, an object will 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 class libraries and the classes of the MIKADO framework) has to be common to every application. This guarantees that classes belonging to Java standard class libraries (and to the IMC package) 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.

Design

The package defines the empty interface `MigratingCode` that must be implemented by the classes representing a code that has to be exchanged among distributed site. This code is intended to be transmitted in a `MigratingPacket`, stored in the shape of a `byte` array:

```
public class MigratingPacket implements java.io.Serializable {
  public MigratingPacket(byte[] b) {...}
  public byte[] getObjectBytes() {...}
}
```

How a `MigratingCode` object is stored in and retrieved from a `MigratingPacket` is taken care of by the these two interfaces:

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

public interface MigratingCodeUnMarshaller {
  MigratingCode unmarshal(MigratingPacket p)
    throws InstantiationException, IllegalAccessException,
      ClassNotFoundException, IOException;
}
```

These marshaller objects are used also by the classes of the `protocols` package (see Section 4). In particular the `Marshaller` and `UnMarshaller` in the package `protocols` rely on instances of `MigratingCodeMarshaller` and `MigratingCode-UnMarshaller`, respectively, to deal with `MigratingPackages`.

Starting from these interfaces, the package `mobility` provides concrete classes that automatically deals with migration of Java objects together with their byte-code, and for transparently deserializing such objects by dynamically loading their transmitted byte-code. These classes are described in the following.

Java byte-code mobility All the nodes that are willing to accept code from remote sites must have a custom *class loader*: a `NodeClassLoader` supplied by this MIKADO sub-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 most important methods that concern a node willing to accept code from remote sites are `addClassBytes` to update the loader's class table, as said above, and `forceLoadClass` to bootstrap the class loader mechanism, as explained later:

```
public class NodeClassLoader extends java.lang.ClassLoader {
 public void addClassBytes(String className, byte[] classBytes) {...}
 public Class forceLoadClass(String className) {...}
}
```

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 define a base class for all objects/process that can migrate to a remote site, `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. Unfortunately, Java only provides single inheritance, thus providing a base class might restrict its usability. The problem arises when dealing with threads: the interface `Runnable` in the standard Java class library could solve the above issue but requires additional programming. For this reason we make `JavaMigratingCode` a subclass of `java.lang.Thread` (with an empty `run` method), so that `JavaMigratingCode` can be extended easily by classes that are meant to be threads. Thus, the most relevant methods for the programmer are the following ones:

```
public class JavaMigratingCode extends Thread implements MigratingCode {
 public void run() { /∗ empty ∗/ }
 public JavaMigratingPacket make_packet() throws IOException {...}
}
```

The programmer will redefine `run` if its class is intended to represent a thread. 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.

Once these class names are collected, their byte code is gathered in the first server from which the object was sent, 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). Notice that the migrating object (namely, its variables) is written in an array of bytes (inherited by `MigratingPacket`) and not in a field of type `JavaMigratingCode`. 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 object 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.

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 `NodeClassLoader` to load the classes stored in the `JavaMigratingPacket` and then on Java serialization to actually deserialize the migrating code contained in the packet.

Now let us examine the code that recovers the object from a `JavaMigratingPacket`, in the `JavaByteCodeUnMarshaller`. As previously hinted, a site that is willing to receive a remote object must use a `NodeClassLoader` that will take care of loading the classes received with a `JavaMigratingPacket`. The Java class loading strategy works as follows: whenever a class *A* is needed during the execution of a program, if it is not already loaded, then the class loader that loaded the class that needs *A*, say *B*, is required to load the class *A*. This usually takes place in the background, and the only class loader involved is the system class loader. In our case, we have to make our `NodeClassLoader` load the classes of the packet of the migrating object. For this reason, we have to make sure that the received object (contained in the `MigratingPacket`) is actually retrieved by a local object whose class is loaded by the `NodeClassLoader`. Since this class is a local class, i.e., a class present in the local class library, we have to force it to be loaded by the `NodeClassLoader` and not by the system class loader. In particular, the sub-package `mobility` provides an interface, `MigratingCodeRecover` and a class, `MigratingCodeRecoverImpl`, for recovering objects and classes from a `MigratingPacket`. The steps to perform are: load the `MigratingCodeRecoverImpl` class through the class loader (by forcing its loading so to avoid it is loaded by the system class loader) and recover the received packet through the `MigratingCodeRecoverImpl` instance:

```
NodeClassLoader classloader = class_loader_factory.createNodeClassLoader();
String recover_name =
 "org.mikado.imc.mobility.MigratingCodeRecoverImpl";
MigratingCodeRecover recover =
 (MigratingCodeRecover) (classloader.forceLoadClass(recover_name, true).newInstance());
```

Notice that `recover` is declared as `MigratingCodeRecover` but its actual class is `MigratingCodeRecoverImpl` (which is a class implementing the interface

MigratingCodeRecover). Indeed, the following code would generate a ClassCast-Exception:

```
MigratingCodeRecoverImpl recover =
 (MigratingCodeRecoverImpl) (classloader.forceLoadClass(recover_name, true).newInstance());
```

since Java considers two classes loaded with different class loader as incompatible. In the wrong code snippet above, for instance, the class MigratingCodeRecoverImpl of the variable recover would be loaded through the system class loader, and it would be assigned an object of the same class MigratingCodeRecoverImpl, but loaded with NodeClassLoader. This is the reason why we have to assign the instance loaded by NodeClassLoader to a variable declared with a superclass of the actually loaded class.

Once this MigratingCodeRecover object is loaded through our NodeClass-Loader, we can deserialize the received object with these two simple instructions:

```
recover.set_packet(pack);
MigratingCode code = recover.recover();
```

The method recover will return the object stored in the MigratingPacket and the classes needed by such object, stored in the packet, will be automatically loaded by the NodeClassLoader. We would like to point out that not all the classes of the received object are necessarily loaded immediately; however, each time such object needs a class to be loaded, this request will be handled transparently by the NodeClassLoader. We observe that once the object is recovered from a packet, it can be used to create another packet to be sent to another site.

By default, the JavaByteCodeUnMarshaller uses a brand new class loader (through an abstract factory) for each MigratingPacket. Thus, each migrating object will be incompatible with other migrating objects, since each one of them is loaded through a different classloader. This name space separation provides a sort of isolation that helps avoiding that migrating objects coming from different sites do not interfere with each other. If this is not the desired behavior, the JavaByteCodeUnMarshaller can be initialized with a specific NodeClassLoader instance that will always be used to load every migrating object. Alternatively, the user can provide the JavaByteCodeUnMarshaller with a customized abstract factory in order to force it to use a customized NodeClassLoader for each migrating object.

Examples

Let us now show a small tutorial on how to use this sub-package for Java byte-code migrating code. First of all the classes of objects we want to migrate must be subclasses of JavaMigratingCode:

```
public class MyCode extends JavaMigratingCode {
  MyVar v = new MyVar();

  public MyRetType getFoo(MyPar p) {...}
  ...
}
```

Now an object of this class (or of one of its possible subclasses) can be sent to a remote site by creating a `MigratingPacket`, through a `JavaByteCodeMarshaller` described above. Once such a packet is created, it can be directly written into an `ObjectOutputStream` that, in turn, is connected, for instance, to a network output stream:

```
public class Sender {
 ...
 void sendCode(OutputStream os) throws Exception {
  MigratingCodeMarshaller marshaller = new JavaByteCodeMarshaller();
  MigratingCode code = new MyCode();
  MigratingPacket pack = marshaller.marshal(code);
  ObjectOutputStream obj_os = new ObjectOutputStream(os);
  obj_os.writeObject(pack);
  obj_os.flush();
 }
}
```

Let us observe that the act of creating a `MigratingPacket` automatically collects all the classes that `MyCode` uses, apart from creating an array of bytes representing the state of the object to migrate. Thus, the classes `MyVar`, `MyRetType` and `MyPar` are stored in the packet as well.

The site that receives a migrating object will basically perform the complementary operations: read a `MigratingPacket` from a stream (e.g., from the network) and use a `JavaByteCodeUnMarshaller` to retrieve the object from the received packet (all the operations for loading the classes will be transparent to the programmer):

```
public class Receiver {
 ...
 JavaMigratingCode receiveCode(InputStream is) throws Exception {
  MigratingCodeUnMarshaller unmarshaller = new JavaByteCodeUnMarshaller();
  ObjectInputStream obj_is = new ObjectInputStream(ss);
  MigratingPacket pack = (MigratingPacket) obj_is.readObject();
  return (JavaMigratingCode) unmarshaller.unmarshal(pack);
 }
}
```

Notice that the object retrieved from the packet is of type `JavaMigratingCode`, thus only the methods defined in that class can be used (e.g., the method `start`, inherited by `Thread`). Moreover a cast to its actual class (that in this example is `MyCode`) is not possible because that class is unknown in the receiving site and, even if it was known such cast would make the system class loader try to load the class `MyCode`; either the system class loader fails to load the class or, however, the two instances would be incompatible as explained above.

This may seem a strong limitation, but the applications that exchange code can agree on a richer interface or base class for the migrating code, say `MyMigratingProcess`, with other methods, say `m` and `n`; such class must be present in all the sites where these applications are running so that it can be loaded by the system class loader. For this reason, the class `MyMigratingProcess` must not be inserted in the `MigratingPacket`.

The class `JavaMigratingCode` provides a method, `setExcludeClasses` that allows to specify which classes must not be inserted in the packet[1]. Thus, the code of the sender shown above should be changed as follows (it delivers a `MyProcess` object, where `MyProcess` inherits from the common base class `MyMigratingProcess` that in turns derives from `JavaMigratingCode`):

```
public class Sender {
 ...
 void sendCode(OutputStream os) throws Exception {
  JavaMigratingCode code = new MyProcess();
  code.setExcludeClasses("mypackage.MyMigratingProcess");
  MigratingPacket pack = code.make_packet();
  ObjectOutputStream obj_os = new ObjectOutputStream(os);
  obj_os.writeObject(pack);
  obj_os.flush();
 }
}
```

The receiving code can then assign the retrieved object to a `MyMigratingProcess` instance and then use the richer interface of `MyMigratingProcess`:

```
MyMigratingProcess code = (MyMigratingProcess) unmarshaller.unmarshal();
code.m();
code.n();
```

An alternative to `setExcludeClasses` is the method `addExcludePackage` that allows to exclude a whole package (or several packages) from the set of classes that are delivered together with a migrating object. For instance, the call to `setExcludeClasses` above could be replaced by the following statement:

```
 code.addExcludePackage("mypackage.");
```

This allows to enforce that the whole excluded package is available on all the sites where the migrating code is dispatched to.

When extending `JavaMigratingCode`, 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.

6 Implementing $D\pi$ with IMC

To evaluate applicability of the components provided by IMC a small framework, called $JD\pi$, has been developed. This framework provides the runtime environment for executing programs developed using a $D\pi$ paradigm. The implementation schema is the

[1] We remind that the mobility sub-package already excludes all the Java system classes and the classes of the IMC package itself.

same as the one adopted for developing KLAVA [5] and X-KLAIM [2, 4]: like KLAVA is the runtime for X-KLAIM so *JDπ* will be the runtime for *Dπ*. In the next future, a compiler will be developed to transform *Dπ* code into Java code that relies on *JDπ*.

Design

Dπ, introduced by Hennessy and Riely [16], is a locality-based extension of the π-calculus [27] that requires processes to be located at nodes. More precisely the top-level consists of a parallel composition of nodes with running processes. The language is also enriched with a `go` primitive that permits processes to migrate to different nodes.

Analyzing the *Dπ* paradigm, one can single out three main concepts: *Nodes*, *Processes* and *Channels*. A *Dπ program* consists of a set of nodes. Each node, which is identified by a locality, contains processes running in parallel. Processes interact with each other, locally, by means of asynchronous communication performed via channels. A process can change its execution environment (the node where it is running) by performing a `go` *l* action: the execution is suspended, the process migrates at the node named *l* and there it restarts its computation. We assume that each host in the network may contain more *Dπ* nodes that are executed within a common environment called *Site*.

The basic *Dπ* ingredients are implemented by using the following classes:

– `JdpiSite`, that implements a container for nodes running on a host
– `JdpiNode`, that implements *Dπ* nodes
– `JdpiAgent`, that implements *Dπ* processes

Classes and interfaces

The UML class diagram of *JDπ* is presented in Figure 2. In the rest of this section, we describe the classes in the diagram.

JdpiSite `JdpiSite`, which extends `org.mikado.imc.topology.MikadoNode` class, is implemented using the pattern *singleton*. This means that only one instance of `JdpiSite` can be created. A reference to this instance can be obtained using the method `getInstance()`. `JdpiSite` also provides the method `init()` that is invoked to initialize the site. Finally a new node can be created using the method `getNewNode()`. `JdpiSite` is also responsible for providing naming services for running nodes. This is obtained at no cost, since all naming capabilities are inherited directly from `MikadoNode`.

JdpiNode `JdpiNode`, implements a *Dπ* node, and each instance of it runs under the control of a `JdpiSite`. Since processes, in order to move across the network, need to refer remotely to nodes, `JdpiNode` implements `Referenceable`. Using this approach, a process does not refer directly to a node, but it uses a `JdpiLocality`. In the present implementation `JdpiLocality` is just an abstraction for a host name and

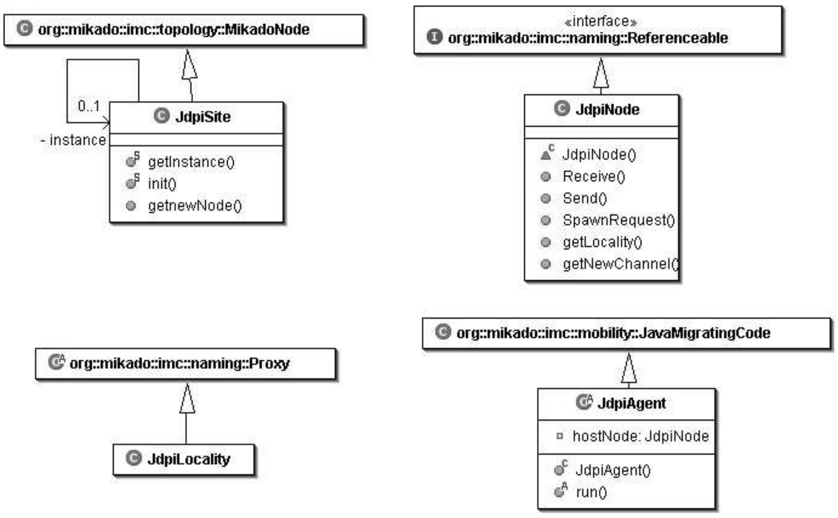

Fig. 2. Jdpi class diagrams

port. This class will extend `org.mikado.imc.naming.Proxy` (the class used to refer to remote `Referenceable` objects) in next development step. The class provides the method `addAgent` to start a new *Dπ* process on it. Note that the only mechanism for interacting with a process running at a remote node is to spawn another process to be evaluated remotely. This is due using the method `SpawnRequest()`, that spawn a process to be evaluated remotely. `JdpiNode` is also responsible for the channel management. Methods `getNewChannel()`, `Send()` and `Receive()` are used, respectively to create a new channel, to send an object over a channel and to receive an object from a channel.

JdpiAgent `JdpiAgent` represents a (mobile) *Dπ* process. For this reason `JdpiAgent` extends `JavaMigratingCode`, which is defined in `org.mikado.imc.mobility`. The infrastructure defined in the IMC framework allow the instances of `JdpiAgent` to migrate from a `JdpiNode` to another. An instance of the `JdpiAgent` class does nothing by itself, acting like the nil process. Programmers need to extend it in order to implement other *Dπ* processes. The process behavior is thus defined overriding the method `execute`. A `JdpiAgent` provides a private attribute, `hostNode`, that represents the hosting node. This can be used to invoke the operations over local channels, as described earlier. This attribute is set when a JdpiAgent constructor is invoked, and then only the Runtime should modify it. A `JdpiAgent` can also send itself to a remote node by invoking the `SpawnRequest` method on the hosting node. Being an extension of `JavaMigratingCode`, `JdpiAgent` extends also the `Thread` class. So a JdpiAgent is executed like a Java Thread, by calling the method `start`. Note that you can't override the `run` method of the `Thread` class. To assure that no `JdpiAgent` should run with a null hostNode, the `run` method has been declared as `final` and when it is called, if the hostNode is defined, it calls the `execute` method, otherwise it returns.

7 Conclusions

We have presented a Java package IMC that aims at providing a framework for fast prototyping distributed applications with code mobility. It aims at providing support to those building run-time systems (or virtual machines) for mobile code languages and calculi. We chose Java as the implementation platform due to its well established role in the development of this kind of software. Indeed, Java provides many useful features that are helpful in building network applications and in dynamically loading code from different sources (e.g., the network itself). However, these mechanisms still require a big programming effort, and, in this respect, they can be thought of as "low-level" mechanisms. Because of this, many existing Java based distributed systems (see, e.g., [1, 7, 8, 23, 30, 31] and the references therein) tend to implement from scratch many components that are typical and recurrent in distributed and mobile applications.

For this reason, we decided to single out the most recurrent entities of this type of applications and pack them together in a Java framework, where the architecture of distributed and mobile applications is addressed by the framework itself. The programmer can then concentrate on those parts that are really specific of his system, while relying on the framework for the recurrent standard mechanisms (node topology, communication and mobility of code). This should make the development of prototype implementations faster and should relieve the programmers from dealing with low level details. Of course, if specific 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), the programmer can still customize the behaviors that concern these mechanisms in the framework.

We experimented on this matter in two respects:

- In the prototype implementation of $JD\pi$ (Section 6) we used the IMC package as it is without resorting to any customization;
- We re-engineered the implementations of our mobile code systems, TYCO and KLAVA, using the IMC package. At this stage, we had to modify/extend only specific parts of the framework (e.g., the mobility code management for TYCO and the communication protocol for KLAVA).

In both cases, we managed to concentrate our programming efforts on the main features and mechanisms of the specific distributed mobile system, and, for the rest, we relied completely on the architecture and the functionalities of the IMC framework.

Apart from the above, the Communication Protocols package was used to define customized protocol stacks by composing micro-protocols in a flexible manner. In particular, this experiment showed how new protocols can be introduced with IMC, by making evident the protocol and session objects involved, and by describing the path followed by messages within a protocol stack [21].

For the rest of the project we shall, on the one hand use the framework to implement richer languages for mobility and on the other hand we shall enrich the components to deal with security issues.

Acknowledgements. We are greatly indebted to Michele Loreti for discussions on the architecture of the framework and, especially, for suggestions about the implementations of *JD*π.

References

1. A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In Vitek and Tschudin [35], pages 111–130.
2. L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`.
3. 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.
4. L. Bettini, R. De Nicola, and R. Pugliese. X-KLAIM and KLAVA: Programming Mobile Code. In M. Lenisa and M. Miculan, editors, *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
5. L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.
6. 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.
7. 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-Verlag, 1998.
8. P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In Vitek and Tschudin [35], pages 213–228.
9. G. Cugola, C. Ghezzi, G. Picco, and G. Vigna. Analyzing Mobile Code Languages. In Vitek and Tschudin [35].
10. B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stefani. Jonathan: an Open Distributed Processing Environment in Java. In *Proceedings MIDDLEWARE'98*, 1998.
11. ExoLab Group. The OpenORB project, 2002. Software available for download at `http://openorb.exolab.org/`.
12. C. Fournet and L. Maranget. The Join-Calculus Language, 1997. Software and documentation available from `http://join.inria.fr/`.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
14. C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research Report 19887, IBM Research Division, 1994.
15. R. Hayton, A. Herbert, and D. Donaldson. Flexinet: a Flexible Component Oriented Middleware System. In *Proceedings ACM SIGOPS European Workshop*, 1998.
16. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3, pages 3–17. Elsevier Science Publishers, 1998.
17. N. Huntchinson and L. Peterson. The *x*-kernel: an Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.
18. 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.
19. F. Knabe. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS workshop on Analysis and Verifi cation of Multiple - Agent Languages*, number 1192 in LNCS. Springer-Verlag, 1996.

20. 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`.
21. M. Lacoste. Building Reliable Distributed Infrastructures Revisited: a Case Study. In *International DOA Workshop on Foundations of Middleware Technologies (WFoMT'02)*, 2002.
22. M. Lacoste. IMC: Flexible Communication Support for Implementing Mobile Process Calculi. Technical report, France Telecom R&D, 2003.
23. D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.
24. F. Le Fessant. The JoCaml System Prototype, 1998. Software and documentation available from `http://join.inria.fr/jocaml`.
25. F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 352–364. ACM Press, 2000.
26. L. Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, University of Porto, 1999.
27. R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.
28. 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.
29. A. Park and P. Reichl. Personal Disconnected Operations with Mobile Agents. In *Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*, 1998.
30. 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)*, LNCS, pages 50–61. Springer-Verlag, 1997.
31. G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, 1999.
32. D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proc. 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*, volume 2076 of *LNCS*, pages 408–420. Springer-Verlag, 2001.
33. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997.
34. V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.
35. J. Vitek and C. Tschudin, editors. *Mobile Object Systems - Towards the Programmable Internet*, number 1222 in LNCS. Springer, 1997.
36. J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.