IMC: Flexible Communication Protocol Support for Implementing Mobile Process Calculi

MARC LACOSTE
France Telecom R&D
marc.lacoste@rd.francetelecom.com

Existing implementations of distributed process calculi exhibit a wide diversity of communication protocols and interaction semantics between locations. Thus, a core framework, such as IMC (*Implementing Mobile Calculi*), aiming to provide the programmer with a toolbox to build implementations of new process calculi, should be flexible enough to support multiple marshalling strategies, while being minimal to enable an easy introduction of new communication protocols. In this paper, we report on the IMC Protocol Management Framework, a Java class library which allows to define customized protocol stacks by composing micro-protocols in a flexible manner. We describe in detail both its interfaces and mechanisms, focusing on the TCP/IP layer, which constitutes the network interface of most protocols stacks used in existing process calculi implementations. In particular, we illustrate how new protocols can be introduced with IMC, by showing the protocol and session objects involved, and by describing the path followed by messages within a protocol stack.

1. INTRODUCTION

When looking at existing implementations [Boreale et al. 2002a] of some common models of distributed programming such as process calculi [Boudol et al. 2002a], one is struck by the diversity of protocols and programming languages used. In first approximation, the communication protocols can be split into two families: high-level protocols such as Java RMI are well integrated with the Java Virtual Machine environment and take advantage of the architectural independence provided by Java (SAM [Sangiorgi and Valente 2001] implementation of Safe Ambients [Levi and Sangiorgi 2000]) ; protocols closer to hardware resources, such as socket-based implementations of TCP/IP, are accessible, either directly in Java (KLAIM/KLAVA [Bettini et al. 2002]) or in other programming languages allowing easier manipulation of system resources such as OCaml (JCL [Fournet and Maranget 1997] and JOCAML [Le Fessant 1998]) or C (DITYCO [Lopes 1999]). Marshalling strategies range from dedicated byte-code structures (JCL, JOCAML, DITYCO) to Java serialization (SAM, KLAIM/KLAVA).

Thus, a common software framework to build various implementations of process calculi cannot restrain itself to a fixed set of communication primitives or marshalling strategies. Instead, an important design requirement for such a middleware like IMC (*Implementing Mobile Calculi*) [Bettini et al. 2003] is to be flexible enough to support multiple marshalling strategies and communication protocols with possible introduction of new strategies and customized protocols. This framework should also aim at minimality to enable an easy introduction of new protocols.

Marc Lacoste, France Telecom R&D, 28 Chemin du Vieux Chêne, 38243 Meylan, France.
This work has been partially supported by EU within the FET - Global Computing Initiative in project MIKADO IST-2001-322222.

This is precisely the objective of the IMC Protocol Management Framework [Mikado Project 2003]: this Java framework allows to define customized protocol stacks by a flexible composition of micro-protocols. In practice, a typical protocol stack found in the implementation of a new process calculus is quite likely to use TCP/IP as the lowest layer of interface to the network. Thus, an in-depth understanding of the TCP/IP communication mechanisms within IMC should be helpful to the programmer to define easily new protocols using IMC.

The aim of this document is to try to summarize the abstractions and implementation classes that make up the TCP/IP protocol layer within the IMC Protocol Management Framework. The goal is to describe how flexible protocols can be designed and implemented in a network of computational nodes.

The IMC Protocol Management interfaces and classes can be found in two main packages:

- The `org.mikado.imc.protocols.apis` package contains some generic IMC interfaces, in particular those concerning the TCP/IP protocol layer.
- The `org.mikado.imc.protocols.libs` package contains the classes which implement those interfaces. A high-level view of TCP/IP is given by a collection of objects of various kinds: sessions, session identifiers, protocols, and protocol graphs. At a lower level, connections represent the real TCP/IP communication channels by encapsulating Java sockets, connection factories create new connections, and connection managers handle the lifecycle of the various connections.

The rest of the paper is organized as follows. Section 2 recalls some of the main abstractions of the IMC Framework. A quick overview of the corresponding Java interfaces is also given. Then, sections 3 and 4 show how these notions can be put in practice, using as a case study the TCP/IP communication layer. Section 3 describes the interface part, while section 4 gives more details concerning the TCP/IP implementation. The last section illustrates over a simple example – the "knock-knock" protocol – how to combine protocols and session objects in order to design flexible communication protocols using IMC.

This document is very much inspired from the `javadoc` documentation of the IMC Protocol Management Framework, itself an adaptation of the Jonathan Object Request Broker distribution [Dumant et al. 1999; Krakowiak 2002; ObjectWeb Consortium 2002]. We present the IMC classes in a "in a nutshell" style [Flanagan 1999]: the purpose of each IMC class is described, and the functions of its main methods are explained. Hints are given as to how the methods are implemented, which may help in understanding the code. Finally, the Java interface of the class is presented.

2. IMC MAIN ABSTRACTIONS

This section provides a quick overview of the main abstractions used throughout this document. These concepts represent the founding notions behind the IMC Java framework. Understanding these notions will help the designer and programmer of a process calculus implementation to take advantage of the flexibility provided by IMC concerning protocols: thus, a programmer may select and then implement the communication protocol which best matches the needs of the platform currently under development. A precise description of the interfaces which are cited here can be found directly in the IMC CVS repository, on the MIKADO project web site [Mikado Project 2003]. Only the abstractions directly relevant to protocol management are presented in this section.

2.1 Naming and Binding

The first IMC abstractions are objects, or basic entities manipulated through their interfaces:

• *Object.* The "objects" of interest when developping an application or a library are conceptual entities of the application or library that may be of an arbitrary complexity and granularity. They can be thought of as components encapsulating a state and information processing. These components, which we call *objects*, may be accessed from their environment at specific points called *interfaces*, each of them having its own identity. These components may be implemented using one or several Java objects.

• *Interface.* An identified access point to an object in the sense of the ODP Reference Model [International Standards Organization 1995].

The main IMC abstractions deal with name management: designating a remote object is clearly separated from the instanciation of the communication resources necessary for a remote invocation to occur:

• *Identifier.* A name that uniquely designates an object interface in a given naming context. An identifier stands for a generic notion of name in IMC, and is represented by the `Identifier` interface.

• *Naming Context.* The role of a naming context is to create and manage names, ensuring that each of the names it manages unambiguously designates a given object interface. For instance, the first example of a naming context is the Java virtual machine itself, that unambiguously designates the objects by their address in memory. The value of an address is not accessible to the Java programmer, but the object identity is manifested to the programmer through an object reference. An object implementing the `NamingContext` interface stands for the most generic notion of a naming context : it manages names of type `Identifier`, used to designate applicative object interfaces in specific ways – distributed, persistent, . . .

2.2 Communication Protocols and Sessions

The IMC communication package is built around three main abstractions: protocols, protocol graphs and sessions.

• *Protocol.* It represents protocols like TCP, IP, or GIOP. A protocol is a naming context that can only deal with a specific family of interfaces called *sessions*, and that manages names called *session identifiers*. A protocol is represented by the `Protocol` interface, or in the case of TCP/IP by the `TcpIpProtocol` class.

• *Protocol Graph.* A directed acyclic graph composed of protocol-specific nodes, and describing a path to be followed by messages when they are sent over the network, or received. A given session can be exported by calling the `export` method on the root of a protocol graph – the appropriate calls will be recursively issued on its sub-graphs. A session identifier is then returned to designate the exported session. A protocol graph is represented by the `ProtocolGraph` interface, or in the case of TCP/IP by the `TcpIpProtocol.TcpIpProtocolGraph` class.

• *Session.* An abstraction for a handle on particular communication channels. A session object is dynamically created by a protocol, and lets messages be sent and received through the communication channel it represents, using that protocol. Sessions have higher and lower interfaces, respectively used to send messages down and up a protocol stack. A session encapsulates *connections* which represent the real communication channels.

Sessions have higher and lower interfaces, and are designated by *session identifiers*:

• *Session_High Interface.* A kind of session used to send messages down to the network. It will usually be some kind of surrogate for a session of type `Session_Low`, which has been exported to a protocol object and is designated by a `Session Identifier` object. A `Session_High` instance may be obtained using the `bind` operation on a session identifier representing a `Session_Low` interface: it is thus a surrogate, or a proxy, for that interface.

• *Session_Low Interface.* A kind of session used to forward messages coming from the network to their actual recipient. `Session_Low` is also the type of interfaces exported to protocols, and designated by session identifiers.

• *Session Identifier.* A *session identifier* is a name, managed by a specific naming context of type `Protocol`, which represents an exported session. The internal structure of a session identifier is protocol-specific: each protocol may define its own `Session Identifier` type to contain protocol-specific information: host name, port, etc. In the case of TCP/IP, a session identifier encapsulates a host name and port number and is represented by the `TcpIpIdentifier` interface. Session identifiers are created when server objects are exported, and can typically be encoded in identifiers. On the client side, they can be decoded and used to establish communication channels.

Some sessions may be created using dedicated *session factories*:

• *Server Session Factory.* A factory for server sessions. Just as sessions encapsulate connections, a server session factory is a higher-level abstraction to a server connection factory to listen for incoming connections from clients. The factory encapsulates a server session identifier, and a higher-level session. A server session factory delegates the work of listening to connections from clients to the server connection factory hidden inside the session identifier. In the case of TCP/IP, a server session factory is implemented by the `SrvSessionFactory` class.

Finally, marshallers and unmarshallers are abstractions of messages which move up and down a protocol stack:

• *Marshalller.* A high-level encoding-independent representation of messages that are about to be sent. It is represented by the `Marshaller` interface.

• *Unmarshaller.* A high-level encoding-independent representation of messages that are about to be received. It is represented by the `UnMarshaller` interface.

• *Marshaller factory.* A factory for both marshallers and unmarshallers. A class implementing the `MarshallerFactory` interface will both create objects implementing the `Marshaller` interface and objects implementing the `UnMarshaller` interface.

2.3 Connection Management

A session encapsulates *connections* which represent the "real" communication channels:

• *Connection.* An encapsulation of a regular socket, e.g., typically an instance of the `java.net.Socket` class. In the case of TCP/IP, a connection is represented by the `TcpIpConnection` interface.

• *Connection Factory.* A fairly low-level abstraction. Its only purpose is to build client-side or server-side connections on demand. For requests concerning the creation of a server-side connection, a connection factory creates server connection factories. These

objects will create connections if there is an incoming connection request from a client. A typical example of a connection factory is the `IPv4ConnectionFactory` class.

• *Connection Manager.* A higher-level abstraction than the connection factory. Its purpose is to manage connections, by keeping track of idle and active connections. For the creation of connections, it will delegate most of the work to a connection factory. A typical example of a connection manager is the `JConnectionMgr` class. A TCP/IP connection manager is represented by the `TcpIpConnectionManager` interface.

• *Server Connection Factory.* An encapsulation of a server socket, e.g., typically an instance of the `java.net.ServerSocket` class. In the IMC terminology, the equivalent of the `ServerSocket.accept` method is a `newSrvConnection` method which returns a connection, i.e., a socket for the client and server to exchange data later on. A TCP/IP server connection factory is represented by the `TcpIpSrvConnectionFactory` interface.

2.4 Resource Management

IMC also provides interfaces to customize the marshalling and unmarshalling of messages:

• *Chunk.* A part of an array of bytes. Chunks are linked to form messages that may be sent from an address space to another. Their use avoids unnecessarily copying arrays of bytes, and helps recovering these arrays without resorting to garbage collection.

• *Chunk Factory.* It builds chunks, possibly implementing a specific chunk management policy. It is represented by the `ChunkFactory` interface.

• *Chunk Provider.* These objects are used to wrap input streams – like socket input streams – in unmarshallers.

The framework also provides basic activity and scheduling primitives, e.g., allowing to program multiple concurrent connections in a connection manager:

• *Job.* The abstraction of an activity in the system. When this activity is sequential (the usual case), it may be thought of as a thread. The role of a job is to execute code. Jobs are created and managed by *schedulers*. They are represented by the `Job` interface.

• *Scheduler.* The role of a scheduler is to create jobs, i.e., a scheduler is a job factory. It manages jobs according to their properties – a priority, a deadline, ... A scheduler is represented by the `Scheduler` interface.

2.5 Customization

The IMC protocol management package may be customized using the *kernel* abstraction, which provides a bootstrap environment that objects in the system can use to retrieve configuration parameters or references to other objects. It is represented by the `Kernel` abstract class.

3. PROGRAMMING INTERFACES

This section now illustrates how the notions described previously can be put in practice, using as a case study a Java implementation of a TCP/IP protocol stack with the IMC framework. This example assumes a basic knowledge of the IMC generic interfaces, like `Identifier`, `NamingContext`, `Protocol`, `Session`, ..., which are described in [Bettini et al. 2003] and are not reproduced here. We only describe the main interfaces and classes relating to TCP/IP.

3.1 `TcpIpConnection` Interface

A TCP/IP connection represents a communication resource for the TCP/IP protocols, and encapsulates a socket. A connection is always related to a given session, i.e., a communication channel. The `getSession` method returns the session attached to this connection, while `setSession` attaches a new session to this connection. The `getInputStream` method returns the input stream to read from this connection, while `getOutputStream` returns the output stream to write to this connection. The `getPort` and `getHostName` methods return the port number and host name of the underlying socket. The `release` method is used to indicate to the connection manager that this connection is no longer used. The `delete` method deletes this connection, removing it from the connection manager, and closing the socket.

```
public interface TcpIpConnection {
   TcpIpSession getSession();
   void setSession(TcpIpSession session);
   InputStream getInputStream();
   OutputStream getOutputStream();
   short getPort();
   String getHostName();
   void release();
   void delete();
}
```

3.2 `TcpIpConnectionMgr` Interface

This interface defines a type for TCP/IP connection managers. The `newCltConnection` method returns a new client connection. This method is called by a protocol object, and should provide in parameters the host name of the distant server, the port number of a server socket on that host, and a TCP/IP session. The `newSrvConnectionFactory` method returns a new server endpoint encapsulating a server socket on the provided port. The `getCanonicalHostName` method returns the canonical host name of the provided host, the purpose of this method being to help checking the equality of two host names.

```
public interface TcpIpConnectionMgr {
   TcpIpConnection newCltConnection(String host,
                                    short port,
                                    TcpIpSession session)
      throws IMCProtocolException;
   TcpIpSrvConnectionFactory newSrvConnectionFactory(short port)
      throws IMCProtocolException;
   String getCanonicalHostName(String hostname);
}
```

3.3 `TcpIpSession` Interface

A TCP/IP session is an abstract represention of a TCP/IP communication channel. The `getHls` method returns the higher level session associated with the target session, i.e., the session to which the data read from the associated TCP/IP connection will be sent.

```
public interface TcpIpSession  extends Session_High {
   public Session_Low getHls();
}
```

3.4 `TcpIpSessionIdentifier` Class

A TCP/IP session identifier contains a host name, and a port number. The constructor builds a new `TcpIpSessionIdentifier` with the specified host name and port number. The `equals` method returns `true` if the `TcpIpSessionIdentifier` object passed in argument has the same host name and port number than those of the target object.

```
abstract public class TcpIpSessionIdentifier
      implements SessionIdentifier {
   public String hostname;
   public short port;
   public TcpIpSessionIdentifier(String hostname,
                                 short port);
   public TcpIpSessionIdentifier();
   public int hashCode();
   public boolean equals(Object o);
   public String toString();
}
```

3.5 `TcpIpSrvConnectionFactory` Interface

The `TcpIpSrvConnectionFactory` interface encapsulates a server socket. The `newSrvConnection` method returns a new connection for the provided session, obtained from the encapuslated server socket. The `getPort`, and `getHostName` methods return the local port and host name of the encapsulated socket. The `close` method tells the factory to release the server socket resources.

```
public interface TcpIpSrvConnectionFactory {
   TcpIpConnection newSrvConnection(TcpIpSession session)
      throws IMCProtocolException;
   short getPort();
   String getHostName();
   void close();
}
```

4. IMC PROTOCOL IMPLEMENTATION

4.1 `TcpIpProtocol` Class

This class is an implementation of a TCP/IP stack. The `verbose` attribute indicates whether warning messages should be printed to `stderr`. The `TcpIpProtocol` object is built from instances of a kernel, a connection manager, a scheduler, a chunk factory, and a marshaller factory. It also maintains a pool of `SrvSessionFactory` instances.

The `newProtocolGraph` method creates a new protocol graph. The `newSessionIdentifier` method creates new client session identifiers – `TcpIpSessionIdentifier` instances – using the provided host name and port number. The `isLocal` method returns `true` if the provided `TcpSessionIdentifier` corresponds to a local connection managed by this protocol. The `remove` method removes a `SrvSessionFactory` object from the pool of instances maintained in the current class. The `send` method sends a message represented by a `Marshaller` object over the network, on the provided output stream. This method is called by the `send(Marshaller message)` method of client-side and server-side sessions.

```
final public class TcpIpProtocol implements Protocol {
   public boolean verbose;
   public TcpIpProtocol(Kernel kernel,
                        TcpIpConnectionMgr connection_mgr,
                        Scheduler scheduler,
                        ChunkFactory chunk_factory,
                        MarshallerFactory mf)
      throws IMCProtocolException;
   public final boolean isAnInvocationProtocol();
   public final ProtocolGraph newProtocolGraph();
   public final ProtocolGraph newProtocolGraph(short port);
   public final TcpIpSessionIdentifier newSessionIdentifier(String host,
                                                            short port);
   public boolean isLocal(TcpIpSessionIdentifier tcp_session_id);
   boolean isLocal(String host,
                   short port);
   void remove(SrvSessionFactory factory);
   void send(Marshaller message,
             OutputStream os)
      throws IOException;
}
```

4.2 `TcpIpProtocol.TcpIpProtocolGraph` Class

Recall that a protocol graph is a directed acyclic graph composed of protocol-specific nodes, and describing a path to be followed by messages when they are sent over the network, or received. This class implements the `ProtocolGraph` interface for the TCP/IP protocol. The main method of this class is the `export` method which informs the TCP/IP protocol that the provided session is willing to accept messages. It returns a `SessionIdentifier` object instance – a specific name that designates the exported session.

The body of the `export` method first checks to see if it cannot reuse an existing server session factory instance. If so, it returns the server session identifier which designates the session on behalf of which the server session factory listens for connection requests from clients. If not, it asks the connection manager for a new server connection factory object, i.e., a new server socket. It builds a new server session identifier from that connection factory. It also creates a new server session factory object from that session identifier, the exported session playing the role of the higher-level session, in the stack of server-side sessions. It then spawns a new thread that executes the body of the `run` method of the newly created server session factory. Finally, this server session factory object is added to the pool of instances maintained by the TCP/IP protocol object, and the `export` method returns the server session identifier.

```
final class TcpIpProtocolGraph
      implements ProtocolGraph {
   TcpIpProtocolGraph(short port);
   TcpIpProtocolGraph();
   public SessionIdentifier export(Session_Low hls)
      throws IMCProtocolException;
}
```

4.3 `TcpIpProtocol.CltSessionIdentifier` Class

This class extends the `TcpIpSessionIdentifier` class by implementing the `unexport` and `bind` methods since the `TcpIpSessionIdentifier` abstract class

implements the `SessionIdentifier` interface. The `unexport` method is meaningless since we are on the client side. The `bind` method returns a `Session_High` object which is a surrogate for a `Session_Low` type of session which has been exported to a protocol object, and is designated by the target `CltSessionIdentifier` object. In other words, the current class designates some remote session, i.e., some host name and port number. A connection to that remote session is established, and a client-side session object encapsulating that connection is returned by the `bind` method.

The body of the `bind` method creates a client session, giving `hls` as the higher-level session parameter, and `this` as the session identifier parameter to the constructor of the `CltSession` class. It then calls the `newCltConnection` method of the connection manager maintained by the TCP/IP protocol instance to obtain a `TcpIpConnection` object attached to the client session. Finally, it calls the `connect` method of the client session with the connection as a parameter. The former operation sends any incoming message (or exception if things go wrong) from the network up the protocol stack to the higher-level session.

```
final class CltSessionIdentifier
      extends TcpIpSessionIdentifier {
   CltSessionIdentifier(String hostname,
                        short port);
   public Protocol getProtocol();
   public void unexport();
   public Session_High bind(Session_Low hls)
      throws IMCProtocolException;
}
```

4.4 `TcpIpProtocol.SrvSessionId` Class

This class designates a server-side TCP/IP session. Since it extends the `TcpIpSessionIdentifier` class, it contains a host name and a port number. It also encapsulates a `TcpIpSrvConnectionFactory` instance – an encapsulation of a server socket. The `getProtocol` method returns a `TcpIpProtocol` instance representing the TCP/IP protocol. The `unexport` method closes the server connection factory. Thus, the underlying server socket is released. Here, the `bind` operation is meaningless.

```
final class SrvSessionId
      extends TcpIpSessionIdentifier {
   SrvSessionId(TcpIpSrvConnectionFactory connection_factory);
   public Protocol getProtocol();
   public void unexport();
   public Session_High bind(Session_Low hls)
      throws IMCProtocolException;
}
```

4.5 `TcpIpProtocol.Session` Class

Recall that a `TcpIpSession` interface extends the `Session_High` interface – it is a kind of session used to send messages down to the network, and will usually be some kind of surrogate for a session of type `Session_Low`, which has been exported to a protocol object and is designated by a `SessionIdentifier` object.

This class is an implementation of the `TcpIpSession` interface. It encapsulates a TCP/IP connection, and a higher-level session : a protocol stack can be viewed as a stack

of sessions ; the higher-level session is the session immediately above the current class in this stack of sessions.

The `getHls` method returns the higher-level session, and the `getConnection` method returns the TCP/IP connection attached to this session. The `getProtocol` method returns the current TCP/IP protocol object. The `direct` method returns true, as the TCP/IP protocol headers are added when then the message is sent, and not before.

The `connect` method connects to the provided `TcpIpConnection` object. It reads all data from the input stream attached to this connection, and saves it in a message – a `TCPChunkProvider` instance. It then spawns a new thread that executes the body of the `run` method. The former method sends any incoming message (or exception if things go wrong) up the protocol stack to the higher-level session, passing in parameter the current instance of `TcpIpSession` for a possible reply. It does so by calling the `send` method of the higher-level session.

The `closeNotify` method is called by the `TCPChunkProvider` class, and sends any incoming message up the protocol stack. Finally, the `unbind` method deletes the connection attached to this session.

```
abstract class Session
      implements TcpIpSession, Runnable {
   Session(Session_Low hls);
   final public void prepare(Marshaller m);
   final public ReplyInterface prepareInvocation(Marshaller m)
       throws IMCProtocolException;
   final public boolean direct();
   public final Session_Low getHls();
   public void connect(TcpIpConnection connection)
       throws IMCProtocolException;
   public final TcpIpConnection getConnection();
   final public void run();
   final void closeNotify(TcpIpChunkProvider message);
   void unbind();
   TcpIpProtocol getProtocol();
}
```

4.6 `TcpIpProtocol.SrvSession` Class

This class implements a server-side TCP/IP session. The `send` method is used to send a message represented by a `Marshaller` object down to the network. It does so by calling the `send` method of the `TcpIpProtocol` class.

```
final class SrvSession extends Session {
   SrvSession(Session_Low hls);
   final public void close();
   public final void send(Marshaller message)
      throws IMCProtocolException;
}
```

4.7 `TcpIpProtocol.CltSession` Class

This class implements a client-side TCP/IP session. A new instance can be created by invoking the constructor, providing as parameters the higher-level session in the stack of sessions, and a `TcpIpSessionIdentifier` class instance. The `send` method is used to send a message represented by a `Marshaller` object down to the network. It does so by calling the `send` method of the `TcpIpProtocol` class. The `rebind` method

drops the connection attached to this session, and asks the connection manager for a new connection on the same host name, and port number. The `equals` method returns `true` if two client-side TCP/IP connections have the same session identifier, and the same higher-level session. The `close` method releases the TCP/IP connection attached to this session.

```
final class CltSession extends Session {
   public final void send(Marshaller message)
      throws IMCProtocolException;
   void rebind() throws IMCProtocolException;
   final public void close();
   public boolean equals(Object object);
   public int hashCode();
}
```

4.8 `SrvSessionFactory` Class

This class serves a double purpose. It is a factory for server session objects, and it starts to listen for connections requests from clients by indirectly calling the `accept` method of a server socket. A pool of instances of this class is maintained by the TCP/IP protocol object. Such an instance is built from a TCP session identifier that designates a server session, the session object above the former server session in the stack of sessions, and the TCP/IP protocol object.

The `register` method checks if the target server session factory listens for connection requests from clients on the port of the provided protocol graph, and if the higher-level session matches with the provided one. If so, it returns the server session identifier that encaspulates a server connection factory listening on that port.

The body of the `run` method is executed during the `export` operation on the server-side: it extracts the server connection factory from the server session identifier. It then creates a new server session – an `SrvSession` instance. It calls the `newSrvConnection` method of the connection factory. This last operation is in fact similar to calling the `accept` method of the server socket hidden in the server connection factory. If there is an incoming connection request from a client, this method will return a `TcpIpConnection` instance – the equivalent of a regular socket for the server and client to exchange data later on. The `connect` method of the server session is then called : any incoming messages from the network will be sent to the higher-level sessions, on the server side.

The `release` method unexports the server session identifier – this will close the server connection factory, and the underlying the server socket.

```
final class SrvSessionFactory
       implements Runnable {
   SrvSessionFactory(SrvSessionId session_id,
                     Session_Low hls,
                     TcpIpProtocol protocol);
   SrvSessionId register(Session_Low hls,
                         TcpIpProtocolGraph protocol_graph);
   public void run();
   void release();
}
```

4.9 `TcpIpProtocol.TcpIpChunkProvider` Class

The `TcpIpChunkProvider` is a `ChunkProvider` implementation encapsulating a socket input stream. It can be built from a TCP/IP session, and from an input stream, or from another `TcpIpChunkProvider`. The chunk provider contains a cache. The `prepare` method reads all readable data from the input stream, stores this data in the cache, and returns the content of the cache as a chunk. The `delete` method releases the chunks associated with the cache data. Eventually, they will be freed when the `finalize` method is called. The `duplicate` method returns a chunk which is a duplicate of the cache data. The `close` method calls the `closeNotify` method of the underlying session.

```
final class TcpIpChunkProvider
      extends Chunk implements ChunkProvider  {
   TcpIpChunkProvider(TcpIpProtocol.Session session,
                      InputStream is);
   TcpIpChunkProvider(TcpIpChunkProvider message);
   public Chunk prepare()
      throws IMCProtocolException;
   public void close();
   protected void finalize();
   final void delete();
   public Chunk duplicate()
      throws IMCProtocolException;
   public Chunk duplicate(int off,int t)
      throws IMCProtocolException;
   public void release();
}
```

4.10 `IPv4ConnectionFactory` Class

This class implements an IPv4 connection factory. The `verbose` attribute indicates whether warning messages should be printed to stderr. The `backlog` attribute represents the maximum length of the queue on incoming connect requests.

The static `init` method builds a new `IPv4ConnectionFactory` instance by calling the constructor with the specified kernel. The `newCltConnection` method builds a new client-side connection with the provided parameters : it builds a regular, i.e., client-side, socket with the provided host name, and port number, and returns a `Connection` object by supplying it with the socket, the former parameters, and a `TcpIpSession` instance.

The `newSrvConnectionFactory` method returns a new server connection factory that encapsulates a server socket on the provided port. This method returns a `SrvConnectionFactory` object. The `getCanonicalHostName` method returns the canonical host name – the string corresponding to the IP address. The `newSrvConnection` method builds a new server-side connection with the provided parameters – a socket, a TCP/IP session, a host name, and a port number. This method is called by the `newSrvConnection` method in the `SrvConnectionFactory` class.

```
public class IPv4ConnectionFactory
      implements TcpIpConnectionMgr {
   public boolean verbose;
   public int backlog;
   public static Object init(Kernel kernel)
      throws IMCProtocolException;
   protected IPv4ConnectionFactory(Kernel kernel)
      throws IMCProtocolException;
   public TcpIpConnection newCltConnection(String host,
                                           short port,
                                           TcpIpSession session)
      throws IMCProtocolException;
   public TcpIpSrvConnectionFactory newSrvConnectionFactory(short port)
      throws IMCProtocolException ;
   public String getCanonicalHostName(String hostname);
   protected Connection newSrvConnection(Socket socket,
                                         TcpIpSession session,
                                         String host,
                                         short port)
      throws IMCProtocolException;
}
```

4.11 `IPv4ConnectionFactory.Connection` Class

This class implements the `TcpIpConnection` interface by encapsulating a socket. The `getInputStream` method returns the input stream to read from this connection, while the `getOutputStream` returns the outputstream to write to this connection. Methods `getHostName`, and `getPort` return the host name, and port number of the underlying socket. Methods `getSession` and `setSession` return and update the session this connection is attached to. Method `delete` deletes this connection, removing it from the connection manager, and closing the socket. Method `release` releases this connection to indicate to the connection manager that the target connection is no longer used.

```
public class Connection
      implements TcpIpConnection {
   protected Connection(Socket socket,
                        TcpIpSession session,
                        String host,
                        short port)
      throws IMCProtocolException;
   public InputStream getInputStream();
   public short getPort();
   public String getHostName();
   public TcpIpSession getSession();
   public void delete();
   public void release();
}
```

4.12 `IPv4ConnectionFactory.SrvConnectionFactory` Class

This class implements the `TcpIpSrvConnectionFactory` interface that encapsulates a server socket. An instance is built by calling the constructor with a hostname, a port number, and the maximum length of the queue on incoming connect requests. The `newSrv Connection` method calls `accept` on the encapsulated server socket. If there is an incoming connection request from a client, it returns a `Connection` object encapsulating the socket returned by `accept` that the server can use to communicate with the client. It does so by calling the `newSrvConnection` method in class `Ipv4ConnectionFactory`.

```
class SrvConnectionFactory
       implements TcpIpSrvConnectionFactory {
   SrvConnectionFactory(InetAddress localhost,
                        short port,
                        int backlog)
       throws IMCProtocolException;
   public TcpIpConnection newSrvConnection(TcpIpSession session)
       throws IMCProtocolException;
   public short getPort();
   public String getHostName();
   public void close();
}
```

4.13 `JConnectionMgr` Class

This class is a default implementation of a connection manager. It is also a wrapper of a connection factory object (typically an `IPv4ConnectionFactory` instance) to which the connection manager delegates most of its method implementations. A connection manager instance can be built from a kernel, and a scheduler. The connection manager maintains a linked list of idle connections, the list elements being instances of the `Connection` class. The `max_idle` attribute is the maximum number of idle connections kept by this manager.

The `getCanonicalHostName` method returns the canonical host name of the provided host. The `newCltConnection` method returns a new client connection to a protocol object. Internally, the class maintains a hashtable that maps a hash code built from a host name, and port number to `Connection` objects. This hashtable represents the set of managed connections. The creation of the `Connection` objects themselves is delegated to the underlying connection factory. The `add` and `remove` methods add / remove a new connection to / from the set of managed connections. These methods also update the linked lists of connections maintained by the connection manager.

The connection manager also maintains a linked list of `SrvConnectionFactory` objects. When invoked, these objects will return a new server endpoint encapsulating a server socket on the provided port. The `newSrvConnectionFactory` method adds a new element to such a list, or returns an existing one if the corresponding port numbers match. The `removeConnectionFactory` method removes an existing server connection factory from the list.

```
public class JConnectionMgr
      implements TcpIpConnectionMgr {
   public int max_idle;
   public JConnectionMgr(Kernel kernel,
                         Scheduler scheduler)
      throws IMCProtocolException;
   public String getCanonicalHostName(String hostname);
   public TcpIpConnection newCltConnection(String host,
                                           short port,
                                           TcpIpSession session)
      throws IMCProtocolException;
   public TcpIpSrvConnectionFactory newSrvConnectionFactory(short port)
      throws IMCProtocolException;
   protected Connection newCltConnection(TcpIpConnection connection)
   void remove(Connection connection);
   void add(Connection connection);
   void removeConnectionFactory(SrvConnectionFactory factory);
   void rehash(int len);
}
```

4.14 `JConnectionMgr.Connection` Class

This class is a list element in several linked lists (idle connections list, collisions list) of connections maintained by the `JConnectionMgr` class. It is a wrapper class for a class that implements the `TcpIpConnection` interface, typically an `IPv4Connection Factory.Connection` class. The methods of this class call the methods bearing the same names in the wrapped connection class. Besides, they update the linked lists maintained by the `JConnectionMgr` class.

The `getInputStream` method returns the input stream to read from this connection, while the `getOutputStream` returns the outputstream to write to this connection. Methods `getSession`, and `setSession` return and update the session this connection is attached to. Method `delete` deletes this connection, removing it from the connection manager, and closing the socket. Method `acquire` returns when the socket is acquired. This information is taken into account by the connection manager to avoid closing connections still in use. Method `release` releases this connection to indicate to the connection manager that the target connection is no longer used.

```
public class Connection
      implements TcpIpConnection {
   protected Connection(TcpIpConnection delegate)
      throws IMCProtocolException;
   public InputStream getInputStream();
   public OutputStream getOutputStream();
   public short getPort();
   public String getHostName();
   public TcpIpSession getSession();
   public void setSession(TcpIpSession session);
   public void acquire();
   public void release();
   void withdraw();
}
```

4.15 `JConnectionMgr.SrvConnectionFactory` Class

This class is a list element in the linked list of server connection factories maintained by the `JConnectionMgr` class. It is a wrapper class for a class that implements the `TcpIpSrvConnectionFactory` interface, typically an `IPv4ConnectionFactory.SrvConnectionFactory` class. The methods of this class call the methods bearing the same names in the wrapped server connection factory class. Besides, the `close` method removes this list element from the aforementioned linked list.

```
class SrvConnectionFactory
      implements TcpIpSrvConnectionFactory {
   SrvConnectionFactory(TcpIpSrvConnectionFactory delegate)
      throws IMCProtocolException;
   public TcpIpConnection newSrvConnection(TcpIpSession session)
      throws IMCProtocolException;
   public short getPort();
   public String getHostName();
   public void close();
}
```

5. DEFINING NEW PROTOCOLS WITH IMC

We now illustrate how to define new protocols over the TCP/IP layer using IMC. We consider a very simple authentication protocol over TCP/IP bindings, the "knock-knock" protocol, whose steps are shown on figure 1.

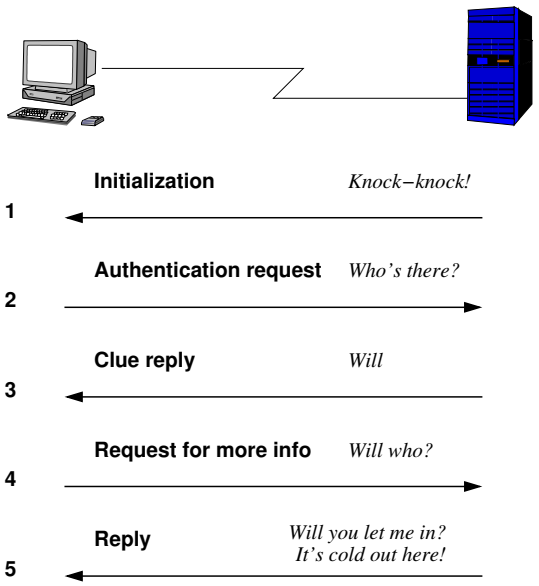

Fig. 1 – The "Knock-Knock" Protocol.

5.1 A "Knock-Knock" Client

A simple "knock-knock" client is given in section A. The client begins by asking the user for a host name, and a port number. It then creates a new instance of the `TcpIpProtocol` class, and calls its `newSessionIdentifier` method to get a client session identifier, corresponding to those parameters. The next step is to establish the communication channel by calling the `bind` method on this identifier. The returned `Session_High` session object can then be used to send messages over the network. Here, a single byte is sent. Finally, the client waits for replies from the server.

The higher-level session of the TCP/IP session object is a `KnockKnockClientSession` which reads directly data from the standard input. On reception of a message, or of an exception from the network, the `KnockKnockClientSession.send` method will be called. In the case of a message, its content is also displayed on the standard output, and any message typed on the standard input device, will be sent to the lower-level session. This will eventually call the `send` method of the `TcpIpProtocol.CltSession` class. The various session and protocol objects involved on the client-side are shown on figure 2.

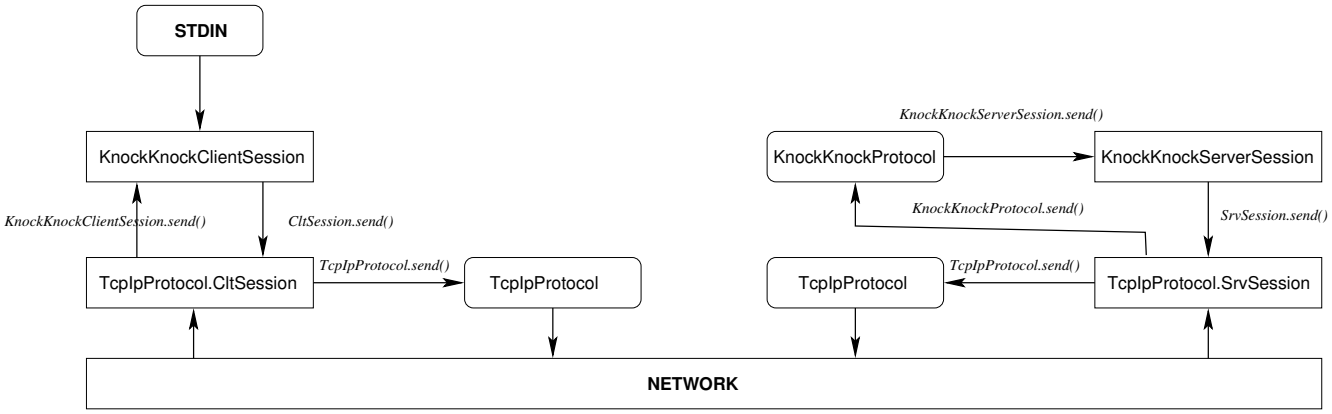

Fig. 2 – Session and Protocol Objects in the "Knock-Knock" Protocol.

5.2 A "Knock-Knock" Server

A simple "knock-knock" server is given in section B. The server begins by creating a new instance of the `TcpIpProtocol` class. The server object which is exported to that protocol is a `KnockKnockProtocol` instance. The "knock-knock" protocol object creates a hashtable where it can store "knock-knock" session objects. The `export` method creates a `SrvSession` object whose higher-level session is the `KnockKnockProtocol` instance. A server socket is created to listen for connection requests from clients.

When a client connects, the `SrvSession.connect` method is called: this reads all messages from the network, and transfers them to the higher-level session – the `KnockKnockProtocol` object. Thus, the `send` method of that class is called with the `SrvSession` object as a second parameter for a possible reply. This `SrvSession` session is, at the beginning, unknown of the hashtable maintained by the `KnockKnockProtocol` object. A new `KnockKnockSession` object is then created, and a new entry in the hashtable is created containing this session, and the sender session i.e. the `SrvSession` session object, so that the "knock-knock" session may know which session to reply to. The `send` method of the "knock-knock" session is then invoked.

The "knock-knock" session `send` method is the heart of the "knock-knock" protocol. It reads the initialization message – a single byte – from the client, and creates a reply message by filling a marshaller with the content of the `theOutput` string. At the beginning of the KK protocol, the content of that string is "Knock Knock!". At later stages of the protocol, the reply is based on the input from the client. The message is then sent to the sender, i.e., the `send` method of the `SrvSession` object is called. This method will in turn call the `send` method of the TCP/IP protocol object which does the real work of sending the data over the network. If the client does not want to dialog any more, a "bye" message is returned by the KK session. The KK session is then closed, and removed from the list of sessions maintained by the KK protocol. The various session and protocol objects involved on the server-side are shown on figure 2.

6. CONCLUSION

We presented the IMC Protocol Management Framework which enables to flexibly define customized protocol stacks by composing micro-protocols. We described the mechanisms of the TCP/IP layer, which constitutes the network interface of most protocols stacks used

in existing process calculi implementations. We believe that the IMC Protocol Management Framework is suitable to support interaction protocols necessary to implement new distributed process calculi. However, to validate the IMC design, we still have to check if the framework is generic enough to be compatible with existing implementations of some distributed process calculi. We plan to explore this approach in the coming months.

A. THE “KNOCK-KNOCK” CLIENT

```
/***
 * MIKADO Project
 * IMC Framework
 * Communication Protocols Package
 * Adapted from the Jonathan Object Request Broker Distribution (ObjectWeb Consortium)
 * Adaptation: Marc Lacoste
 * Copyright (C) 2003 France Telecom R&D
 */
package org.mikado.imc.protocols.examples;

import java.io.BufferedReader;
import java.io.InputStreamReader;

import org.mikado.imc.protocols.apis.IpSessionIdentifier;
import org.mikado.imc.protocols.apis.Kernel;
import org.mikado.imc.protocols.apis.Marshaller;
import org.mikado.imc.protocols.apis.MarshallerFactory;
import org.mikado.imc.protocols.apis.MikadoProtocolException;
import org.mikado.imc.protocols.apis.Session_High;
import org.mikado.imc.protocols.apis.Session_Low;
import org.mikado.imc.protocols.apis.UnMarshaller;
import org.mikado.imc.protocols.libs.IMCMarshallerFactory;
import org.mikado.imc.protocols.libs.JChunkFactory;
import org.mikado.imc.protocols.libs.JConnectionMgr;
import org.mikado.imc.protocols.libs.JScheduler;
import org.mikado.imc.protocols.libs.TcpIpProtocol;


//****************************************
//* Knock knock client session definition *
//****************************************
class KnockKnockClientSession implements Session_Low {

        // The associated marshaller factory
        static MarshallerFactory marshaller_factory;

        // A buffered reader
        BufferedReader reader;

        KnockKnockClientSession(BufferedReader reader) {
                this.reader = reader;
        }

        public void send(MikadoProtocolException e, Session_High session) {
                System.out.println("Exception received.");
                e.printStackTrace();
        }

        public synchronized void send(
                UnMarshaller unmarshaller,
                Session_High session) {

                try {

                        String fromServer, entree;

                        // Read reply from knock knock server
                        fromServer = unmarshaller.readString8();
                        unmarshaller.close();

                        // End of session
                        if (fromServer.equals("Bye")) {
                                session.close();
                                System.exit(0);
                        } else {
                                System.out.println("Server: " + fromServer);
                                System.out.print("Client: ");
                                System.out.flush();

                                // Read input from user
                                entree = reader.readLine();

                                // Send user input to the server
                                Marshaller marshaller = marshaller_factory.newMarshaller();
                                session.prepare(marshaller);
                                marshaller.writeString8(entree);
                                session.send(marshaller);
                        }
```

```
                } catch (Exception e) {
                        e.printStackTrace();
                        session.close();
                }
        }
}

//*****************************************
// * The client for the knock knock protocol *
//*****************************************
public class KnockKnockClient {

        public static void main(String[] args) {
                try {

                        // Buffered reader for standard input
                        BufferedReader reader =
                                new BufferedReader(new InputStreamReader(System.in), 1);

                        // Ask the user the host to connect to
                        System.out.print("Host ? ");
                        System.out.flush();
                        String host = reader.readLine();

                        // Ask the user on which port number to connect
                        System.out.print("Port number ? ");
                        System.out.flush();
                        int port = Integer.parseInt(reader.readLine());

                        // Initialize kernel
                        Kernel kernel = Kernel.init(null, null);

                        // Initialize scheduler
                        JScheduler scheduler = JScheduler.init(kernel);

                        // Initialize chunk factory
                        JChunkFactory cf = JChunkFactory.init(kernel);

                        // Initialize marshaller factory
                        IMCMarshallerFactory marshaller_factory =
                                new IMCMarshallerFactory(cf);
                        KnockKnockClientSession.marshaller_factory = marshaller_factory;

                        // Instantiate TCP/IP protocol stack
                        TcpIpProtocol protocol =
                                new TcpIpProtocol(
                                        kernel,
                                        new JConnectionMgr(kernel),
                                        scheduler,
                                        cf,
                                        marshaller_factory);

                        // Create TCP/IP Session Identifier
                        IpSessionIdentifier participant =
                                protocol.newSessionIdentifier(host, (short) port);

                        // Bind to remote TCP/IP session
                        Session_High session = participant.bind(new KnockKnockClientSession(reader));

                        // Prepare and send "KNOCK KNOCK" message
                        Marshaller init = marshaller_factory.newMarshaller();
                        session.prepare(init);
                        init.writeByte((byte) 0);
                        session.send(init);

                        // Wait for replies from the server
                        Object lock = new Object();
                        synchronized (lock) {
                                lock.wait();
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

B. THE “KNOCK-KNOCK” SERVER

```
/***
 * MIKADO Project
 * IMC Framework
 * Communication Protocols Package
 * Adapted from the Jonathan Object Request Broker Distribution (ObjectWeb Consortium)
 * Adaptation: Marc Lacoste
 * Copyright (C) 2003 France Telecom R&D
 */
package org.mikado.imc.protocols.examples;

import java.util.Hashtable;
```

```
import org.mikado.imc.protocols.apis.Kernel;
import org.mikado.imc.protocols.apis.Marshaller;
import org.mikado.imc.protocols.apis.MarshallerFactory;
import org.mikado.imc.protocols.apis.MikadoProtocolException;
import org.mikado.imc.protocols.apis.Session_High;
import org.mikado.imc.protocols.apis.Session_Low;
import org.mikado.imc.protocols.apis.UnMarshaller;
import org.mikado.imc.protocols.libs.IMCMarshallerFactory;
import org.mikado.imc.protocols.libs.JChunkFactory;
import org.mikado.imc.protocols.libs.JConnectionMgr;
import org.mikado.imc.protocols.libs.JScheduler;
import org.mikado.imc.protocols.libs.TcpIpProtocol;

//****************************************
//* Knock knock server session definition *
//****************************************
class KnockKnockServerSession {

        private static final int WAITING         = 0;
        private static final int SENTKNOCKKNOCK = 1;
        private static final int SENTCLUE        = 2;
        private static final int ANOTHER         = 3;

        private static final int NUMMESSAGES = 5;

        private int state = WAITING;
        private int currentMsg = 0;
        private boolean finished = false;

        // Protocol Messages : clues
        private static String clues[] =
                { "Will", "Tennis", "Hawaii", "Who", "Who" };

        // Protocol Messages : answers to the clues
        private static String answers[] =
                {
                        "Will you let me in, it's cold out here!",
                        "Tennis-see!",
                        "I'm fine. Hawaii you?",
                        "Is there an owl in here?",
                        "Is there an echo in here?" };

        // The associated protocol
        KnockKnockProtocol protocol;

        // The associated marshaller factory
        static MarshallerFactory marshaller_factory;

        // Builds a new knock knock session and updates the protocol field
        KnockKnockServerSession(KnockKnockProtocol protocol) {
                this.protocol = protocol;
        }

        // Send back a reply to the input message coming from the network
        synchronized void send(UnMarshaller unmarshaller, Session_High sender) {
                try {
                        String theOutput = null;

                        // Read input message and determine reply message
                        if (state == WAITING) {

                                // Read init header
                                unmarshaller.readByte();
                                theOutput = "Knock Knock!";
                                state = SENTKNOCKKNOCK;
                                unmarshaller.close();
                        } else {
                                String theInput = unmarshaller.readString8();
                                //sender.close();
                                System.out.println("Incoming message from client: " + theInput);

                                if (state == SENTKNOCKKNOCK) {
                                        if (theInput.equalsIgnoreCase("Who's There?")) {
                                                theOutput = clues[currentMsg];
                                                state = SENTCLUE;
                                        } else {
                                                theOutput =
                                                        "You were supposed to say \"Who's There?\"!\n"
                                                                + "I would have replied: "
                                                                + clues[currentMsg];
                                                state = SENTCLUE;
                                        }
                                } else if (state == SENTCLUE) {
                                        if (theInput
                                                .equalsIgnoreCase(clues[currentMsg] + " who?")) {
                                                theOutput =
                                                        answers[currentMsg] + " Want another? (y/n)";
                                                state = ANOTHER;
                                        } else {
                                                theOutput =
                                                        "You were supposed to say \""
                                                                + clues[currentMsg]
```

```
                                                + " who?\" !\n"
                                                + "I would have replied "
                                                + answers[currentMsg]
                                                + " !\n"
                                                + "Another try? (y/n)";
                                        state = ANOTHER;
                                }
                        } else if (state == ANOTHER) {
                                if (theInput.equalsIgnoreCase("y")) {
                                        theOutput = "Knock Knock!";
                                        if (currentMsg == (NUMMESSAGES - 1))
                                                currentMsg = 0;
                                        else
                                                currentMsg++;
                                        state = SENTKNOCKKNOCK;
                                } else {
                                        theOutput = "Bye";
                                        finished = true;
                                        state = WAITING;
                                }
                        }
                        unmarshaller.close();
                }

                // Prepare reply message
                Marshaller marshaller = marshaller_factory.newMarshaller();
                sender.prepare(marshaller);

                // Send reply message on the network
                marshaller.writeString8(theOutput);
                System.out.println("Sending message to client: " + theOutput);
                sender.send(marshaller);
                if (finished) {
                        protocol.unregister(sender);
                        sender.close();
                        finished = false;
                }
        } catch (MikadoProtocolException e) {
                System.out.println("EXCEPTION");
                e.printStackTrace();
                protocol.unregister(sender);
                sender.close();
        }
    }
}

//************************************
// * Knock knock protocol definition *
//************************************
class KnockKnockProtocol implements Session_Low {

    // A pool of session objects.
    Hashtable kk_sessions;

    // Instantiate a new pool of session object.
    KnockKnockProtocol() {
        kk_sessions = new Hashtable();
    }

    public void send(MikadoProtocolException e, Session_High session) {
        //      System.out.println("Exception received.");
        //      e.printStackTrace();
    }

    //      Send back a reply to the input message coming from the network
    public void send(UnMarshaller message, Session_High sender) {
        KnockKnockServerSession session = null;

        // Determine the reply session to use
        synchronized (this) {
            session = (KnockKnockServerSession) kk_sessions.get(sender);
            if (session == null) {
                session = new KnockKnockServerSession(this);
                kk_sessions.put(sender, session);
            }
        }
        // Send the reply message over the network
        session.send(message, sender);
    }

    // Remove a session from a pool of session objects
    void unregister(Session_High sender) {
        kk_sessions.remove(sender);
    }
}

//*****************************************
// * The server for the knock knock protocol *
//*****************************************
public class KnockKnockServer {

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

```
                try {
                        // Initialize kernel
                        Kernel kernel = Kernel.init(null, null);

                        // Initialize scheduler
                        JScheduler scheduler = JScheduler.init(kernel);

                        // Initialize chunk factory
                        JChunkFactory cf = JChunkFactory.init(kernel);

                        // Initialize marshaller factory
                        MarshallerFactory marshaller_factory = new IMCMarshallerFactory(cf);
                        KnockKnockServerSession.marshaller_factory = marshaller_factory;

                        // Instantiate TCP/IP protocol stack
                        TcpIpProtocol protocol =
                                new TcpIpProtocol(
                                        kernel,
                                        new JConnectionMgr(kernel),
                                        scheduler,
                                        cf,
                                        marshaller_factory);

                        // Create protocol stack KnockKnock/TCP/IP
                        protocol.newProtocolGraph().export(new KnockKnockProtocol());

                        // Wait for invocations
                        Object lock = new Object();
                        synchronized (lock) {
                                lock.wait();
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

REFERENCES

BETTINI, L., DE NICOLA, R., LACOSTE, M., LOPES, L., AND VASCONCELOS, V. 2003. Virtual Machine Technologies : Core Software Framework v0 – towards MIKADO Abstract Machines. Tech. Rep. D3.1.0, IST MIKADO Project.

BETTINI, L., DE NICOLA, R., AND PUGLIESE, R. 2002. Klava : a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience 32,* 14, 1365–1394.

BOREALE, M., DE NICOLA, R., LACOSTE, M., AND VASCONCELOS, V. 2002a. Analysis of Distribution Structures : State of the Art. MIKADO Global Computing Project Deliverable D3.1.1.

BOREALE, M., DE NICOLA, R., LACOSTE, M., AND VASCONCELOS, V. 2002b. Analysis of Distribution Structures : State of the art. Tech. Rep. D3.1.1, IST MIKADO Project.

BOUDOL, G., CASTELLANI, I., GERMAIN, F., AND LACOSTE, M. 2002a. Models of Distribution and Mobility : State of the Art. MIKADO Global Computing Project Deliverable D1.1.1.

BOUDOL, G., CASTELLANI, I., GERMAIN, F., AND LACOSTE, M. 2002b. Models of Distribution and Mobility : State of the art. Tech. Rep. D1.1.1, IST MIKADO Project.

DUMANT, B., HORN, F., TRAN, F. D., AND STEFANI, J.-B. 1999. Jonathan : an Open Distributed Processing Environment in Java . *Distributed Systems Engineering 6,* 1, 3–12. Also in *Proc. MIDDLEWARE'98*.

FLANAGAN, D. 1999. *Java in a Nutshell, a Desktop Quick Reference*, 3rd Edition ed. O'Reilly.

FOURNET, C. AND MARANGET, L. 1997. The Join-Calculus Language. Software and documentation available from `http://join.inria.fr/`.

HUTCHINSON, N. AND PETERSON, L. 1991. The x-Kernel : An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering 17,* 1, 64–76.

INTERNATIONAL STANDARDS ORGANIZATION. 1995. ODP Reference Model : Foundations. Tech. Rep. ISO/IEC JTC1/SC21/WG7 10746-2.

KRAKOWIAK, S. 2002. *The Jonathan Tutorial : Overview, Binding, Communication, Configuration and Resource Frameworks*. ObjectWeb Consortium.

LE FESSANT, F. 1998. The JoCaml System Prototype. Software and documentation available from `http://join.inria.fr/jocaml`.

LEVI, F. AND SANGIORGI, D. 2000. Controlling Interference in Ambients. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL'00)*. ACM Press, Boston, Massachusetts (USA), 352–364.

LOPES, L. 1999. On the Design and Implementation of a Virtual Machine for Process Calculi. Ph.D. thesis, University of Porto.

MIKADO PROJECT. 2003. See `http://mikado.di.fc.ul.pt/`.

OBJECTWEB CONSORTIUM. 2002. Jonathan : an Open Distributed Objects Platform. See `http://jonathan.objectweb.org/`.

SANGIORGI, D. AND VALENTE, A. 2001. A Distributed Abstract Machine for Safe Ambients. In *Proc. 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*. Lecture Notes in Computer Science, vol. 2076. Springer-Verlag, Heraklion, Greece, 408–420.