Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 181 (2007) 63–79
www.elsevier.com/locate/entcs

Implementing a Distributed Mobile Calculus Using the IMC Framework[1]

Lorenzo Bettini, Rocco De Nicola
Daniele Falassi and Michele Loreti[2]

Dipartimento di Sistemi e Informatica
Università di Firenze
Firenze, Italy

Abstract

In the last decade, many calculi for modelling distributed mobile code have been proposed. To assess their merits and encourage use, implementations of the calculi have often been proposed. These implementations usually consist of a limited part dealing with mechanisms that are specific of the proposed calculus and of a significantly larger part handling recurrent mechanisms that are common to many calculi. Nevertheless, also the "classic" parts are often re-implemented from scratch. In this paper we show how to implement a well established representative of the family of mobile calculi, the distributed π-calculus, by using a Java middleware (called IMC - *Implementing Mobile Calculi*) where recurrent mechanisms of distributed and mobile systems are already implemented. By means of the case study, we illustrate a methodology to accelerate the development of prototype implementations while concentrating only on the features that are specific of the calculus under consideration and relying on the common framework for all the recurrent mechanisms like network connections, code mobility, name handling, etc.

Keywords: Code Mobility, Language Implementation, Network Programming

1 Introduction

Prompted by the impressive development of networks technologies, in the last decade there has been a high number of proposals of calculi for modelling and reasoning about distributed systems that also encompass mobility of code and processes. These formalisms, in general, provide constructs and mechanisms at different abstraction levels, for modelling the execution contexts of the network where applications roam and run, for coordinating and monitoring the use of resources, for expressing process communication and mobility, and for specifying and enforcing security policies.

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

[2] Email:{bettini,denicola,falassi,loreti}@dsi.unifi.it

doi:10.1016/j.entcs.2007.01.054

The ancestor of these calculi is the π-calculus [15], a very simple formalism relying on a small number of combinators, that nevertheless can be used to model non trivial systems. The π-calculus, however, does not have explicit primitives for modelling distribution and has a somehow too basic communication mechanism. There have thus been direct generalizations of π-calculus such as the *Distributed Join Calculus* [8], the *Seal Calculus* [19], *Nomadic Pict* [18], Dπ-calculus [12], *lsd*π-calculus [16]. But also calculi based on other approaches such as the *Ambient calculus* [7] or the *Linda* model [11]. Among the calculi based on Ambient we mention *Safe Ambients* [14] and *Boxed Ambients* [6]. Among the calculus based on Linda we have many variants of *Klaim* [2].

Very often to assess the quality of the proposed new calculi the different research groups have also produced prototype implementations of the calculi that could be used as kernel programming languages for mobile distributed systems. Prompted by the growing number of experiments and from the need of easing the implementation phase, we have implemented a generic Java framework called IMC (*Implementing Mobile Calculi*) that can be used as a kind of middleware for the implementation of different distributed calculi [3]. IMC aims at providing the necessary tools for implementing the run-time system of new languages directly derived from calculi for mobility. Our aim has been that of enabling the implementer of a new language to concentrate on the parts that are really specific of his/her system, and to rely on our framework for the recurrent standard mechanisms for distribution and mobility thus avoiding to deal with low-level details. Java has been chosen as the production language because it provides many useful features for building network applications with mobile code (indeed, many existing implementations of mobile and distributed systems are written in Java).

IMC provides means for transparent code mobility, for building communication protocols by composing sub-components dynamically and for managing node topology. All these mechanisms are rendered as abstract as possible to ease, e.g., switching from a specific communication protocol to another, without modifying the other parts of an application. IMC can be straightforwardly used if no specific advanced feature is needed. A user can however customize parts of the framework by providing its own implementations for the interfaces used in the package. Customizations can take advantage of design patterns such as *factory method*, *abstract factory*, *template method* and *strategy* [10] that are used throughout the packages.

The framework was designed to achieve both *transparency* and *adaptability*. For instance, for 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. To the best of our knowledge there are no others similar general frameworks available in the literature.

In this paper we shall describe a practical use of IMC by showing how it can

be used to implement a well established representative of the family of mobile calculi, the distributed π-calculus (Dπ). We shall first use IMC to build the runtime support for Dπ then we will use it for implementing `JDpi`, the generalization of the calculus to a simple programming language. Building the run time support will require first analyzing the distribution model, the communication mechanisms and the mobility aspects of the calculus to determine the part of IMC to be used, and then writing the appropriate code to interact with IMC and to implement specific parts. The main intent of the paper is to illustrate, by means of the case study, a possible methodology to accelerate the development of prototype implementations while concentrating only on the features that are specific of the calculus under consideration and relying on the common framework for all the recurrent mechanisms like network connections, code mobility, name handling, etc.. Please notice that other $D\pi$ constructs, like parallel composition or fresh name generation, can be implemented using standard `Java` primitives like, for instance, multi-threading.

The rest of the paper is organized as follows. Section 2 provides a brief overview of IMC, Section 3 presents Dπ while in Section 4 the actual implementation of `JDpi` is presented. The final section contains an example of `JDpi` program and some concluding remarks.

2 The IMC framework

We now sketch the main functionalities and interfaces of the framework. For the sake of simplicity, we will not get into deep details. IMC consists of three main subpackages: `protocols`, `mobility` and `topology` that deal with communication protocols, code mobility and network topology, respectively. We present the IMC components in the order we suggest to use them when implementing a run-time system for a mobile calculus. The first thing the developer should think of is the implementation of the communication protocol; then he/she can implement the node functionalities by using the communication protocols. Finally, he can implement the functionalities of processes that will rely on the features provided by the implementation of nodes. Of course, this is not a mandatory schema, but we found this path very useful when using IMC (see also Section 4).

2.1 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 to protocols closer to hardware resources. A generic communication framework should permit introducing support for new protocols with little effort, without need to re-implement a new communication library. Thus, IMC provides tools to define customized protocol stacks, which are viewed as a flexible composition of micro-protocols, and enables to achieve adaptable forms of communication transparency, which are needed when implementing an infrastructure for global computing.

In IMC, a *network protocol* 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`. The `Protocol` class aggregates the protocol states and, following the design patter "*template method*" [10], provides the method `start` that will execute each state at a time, starting from the first protocol state up to the final one (each state defines the next state to enter). Thus, the programmer must simply provide the implementation of each state, put them in a protocol instance, and then start the protocol.

The protocol states abstract away from the specific communication layers. 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: `Marshaler` (for writing) and `UnMarshaler` (for reading). These streams provide high-level and encoding-independent representations of information to be sent or received. They are basically an extension of standard `DataOutput` and `DataInput` Java streams, with the addition of means to send and receive mobile code (explained later) and serialize and deserialize objects.

The data in these streams can be "pre-processed" by some customized *protocol layers* that remove some information from the input and add some information to the output: typically this information is protocol-specific headers removed from the input and added to the output. 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. Each layer is independent and the composition of layers in a protocol stack takes place at run-time. 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*, e.g., to implement a layer to tunnel an existing protocol into HTTP. Figure 1 shows a protocol made up of 4 states that is using a protocol stack made up of 3 layers, the lowest one being the actual TCP socket layer, and the middle one being an HTTP tunneling layer.

To read something from a stack, a protocol state must obtain an `UnMarshaler` instance from the stack by calling the method `createUnMarshaler`: this allows the stack layers to retrieve their own headers. When the state finished to read, it must release the `Marshaler` by calling `releaseUnMarshaler`. In the same way, to write information into a stack, the state must obtain a `Marshaler` instance from the stack by calling the method `createMarshaler`, so that the stack layers can add their own headers into the output. When the state finished to write, it must notify the stack by calling the method `releaseMarshaler`, in order to flush the output buffer. Typically, these stream 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. Here are

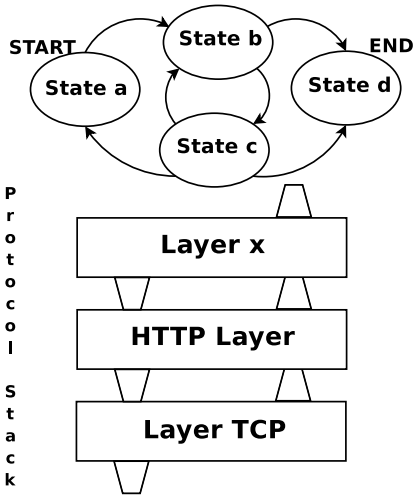

Fig. 1. The interaction between protocol states and protocol stack.

the typical instruction sequences for writing (left) and reading (right) by using the mechanisms we described so far:

```
Marshaler m =
  protocolStack.createMarshaler();
m.setMigratingCodeFactory(
  new JavaByteCodeMigratingCodeFactory());
m.writeStringLine("obj");
m.writeInt(obj.size());
m.writeMigratingCode(obj);
protocolStack.releaseMarshaler(m);
```

```
UnMarshaler u =
  protocolStack.createUnMarshaler();
u.setMigratingCodeFactory(
  new JavaByteCodeMigratingCodeFactory());
s = u.readStringLine();
i = u.readInt();
obj = u.readMigratingCode();
protocolStack.releaseUnMarshaler(u);
```

In particular, `writeMigratingCode` and `readMigratingCode` deal with code mobility: this is transparently handled by the subpackage `mobility` that will take care of serializing an object together with its byte-code, upon writing, and of dynamically loading received byte-code, upon reading (we refer the reader to [1] for the details of this subpackage). By providing a specialized `MigratingCodeFactory`, the programmer can customize the code migration mechanism.

2.2 Nodes and Processes

A participant in a network is an instance of the class `Node` of the package `topology`. A node is also a container of running processes that can 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. A process is an instance of 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`. 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`, the abstract method in `NodeProcess` that must be implemented by the programmer. Actually, a process can interact with

the node it is running on only through a `NodeProxy`, which ensures security by restricting the node interface visibility to a subset. If the class `Node` is extended by a derived class with new functionalities that we want to make available to the processes, we will also have to extend `NodeProxy` (Section 4.2 uses this technique).

The framework already provides some implemented protocols to deal with *sessions*. The concept of session 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). A `SessionManager` instance will keep track of all the sessions. 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 different kind of process, called *node coordinator*, is allowed to execute privileged actions, such as establishing a session, accepting connections from other nodes, closing a session, 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`.

A `Session` instance is identified by two `SessionId` objects, one indicating the local end and the other one indicating the remote end. A `SessionId` contains information about the "location" or "address" of a node; this concept depends on the specific communication medium: for instance, for an IP communication it will be a string of the shape IP:port. Moreover, it contains information about the specific low level communication protocol. For instance, `"udp-myhost.com:9999"` represents a UDP communication with the host `"myhost.com"` on port 9999. Upon establishing a session, the `SessionId` is used to determine the low level communication layer. Thus, switching from a communication layer to another is only a matter of changing the `SessionId`, while all the other classes in IMC are independent from this, and do not need to be changed. A `Session` can be established by using the method `connect`,, of class `Node`, specifying the `SessionId` of the remote end; a session request can be accepted by using the method `accept`, by specifying the local `SessionId`. These methods return a `ProtocolStack` object (where the lowest layer is already set as explained above); this can then be customized by adding specific `ProtocolLayer` objects. Finally it can be passed to a `Protocol` instance that will run upon it. The following is a code snippet executed by a `NodeCoordinator` that accepts a session request using a specific `SessionId`, adds an HTTP tunnel layer, and starts a specific communication protocol (a `NodeCoordinator` on a node that establishes the session will perform similar actions, but will use `connect`):

```
ProtocolStack s = accept(id);
s.insertLayer(new HTTPTunnelLayer());
Protocol p = new MyProtocol(s);
p.start();
```

Finally, inter-objects communication takes place via the *event* based functionali-

Systems

M, N	::=	$\mathbf{0}$	Empty
	|	$M \mid N$	Composition
	|	$(\nu e).N$	Restriction
	|	$l[\![P]\!]$	Agent

Threads

P, Q, R	::=	`stop`	Termination
	|	$P \mid Q$	Composition
	|	$(\nu e)P$	Restriction
	|	`go` $u.P$	Movement
	|	$u!\langle v\rangle.P$	Output
	|	$u?(X).P$	Input
	|	`rec` $A.P$	Recursion
	|	A	Process Identifier
	|	`if` $u = v$ `then` P `else` Q	Matching

Names

e, f	::=	$h, k, l, \ldots$	Locations
	|	$a, b, c \ldots$	Resource

Variables

X, Y	::=	x	Variable
	|	$X@z$	Located Pattern

Values

u, v, w	::=	`bv`	Basic Value
	|	e	Name
	|	x	Variable
	|	$u@w$	Located Value

Table 1
$D\pi$ syntax

ties provided by IMC. In particular, most classes of the framework are endowed with event generation capabilities (e.g., `ProtocolState`, `ProtocolLayer`, `Node`, etc.). This allows to keep the classes loosely coupled and communications among objects in the framework highly flexible. It is then easy to intercept, e.g., new connection requests or connection failures. In the implementation of `JDpi`, events are used to deal with commands received by remote sites (see Section 4.1).

3 $D\pi$ a Language for Distributed Processes

$D\pi$, introduced by Hennessy and Riely [12], is a locality-based extension of the π-calculus [15] where processes are distributed over a set of *nodes* (or *locations*) each of which is univocally identified by a *name* (or *location*).

Like in the π calculus, processes interact via message passing over *channels*. However, only local communication is permitted: Two processes can interact only if they are located at the same node. A process can change its execution environment performing action `go`. $D\pi$ does not assume a specific network topology. Indeed, this aspect is subspecified in the standard calculus.

$D\pi$ syntax is defined in Table 1. There, and in the rest of this paper, $a, b, c \ldots$ are used as channel names, and $h, k, l, \ldots$ as location names; while $e, f, \ldots$ are used when the distinction does not play any role.

The main syntactic category is that of *systems* (N, M, ...). Intuitively a $D\pi$ system consists of a set of agents, $l[\![P]\!]$, running independently in parallel, where l is the location where thread P is running. $\mathbf{0}$ is used to describe the empty system, i.e. the system where no agent is running. Systems are composed using parallel composition ($N|M$), and can share private names ($\nu e.N$).

Threads (P, Q, ...) are essentially π-calculus processes that can additionally create new locations or names ($(\nu e)P$) and migrate to other locations (`go` $l.P$). The conditional (**if**) corresponds to the matching and mismatching operators of the

(Comm) $l[\![a!\langle v\rangle.P]\!]|l[\![a?(X).Q]\!] \mapsto l[\![P]\!]|l[\![Q[v/X]]\!]$

(Go) $l[\![\texttt{go}\ k.P]\!] \mapsto k[\![P]\!]$

(Par) $$\frac{N \mapsto N'}{M|\ N\ \mapsto M|\ N'}$$

(Eq_1) $l[\![\texttt{if}\ (u=u)\ \texttt{then}\ P\ \texttt{else}\ Q]\!] \mapsto l[\![P]\!]$

(Eq_2) $$\frac{u \neq v}{l[\![\texttt{if}\ (u=v)\ \texttt{then}\ P\ \texttt{else}\ Q]\!] \mapsto l[\![Q]\!]}$$

(Restr) $$\frac{N \mapsto N'}{(\nu e)N \mapsto (\nu e)N'}$$

(Cong) $$\frac{N \equiv M \quad M \mapsto M' \quad M' \equiv N'}{N \mapsto N'}$$

Table 2
Reduction Relation for $D\pi$

$$l[\![\texttt{stop}]\!] \equiv \mathbf{0} \qquad l[\![P|Q]\!] \equiv l[\![P]\!]|l[\![Q]\!]$$
$$l[\![\texttt{rec}A.P]\!] \equiv l[\![P[\texttt{rec}A.P/A]]\!] \qquad l[\![(\nu e)P]\!] \equiv (\nu e)l[\![P]\!]$$
$$M|(\nu e)N \equiv (\nu e)(M|N) \text{ if } e \notin fn(M)$$

Table 3
Structural Congruence for $D\pi$

π-calculus. The definitions of free and bound names are similar to those for the π-calculus. By convention, $\texttt{go}\ l.P|Q$ will stand for $(\texttt{go}\ l.P)|Q$. Like in π-calculus, $D\pi$ threads interact with each other via name passing over channels. However, differently from π-calculus, names can be *located*. Indeed, $a@l$ can be used to relate channel a to locality l.

The semantics for $D\pi$ is defined by reduction relation $\mapsto$ defined in Table 2, where the structural congruence induced by rules of Table 3 is used. Agent migration is performed using rule (Go) that allows an agent located at l to migrate to location k. This transition can occur only when k is a known locality. Rule (Comm) permits co-located processes to interact via a channel a.

The distribution model of $D\pi$ is an extremely simple flat locality structure. If a node wants to spawn an agent remotely, it needs only to know the location of the remote node. This means that the knowledge of network topology structure is completely hidden to programmers (of $D\pi$ processes). In [9] a variant of $D\pi$ in which individual nodes may fail, or the links among them may be created and broken. The original language, $D\pi$, is extended with a new construct that permits processes to detect and react to these failures. In DpiF an explicit notion of *link* is introduced. Indeed, a network is *evaluated* considering a given topology that stores information about the state of nodes and the connections between them.

4 From $D\pi$ to JDpi

In this section we present the implementation of $D\pi$, JDpi. The implementation of a calculus typically consists of a run-time system (a sort of abstract machine) implemented in a language such as Java, and of a compiler that, given a program written in the programming language based on the calculus, produces code (Java in our case) that uses the run-time system above. In this paper we concentrate on the implementation of the run-time system in Java using IMC. Even without using the compiler, JDpi allows the programmer to write distributed and mobile

Java applications based on the $D\pi$ paradigm.

A $D\pi$ system is implemented as a set of distributed IMC nodes, each of which implements a $D\pi$ location, that communicate using a specific protocol (that permits sending a process to a remote node, and that deals with connections among nodes). In the rest of this section we first introduce the protocol used by remote participants to interact. Then we show how $D\pi$ location can be implemented using the IMC class `Node`. Finally, we present an implementation for $D\pi$ threads.

4.1 Protocol

In this section we describe the protocol used by the participants of a `JDpi` network. First, messages exchanged by these participants are presented, hence the `Protocol` used by each network component is introduced. The implementation of this part of the framework is considerably simplified by the use of both `mobility` and `protocol` packages of IMC.

In $D\pi$ two nodes can interact only when a process is spawned from one to the other. However, in `JDpi`, other kinds of interactions between two nodes can occur. For instance, two nodes interact when the topology of the network changes (a new node gets connected or an existing one disconnects). All interactions are modelled as a command that is sent from one node to another in order to achieve a specific task. We thus introduce the abstract class `JDpiCommand` that models a generic request. This class will be specialized to implement concrete commands.

```
public abstract class JDpiCommand extends JavaMigratingCode {
  public void setCommandId(String id) { ... }
  public String getCommandId() { ... }
  public abstract JDpiReply execute(JDpiNodeProxy proxy);
}
```

Each `JDpiCommand` is identified by a string, which is set automatically by the framework using method `setCommandId` when a command is sent remotely. This identifier is used for communicating the result of command execution. A concrete command has to implement method `execute`. This method is invoked when the command is received remotely in order to achieve the command task. For instance, in the case of a request for executing a thread remotely, the method `execute` will add the spawned process to the current node. As explained in Section 2, the parameter `proxy` permits using the *target* node without interacting directly with it. Since a command can contain a process to execute at a remote site, where its code is likely not to be available, it is crucial to use code mobility features; for this reason `JDpiCommand` extends the IMC base class `JavaMigratingCode`. Then, a command, together with its code, can be easily transmitted over the network as follows:

```
Marshaler m = protocolStack.createMarshaler();
m.setMigratingCodeFactory(new JavaByteCodeMigratingCodeFactory());
m.writeStringLine(command.getProtocolString());
m.writeMigratingCode(command);
protocolStack.releaseMarshaler(m);
```

The method `execute` returns a `JDpiReply`. This object is sent to the remote participant that originated the command for communicating the result of the exe-

cution.

```
public class JDpiReply implements Serializable {
  public boolean getResult() { ... }
  public String getDetails() { ... }
}
```

The method `getResult()` returns `true` if the command has been executed without errors, `false` otherwise (in this case the method `getDetails()` permits obtaining a string representation of the occurred error).

The `Protocol` used by participants of a `JDpi` system contains three states: `JDpiProtocolState`, `CommandState` and `NotifyState`. `JDpiProtocolState` is the main state. An object of this class selects the next state of the protocol by considering the value read from the underlying protocol stack. Accepted values are `"COMMAND"`, `"NOTIFY"` and `"STOP"`. When the string `"COMMAND"` is received a `CommandState` is activated. This is an inner class of `JDpiProtocolState` that when executed registers the command read from the protocol stack as a new event in the system. This permits notifying all the registered components, that will execute the appropriate handler to manage the command. After that, the protocol goes back to the main state. The string `"NOTIFY"` precedes `JDpiReply` that contains the execution result of a previous sent command. When such a string is received, a `NotifyState` is activated. This state, using the identifier read from the protocol stack, communicates the result of command execution to the sender of the command. Finally, when a string `"STOP"` is received, the protocol terminates.

This can be easily achieved by using the protocol state compositional features of IMC: in this case we use the class `ProtocolStateSwitch` that reads a request string from the input and selects the next state corresponding to the read string (it also deals with errors due to unrecognized requests). The user of this class has to associate a received string with the corresponding state by using `addRequestState`. Without entering into deep details, it should be straightforward to understand how the above described `JDpi` protocol is implemented by the following code:

```
public class JDpiProtocolState extends ProtocolSwitchState {
   public JDpiProtocolState(WaitingForNotification waiting, EventManager em) {
      addRequestState("COMMAND", new CommandState(em));
      addRequestState("NOTIFY", new NotifyState(waiting));
      addRequestState("STOP", Protocol.END);
      setNextState(Protocol.START);
   }
   ...
}
```

4.2 Nodes

A $D\pi$ node provides an abstraction for a computational environment that hosts execution of $D\pi$ threads and provides basic functionalities for thread interactions via channel communication. The IMC framework provides the `topology` package, and in particular the class `Node`. An object of this class implements a generic participant in the network and acts as a container of running processes. Moreover, IMC provides all the means for a process to access the resources contained in a

node (via a proxy) and to migrate to other nodes.

A general implementation of a $D\pi$ node is provided by `JDpiAbstractNode`. This class extends `Node` by providing new primitives for process interactions and for threads migration. To refer to a `JDpiAbstractNode`, class `JDpiLocality` is introduced. Since this class extends class `SessionId`, a `JDpiLocality` object can be used to retrieve information about the "location" or "address" of a remote node.

Class `JDpiAbstractNode` implements primitives for channel communication:

```
public <T> void out(JDpiChannelName<T> c, T v)
public <T> T in(JDpiChannelName<T> c)
```

Channels are referenced using class `JDpiChannelName<T>` that enables the identification of a channel used to exchange values of type `T`. The method `out` permits sending an object of type `T` over the channel referenced by `c`. Conversely, method `in` permits retrieving an object of type `T` from the channel referenced by `c`.

`JDpiAbstractNode` also provides support for process mobility. A process can be spawned to be evaluated remotely using the following method:

```
public JDpiReply go( JDpiProcess p, JDpiLocality l ) throws IMCException {
  if (self.sameId( l )) {
    this.addNodeProcess( p );
    return new JDpiReply(true);
  }
  JDpiCommand command = new JDpiEvalCommand(p, l);
  return sendCommand(command, l);
}
```

This method permits spawning a `JDpiProcess` (see below) to be evaluated remotely at the node referenced by the `JDpiLocality l`. If `l` refers to the current location (referenced by field `self`), the process is added to the current node. Otherwise, `p` has to be evaluated remotely; in this case, the process is first encapsulated within a `JDpiEvalCommand`. This class, which extends `JDpiCommand`, represents a request of executing a given process.

`JDpiAbstractNode` provides two methods that permits sending and executing a `JDpiCommand`:

```
protected JDpiReply sendCommand(JDpiCommand command, JDpiLocation l)
throws ProtocolException {
  ProtocolStack protocolStack = getNodeStack( l );
  JDpiSender sender;
  sender = new JDpiSender( getNewId(), command, protocolStack );
  sender.send();
  return sender.getReply();
}

public JDpiReply executeCommand(JDpiCommand c) {
  return c.execute( createNodeProcessProxy() );
}
```

The method `executeCommand` simply invokes method `execute` on the command instance `c` with the proxy for the current node. A command is sent remotely using a `JDpiSender`. This is an object that sends a command over a given protocol stack (`sender.send()`) and then waits for the result (`sender.getReply()`).

`JDpiAbstractNode` overrides method `createNodeProxy()` of the IMC class `No-`

de. Indeed, a JDpiNodeProxy (which extends NodeProcessProxy) is returned to allow processes to use the additional functionalities provided by JDpiAbstractNode, with respect to Node.

Since no specific network topology is considered, JDpiAbstractNode is an abstract class that provides three abstract methods:

```
public ProtocolStack getNodeStack( JDpiLocality l )
public JDpiReply forwardCommand(JDpiCommand command, JDpiLocality l )
public void start()
```

Method getNodeStack permits retrieving the ProtocolStack (see Section 2) to interact with the node identified by l. Please notice that the node referenced by l might not be directly connected to the local one. In this case, the actual implementation of JDpiAbstractNode has to choose (if it exists) a remote participant to use for communicating with l. Hence, every node could play also the role of intermediary in a communication. To define how a node behaves when a message (a JDpiCommand) for another node is received, abstract method forwardCommand has to be implemented. Finally, by implementing method start the programmer provides the initialization procedure for the node.

Now we describe how JDpiAbstractNode is used to implement two different kinds of network topologies: a flat topology, which implements standard $D\pi$ topology, and a DpiF-like topology.

Flat topology

Since $D\pi$ does not consider a specific implementation for nodes topology, a system can be implemented as a single server that accepts connections from nodes. This approach implements a flat topology and relies on the use of two kinds of nodes: JDpiDomain and JDpiNode. Both these classes extend JDpiAbstractNode. A JDpiDomain implements the central server that accept connections from the nodes involved in the network. All the incoming connections are handled by the following NodeCoordinator:

```
addNodeCoordinator(new AcceptNodeCoordinator(
  new ProtocolFactory() {
    public Protocol createProtocol() throws ProtocolException {
      Protocol protocol = new Protocol(new JDpiProtocolState(waiting, eventManager));
      return new ProtocolComposite(new ReadLocalityState(nodes), protocol);
  }
}, self));
```

AcceptNodeCoordinator is an IMC specialized NodeCoordinator that continuously waits for incoming connections: when a new connection is established this will be handled by a thread executing the protocol created through the specified ProtocolFactory. In this case we create an instance of the protocol described above and compose it (through the specialized IMC protocol, ProtocolComposite) with an initial state, ReadLocalityState (not detailed here) that reads the locality of the connected node. Even in this case we use the protocol compositionality features of IMC. Basically, AcceptNodeCoordinator implements a recurrent programming pattern for implementing a multithreaded server, that can be customized by the

programmer.

A `JDpiDomain` class keeps track of all the nodes available in a system by relying on the functionalities already provided by the IMC class `Node`. Thus, the implementation of `getNodeStack` uses these functionalities to enable the stack to communicate (either directly or indirectly with the destination).

Moreover, all the communications in a system pass through the domain. When a command is received by a domain, it simply forwards the command to the right location:

```
public JDpiReply forwardCommand(JDpiCommand command, JDpiLocality location) {
  return sendCommand(command, location);
}
```

However, a domain cannot host threads and cannot create outgoing connections. For these reasons, some of methods in `JDpiAbstractNode` have been overridden to forbid their execution:

```
public void addNodeProcess(JDpiProcess nodeProcess) {
  throw new JDpiIllegalOperation(
    "No processes can be executed at a JDpiDomain" );
}
```

$D\pi$ nodes that use this flat topology are implemented by means of class `JDpiNode`. This class extends the `JDpiAbstractNode` in such a way that:

- no incoming connections are accepted;
- only one outgoing connection to a `JDpiDomain` can be created.

A `JDpiNode` cannot play the role of an intermediary in a communication. For this reason, it provides a trivial implementation for method `forwardCommand`:

```
public JDpiReply forwardCommand(JDpiCommand command, JDpiLocality location) {
  return new JDpiReply(false,"Unknown location");
}
```

Moreover, a `JDpiNode` sends all the outgoing messages to the domain. For this reason, method `getNodeStack` simply returns the `ProtocolStack` that connects the node to the domain.

DpiF topology

To model this kind of topology class `JDpiFNode`, which extends `JDpiAbstractNode`, is introduced. A `JDpiFNode` can get connected to and accept connections from different nodes. Following the same interaction model proposed in DpiF, two nodes can interact if and only if they are directly connected. A `JDpiFNode` behaves exactly like a domain but for the fact that a `JDpiFNode` can host threads execution and can open outgoing connections.

4.3 Threads

$D\pi$ threads are implemented using the (abstract) class `JDpiProcess`, which is a subclass of the IMC class `NodeProcess` that is already equipped with code mobility support. Each process implementation must provide method `body()` that describes

behavior of the implemented process (the `NodeProcess` abstract method `execute` is implemented in `JDpiProcess` in order to perform further initialization procedures). A `JDpiProcess` interacts with the hosting node by using a `JDpiNodeProxy` described before. Indeed, `JDpiProcess` simply delegates the execution of these operations to its proxy, e.g.:

```
public <T> boolean out( JDpiChannelName<T> c , T v ) {
   return getJDpiProxy().out(c,v);
}
```

A process can migrate to remote locality `l` by invoking method `go(JDpiLocality l)`. If migration is completed successfully, process execution is terminated locally. Otherwise, `false` is returned and the process continues its execution locally.

4.4 Examples

In this section we describe two simple mobile agents implemented in `JDpi`. However, due to lack of space, all the implementation details are not presented here. We refer the interested reader to [13] where IMC and `JDpi`, with a few simple applications and examples, are available for downloading.

Example 4.1 The following $D\pi$ process that, after reading a locality from channel `ex`, spawns itself at the read locality:

$$\texttt{rec}\ X.ex?(u).\texttt{go}\ u.X$$

can be implemented as follows:

```
class MyProc extends JDpiProcess {
  public void body() {
    JDpiChannelName<JDpiLocality> inC = new JDpiChannelName<JDpiLocality>("ex");
    JDpiLocality l = in( inC );
    go(l);
  }
}
```

Example 4.2 In this small example we show how to create and use a new fresh channel name. We consider the $D\pi$ process

$$\nu a.ex!a.\mathbf{0}$$

Its `body` method can be implemented as follows:

```
public void body(){
  JDpiChannelName<String> a = new JDpiChannelName<String>();
  JDpiChannelName<JDpiChannelName<String>> outChannel =
       new JDpiChannelName<JDpiChannelName<String>>("ex");
  out(outChannel, a );
}
```

Indeed, a new channel name can be created by using the *defaul* constructur `JDpiChannelName<String>()`: The framework will select a special name (based on the address of the allocated object) that cannot be selected (or guessed) by other processes.

Example 4.3 The following is the code of an agent that migrates over a set of localities, each of which plays the role of an *electronic market*, in search of the best place where a given article (`art`) can be bought. At the end of the search, the agent migrates to locality `home` and provides its result on channel `result`.

```
public void body() {
 while (count < localities.size()) {
    JDpiChannelName<Article> c = new JDpiChannelName<Article>(art, Article.class );
    Article a = in( c );
    if (( lowestPrice == 0)||(a.getPrice()<lowestPrice)) {
      locality = localities.get(count);
      lowestPrice = a.getPrice();
    }
    count++;
    if (count < localities.size())
      go(localities.get(count));
    else
      go(home);
  }
 }
 JDpiChannelName<JDpiLocality> result =
   new JDpiChannelName<JDpiLocality>( "result" , JDpiLocality.class );
 out( result , locality );
}
```

5 Conclusions

The implementation of a language based on a process calculus typically consists of a run-time system (a sort of abstract machine) implemented in a high level language like Java, and of a compiler that, given a program written in the programming language based on the calculus, produces code that uses the run-time system above. In this paper we have illustrated, by means of a case study, a possible methodology to accelerate the development of prototype implementation of such a run-time system, by relying on the IMC framework.

In particular, we have described the implementation of $D\pi$, a well established representative of the family of mobile calculi. The use of IMC has permitted accelerating the development of prototype implementations while concentrating only on the features that are specific of the $D\pi$. The Java implementation `JDpi` is composed only by 28 classes and about 1000 lines of code. These classes provide 152 methods, and the average number of lines per method is 3.5. All the packages and the prototype of compiler are available for download at [13].

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. Since this characteristics can be found in many different calculi, IMC has a wide range of use: it could be a valid aid to implement entities like membranes [5] or ambients [7,17]. Indeed IMC has been used to model a variant of $D\pi$ ([9]) in which individual nodes may fail, or the links among them may be created and broken. Moreover, a re-implementation of KLAVA [4] using IMC is also under development.

In the close future we plan to use our framework to experiment with and compare the relative merits of the new calculi for Service Oriented Computing that are now

being developed by many research groups.

Acknowledgement

We are grateful to all people involved in the MIKADO project, in particular, we would like to thank M. Lacoste, L. Lopes and V. Vasconcelos that contributed to the initial design of IMC.

References

[1] Bettini, L., *A Java Package for Transparent Code Mobility*, in: N. Guelfi, G. Reggio and A. Romanovsky, editors, *FIDJI 2004, Int. Workshop on scientific engineering of distributed Java applications*, LNCS **3409** (2004), pp. 112–122.

[2] Bettini, L., 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. Programming Environments, Languages, Security, and Analysis of Systems, IST/FET International Workshop, GC 2003, Revised Papers*, LNCS **2874** (2003), pp. 88–150.

[3] Bettini, L., R. De Nicola, D. Falassi, M. Lacoste and M. Loreti, *A Flexible and Modular Framework for Implementing Infrastructures for Global Computing*, in: *Proc. of 5th IFIP Int. Conf. on Distributed Applications and Interoperable Systems (DAIS)*, LNCS **3543** (2005), pp. 181–193.

[4] Bettini, L., R. De Nicola and R. Pugliese, *Klava: a Java Package for Distributed and Mobile Applications*, Software - Practice and Experience **32** (2002), pp. 1365–1394.

[5] Boudol, G., *A generic membrane model*, in: *Second Global Computing Workshop*, 2004.
URL http://mikado.di.fc.ul.pt/repository/boudol_generic-membrane-model.pdf

[6] Bugliesi, M., G. Castagna and S. Crafa, *Access control for mobile agents: The calculus of Boxed Ambients*, ACM Trans. Program. Lang. Syst **26** (2004), pp. 57–124.

[7] Cardelli, L. and A. Gordon, *Mobile Ambients*, Theoretical Computer Science (TCS) **240** (2000), pp. 177–213.

[8] Fournet, C., G. Gonthier, J. J. Levy, L. Maranget and D. Remy, *A Calculus of Mobile Agents*, in: U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, LNCS **1119** (1996), pp. 406–421.

[9] Francalanza, A. and M. Hennessy, *A Theory of System Behaviour in the Presence of Node and Link Failures*, in: *CONCUR 2005*, LNCS **3653** (2005), pp. 368–382.

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

[11] Gelernter, D., *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[12] Hennessy, M. and J. Riely, *Resource access control in systems of mobile agents*, in: U. Nestmann and B. C. Pierce, editors, *Proc. of HLCL '98: High-Level Concurrent Languages*, ENTCS **16.3** (1998).

[13] *JDpi home page,* `http://music.dsi.unifi.it/jdpi`.

[14] Levi, F. and D. Sangiorgi, *Controlling Interference in Ambients*, in: *POPL* (2000), pp. 352–364.

[15] Milner, R., J. Parrow and J. Walker, *A Calculus of Mobile Processes, I and II*, Information and Computation **100** (1992), pp. 1–40, 41–77.

[16] Ravara, A., A. Matos, V. Vasconcelos and L. Lopes, *Lexically scoping distribution: what you see is what you get*, in: *FGC: Foundations of Global Computing*, ENTCS **85(1)** (2003).

[17] Sangiorgi, D. and A. Valente, *A Distributed Abstract Machine for Safe Ambients*, in: *Proc. 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*, LNCS **2076** (2001), pp. 408–420.

[18] Unyapoth, A. and P. Sewell, *Nomadic Pict: correct communication infrastructure for mobile computation*, in: *POPL* (2001), pp. 116–127.

[19] Vitek, J. and G. Castagna, *Seal: A Framework for Secure Mobile Computations*, in: *Internet Programming Languages*, number 1686 in LNCS, Springer, 1999 .