Building Distributed and Mobile Applications with IMC

Lorenzo Bettini

Dipartimento di Sistemi e Informatica
Università di Firenze

Dipartimento di Informatica
Università di Torino

http://www.lorenzobettini.it

Miniscuola WOA, Sept. 2007

Implementing Distributed Applications & Code Mobility

- Java provides useful means for implementing distributed applications:

  and code mobility:

Implementing Distributed Applications & Code Mobility

- Java provides useful means for implementing distributed applications:
  - Java network library
  - language synchronization features

  and code mobility:
  - object serialization
  - dynamic class loading

Implementing Distributed Applications & Code Mobility

- Java provides useful means for implementing distributed applications:
  - Java network library
  - language synchronization features

  and code mobility:
  - object serialization
  - dynamic class loading
- These mechanisms are low-level
  - Most Java-based distributed and mobile systems re-implement from scratch components for distribution and mobility

IMC - Implementing Mobile Calculi

- Is a middleware/framework for implementing distributed and mobile code run-time systems
- Aims at simplifying the implementation of distributed mobile code applications:
  - is based on recurrent standard mechanisms and patterns
  - permits concentrating on the features that are specific of a particular language
  - can be easily extended/customized to fit language-specific requirements
- Provides components for
  - Network topology
  - Communication protocols
  - Code mobility
- We have used IMC to implement run-time systems for some mobile and distributed calculi (e.g., JDPI, KLAVA, etc.).

The MIKADO project

The IMC framework is being built within an European project (EU-FET) on *Global Computing*

Calculi for Mobility

- The main intent is to investigate new mobile code calculi linguistic features:
  - The meta theory
  - Provide prototype implementations

One of the tasks of the project was to build a framework for developing the run-time systems for mobile code languages.

Protocols

- Primitives for implementing specific protocols
  - low level protocols (protocol layers)
  - high level protocols (protocol states)
- Build a protocol starting from small components
- Make the components re-usable:
  - the components are abstract and independent from specific communication layers.

Network Topology

- Primitives for connection and disconnection (both physical and logical)
- Node creation and deletion
- Keeps track of the topology of the network
  - flat
  - hierarchical

Mobility

- Make code mobility transparent to the programmer
- All issues are dealt with by the package:
  - code collecting, marshalling
  - code dispatch
  - dynamic loading of code received from a remote site
- Provide abstract interfaces and implementations for Java byte-code mobility

Network protocols...

- Each *network protocol* is viewed as an aggregation of *protocol states*:
  - a high-level communication protocol is described as a state automaton
- Each protocol state is implemented:
  - by extending the `ProtocolState` abstract class
  - by providing the implementation for the method `enter`, which returns the next state to execute.
- The `Protocol` class:
  - aggregates the protocol states
  - provides a *template method* (`start`) that will execute each state at a time

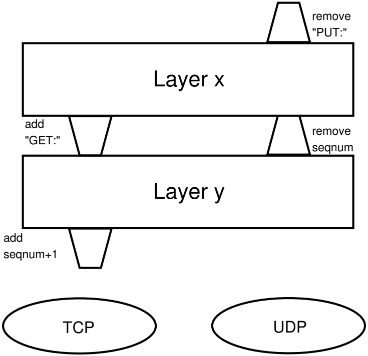
Marshalers and UnMarshalers

- Specialized streams: `Marshaler` and an `UnMarshaler` to write/read from the actual communication layer.
- They provide means to write/read any primitive data type (inherited from `DataOutput` and `DataInput`).
- They deal with code mobility (relying on the `mobility` sub-package).

```
public interface Marshaler extends
        DataOutput, Closeable, Flushable, MigratingCodeHandler {
    void writeStringLine(String s) throws IOException;
    void writeReference(Serializable o) throws IOException;
    void writeMigratingCode(MigratingCode code) throws IOException,
                                        MigrationUnsupported;
    void writeMigratingPacket(MigratingPacket packet) throws IOException;
    ...
}
```

Protocol layers

- The data in the streams can be "pre-processed" by some customized *protocol layers*
- Each protocol layer takes care of the shape of messages (*low-level* protocol)
- These layers are then composed into a `ProtocolStack` that ensures the order of preprocessing passing through all the layers in the stack.



Using a ProtocolStack

Writing

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

Using a ProtocolStack

Writing

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

Reading

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

How layers work

- When a marshaler is created, a new “communication” is started (using the underlying session);
- The headers (of the low-level protocol layers) are created;
- The contents written using the marshaler can be buffered (e.g., for UDP, into an UDP packet);
- When the marshaler is released the buffer is actually flushed (e.g., for UDP, the packet is actually sent).

A customized layer

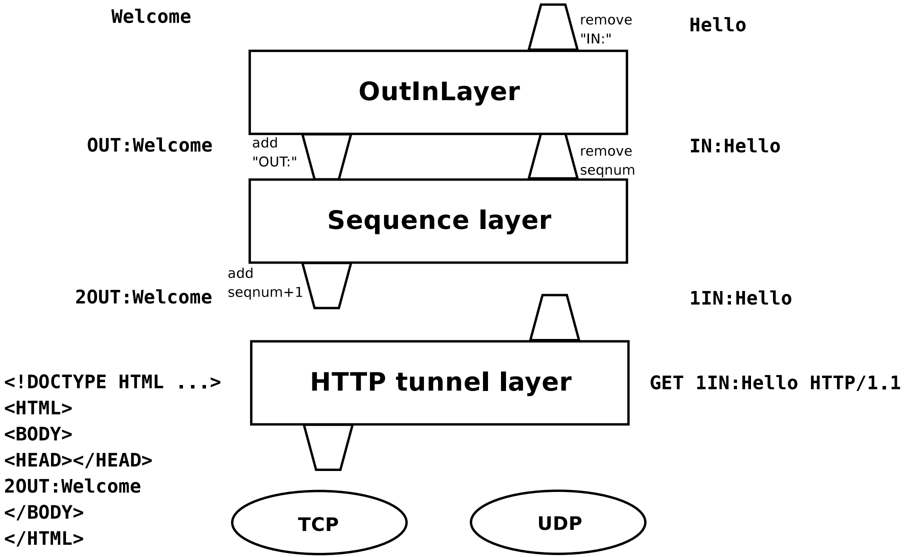
The following specialized protocol layer removes a header from the input that consists of the line `"IN"`; it also adds a header to the output that consists of the line `"OUT"`:

```
public class OutInLayer extends ProtocolLayer {
    protected Marshaler doCreateMarshaler(Marshaler marshaler)
        throws ProtocolException {
        marshaler.writeStringLine("OUT");
        return marshaler;
    }

    protected UnMarshaler doCreateUnMarshaler(UnMarshaler unMarshaler)
        throws ProtocolException {
        String header = unMarshaler.readStringLine();
        if (!header.equals("IN"))
            throw new ProtocolException("wrong header: " + header);
        return unMarshaler;
    }
}
```

Tunneling

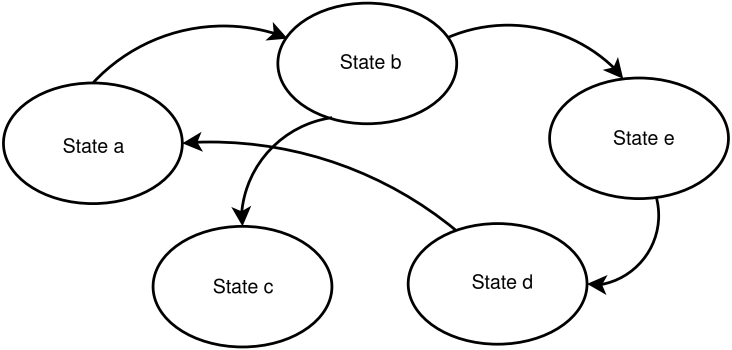
A specialized subclass, `TunnelProtocolLayer` is provided that permits to "tunnel" a protocol layer into another (high level) protocol: e.g., encapsulate a message into HTTP requests and responses.

Tunneling


Welcome
remove
"IN:"
Hello
OutInLayer
OUT:Welcome
add
"OUT:"
remove
seqnum
IN:Hello
Sequence layer
add
seqnum+1
2OUT:Welcome
1IN:Hello
<!DOCTYPE HTML ...>
<HTML>
<BODY>
<HEAD></HEAD>
2OUT:Welcome
</BODY>
</HTML>
HTTP tunnel layer
GET 1IN:Hello HTTP/1.1
TCP
UDP

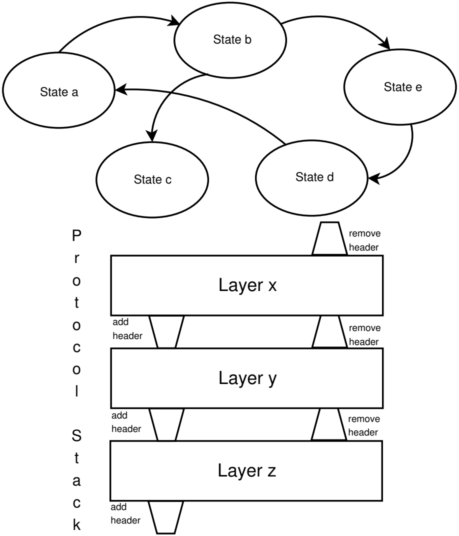
Protocol States

A high level protocol can be described as a state automaton.
The `protocols` package provides features to implement protocol states and compose them in an automaton.

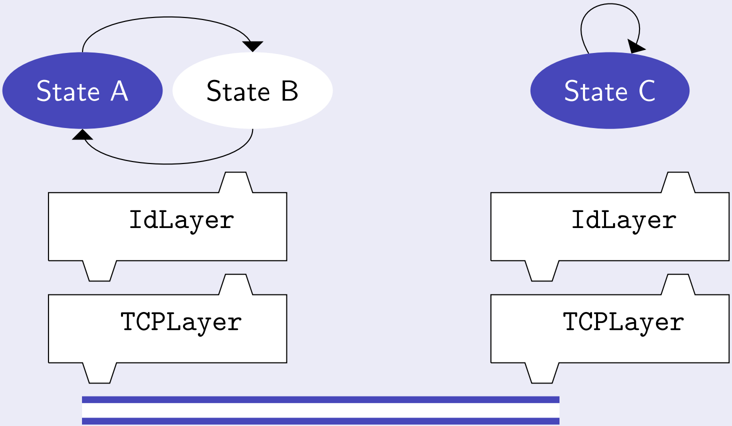


The class `Protocol`

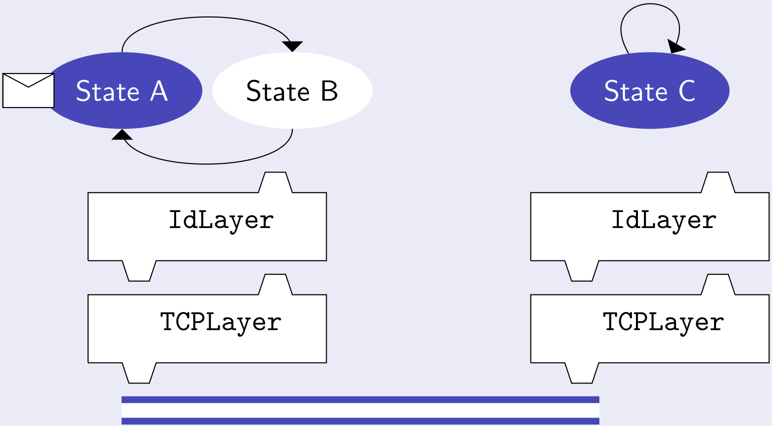
A collection of protocol states and a reference to a protocol stack:



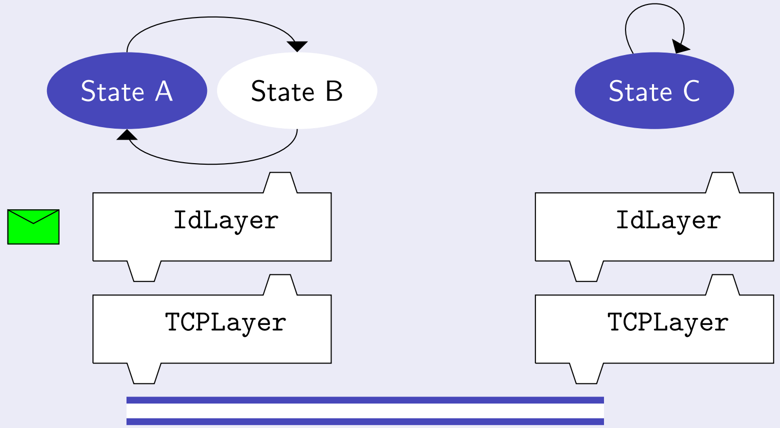
Communication Protocols: an example



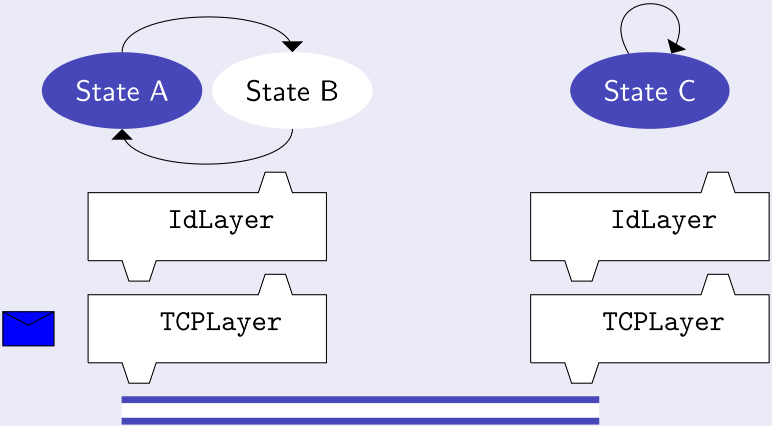
Communication Protocols: an example



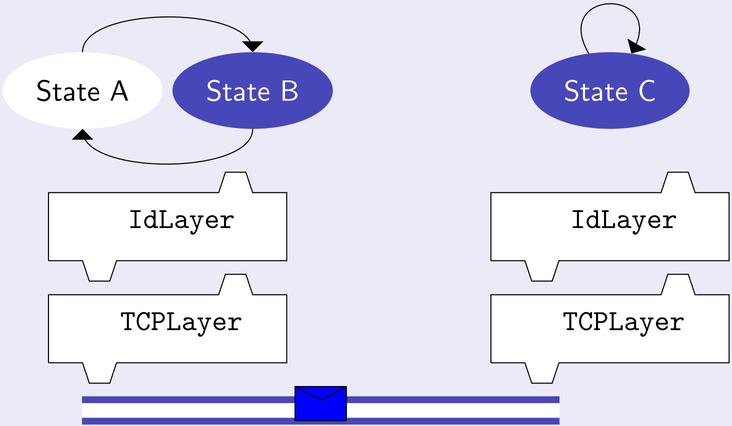
Communication Protocols: an example



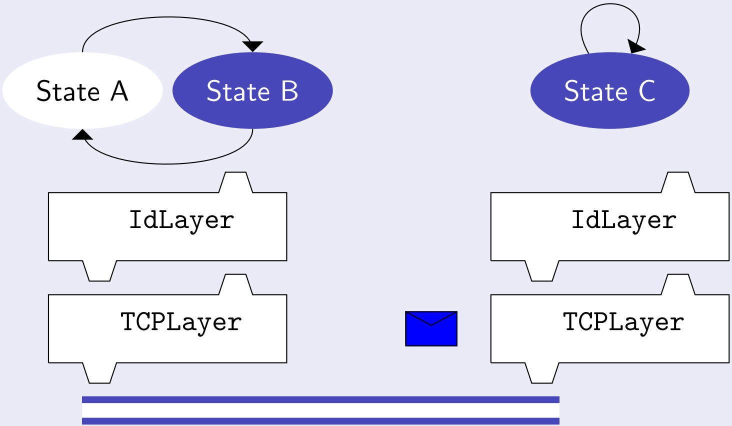
Communication Protocols: an example



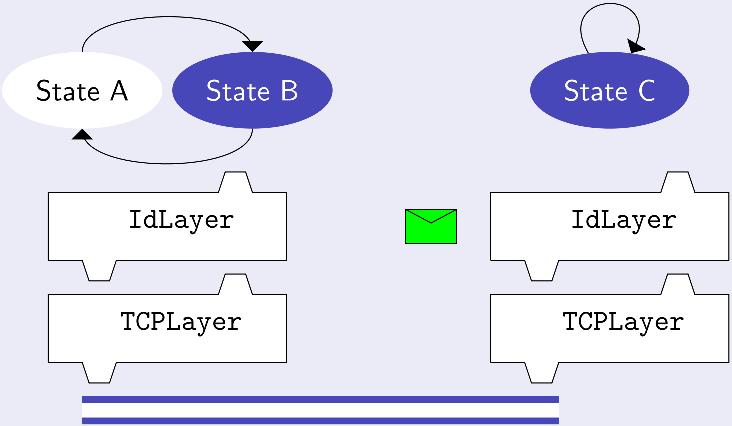
Communication Protocols: an example



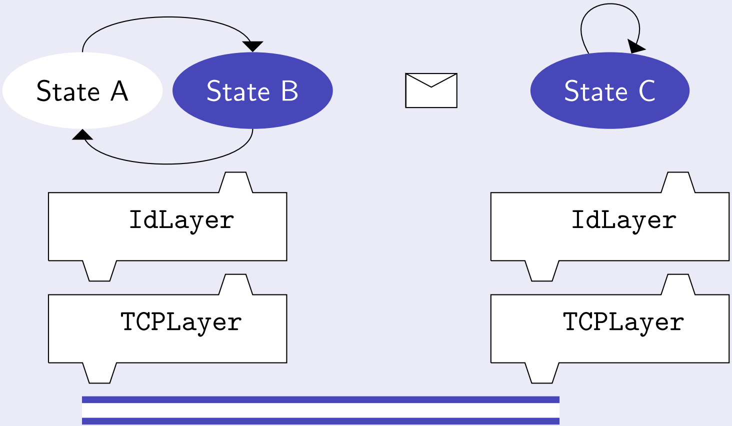
Communication Protocols: an example



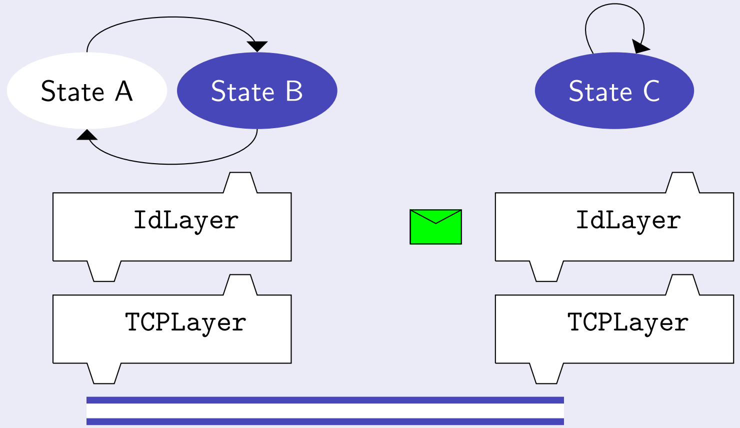
Communication Protocols: an example



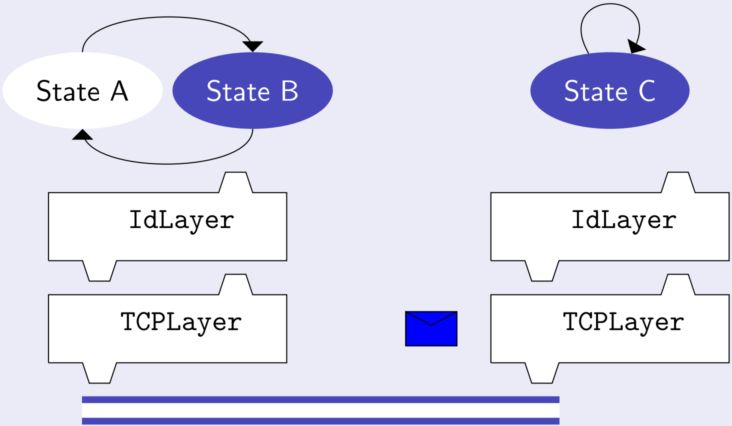
Communication Protocols: an example



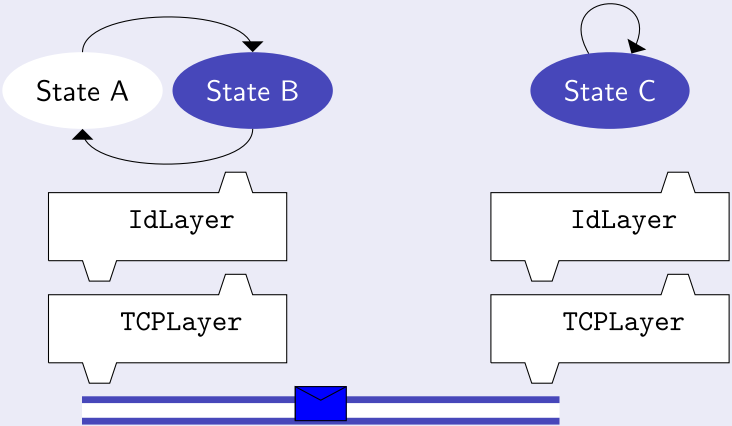
Communication Protocols: an example



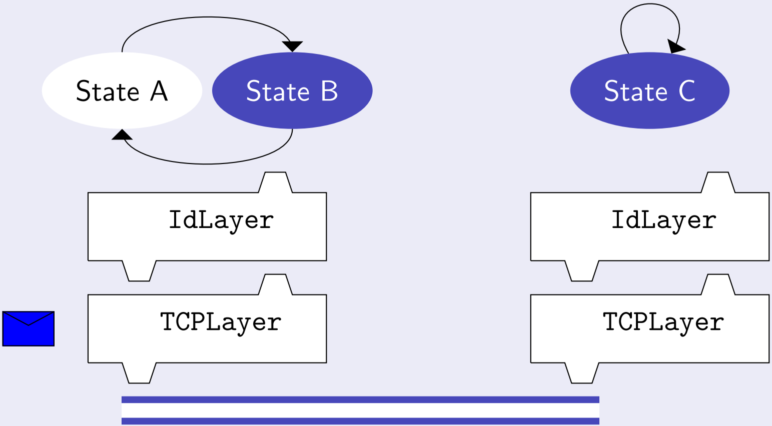
Communication Protocols: an example



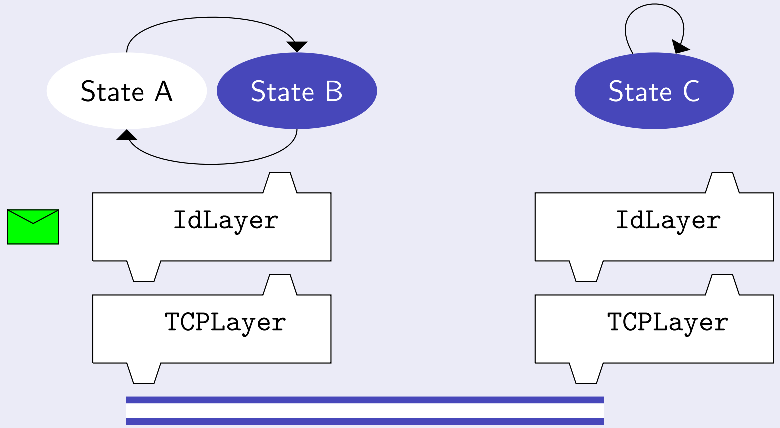
Communication Protocols: an example



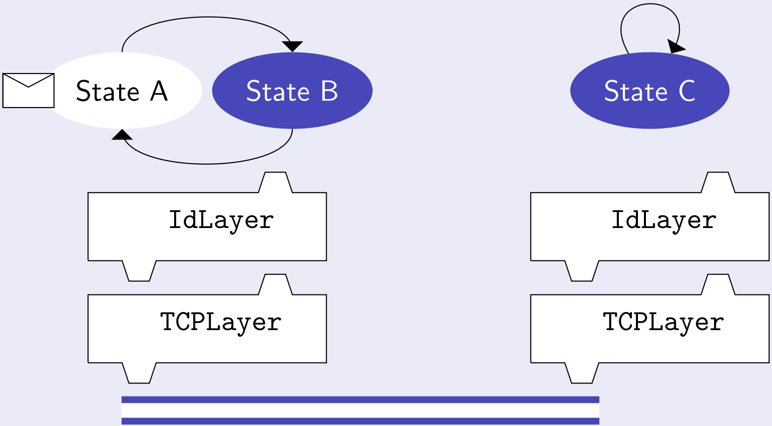
Communication Protocols: an example



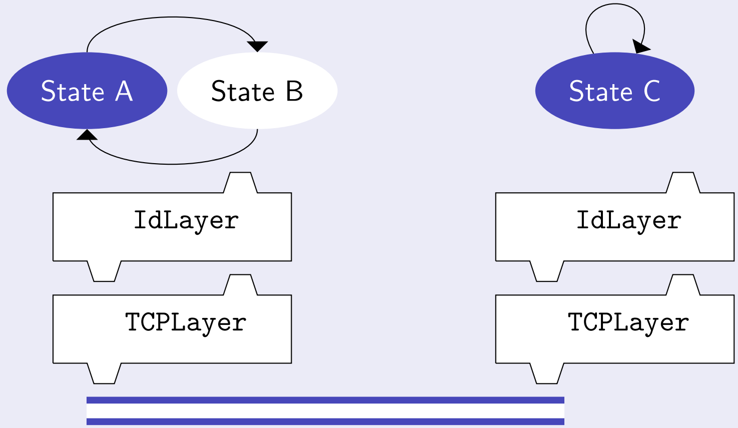
Communication Protocols: an example



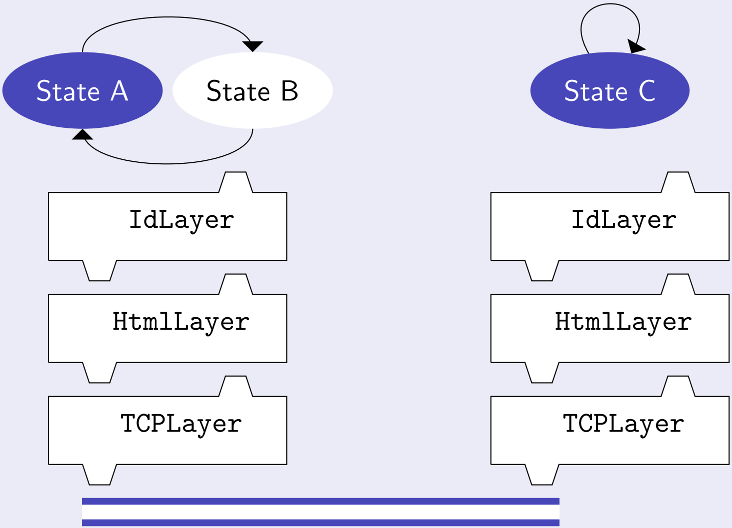
Communication Protocols: an example



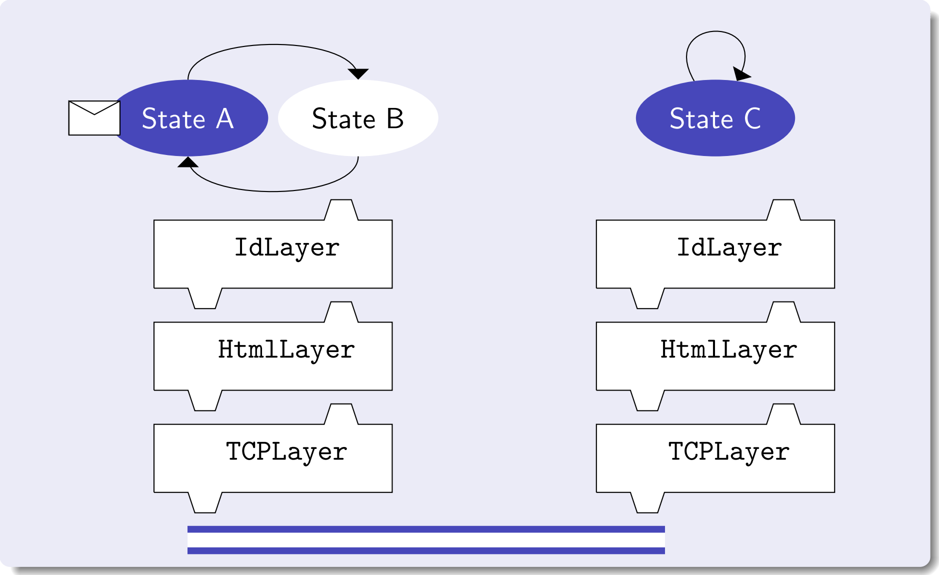
Communication Protocols: an example



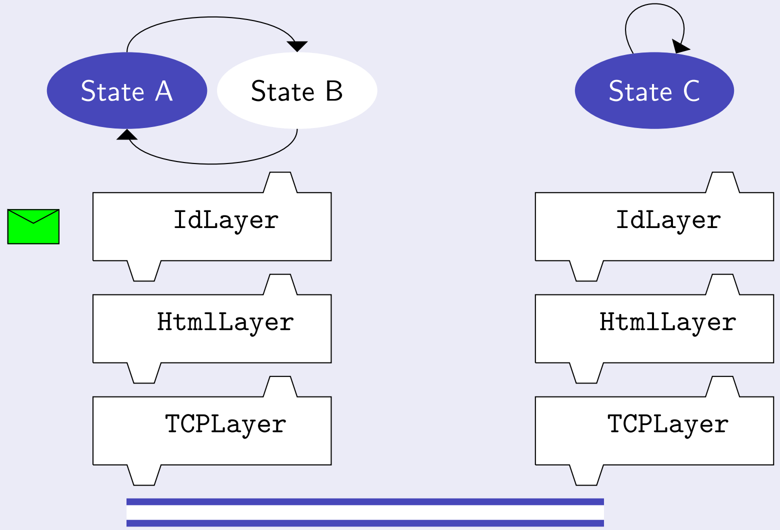
Communication Protocols: an example



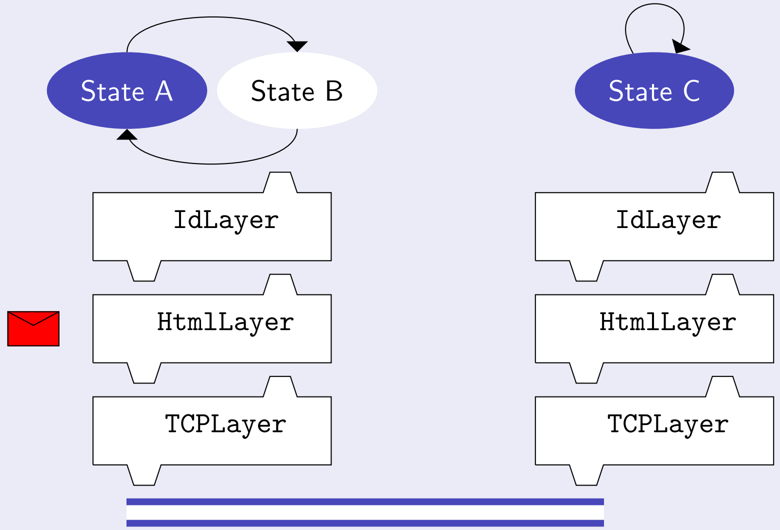
Communication Protocols: an example



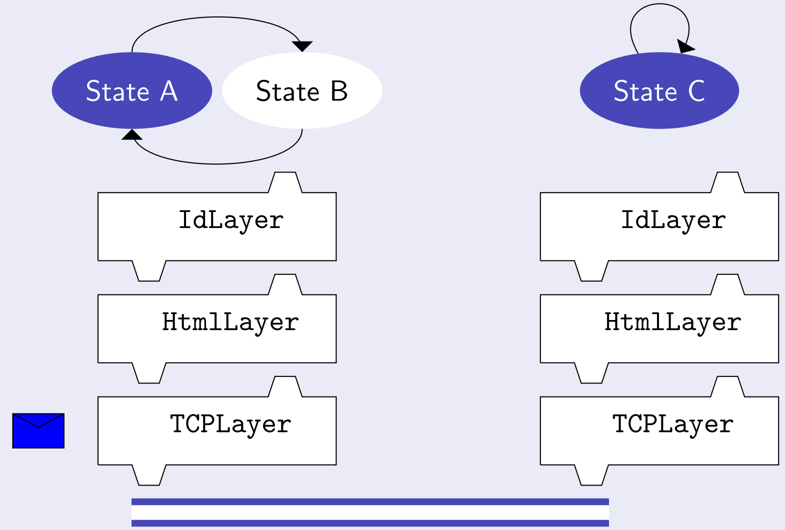
Communication Protocols: an example



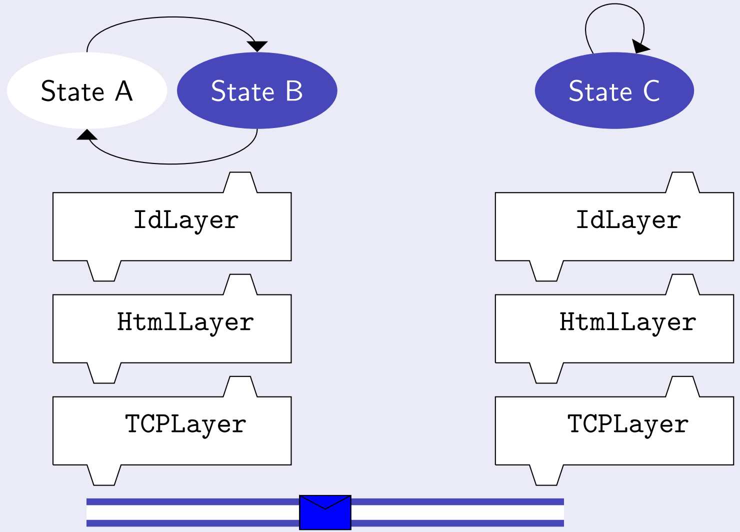
Communication Protocols: an example



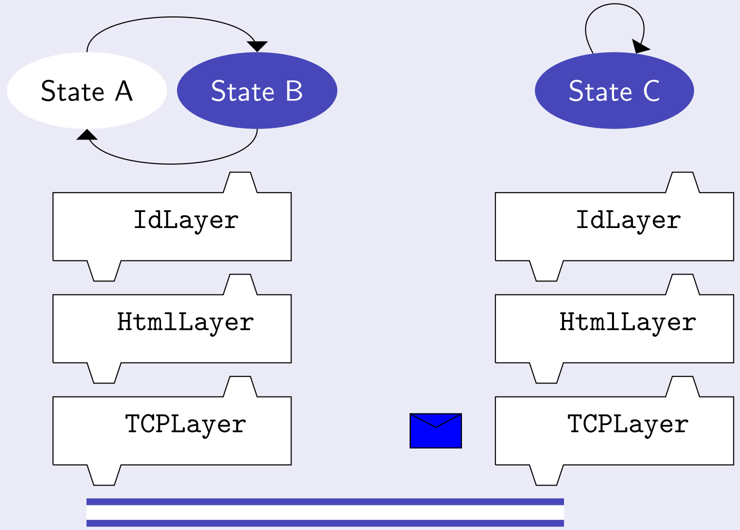
Communication Protocols: an example



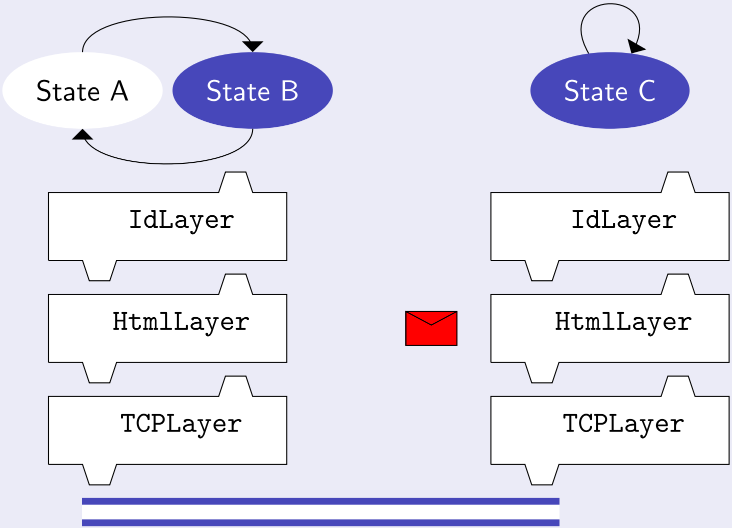
Communication Protocols: an example



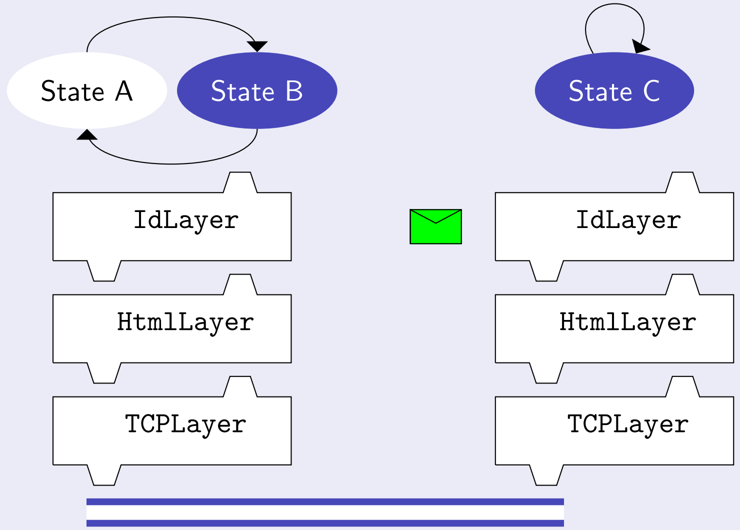
Communication Protocols: an example



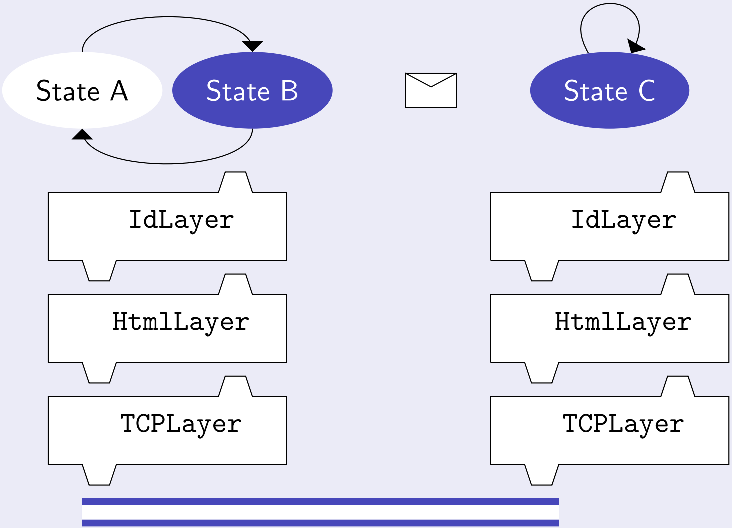
Communication Protocols: an example



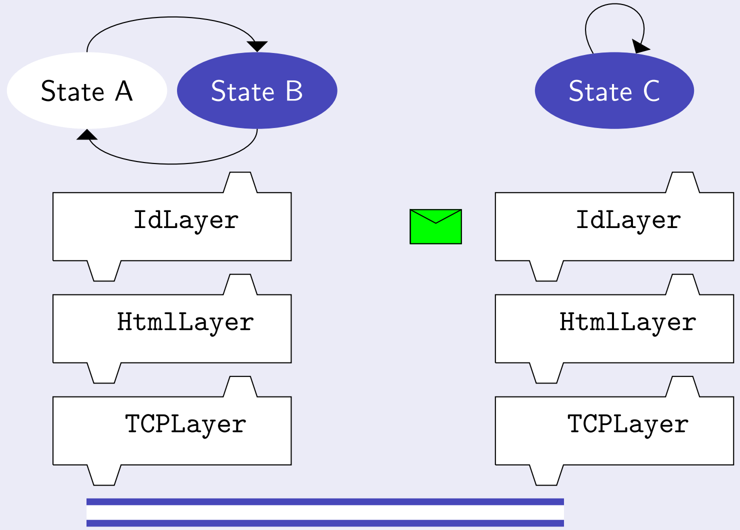
Communication Protocols: an example



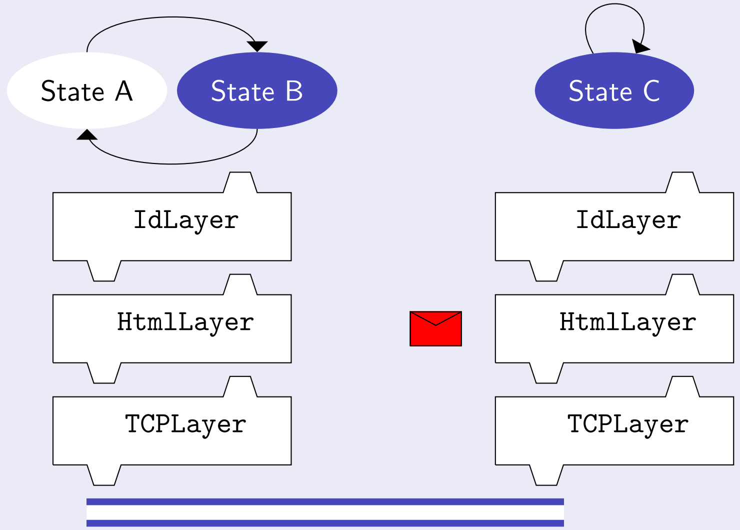
Communication Protocols: an example



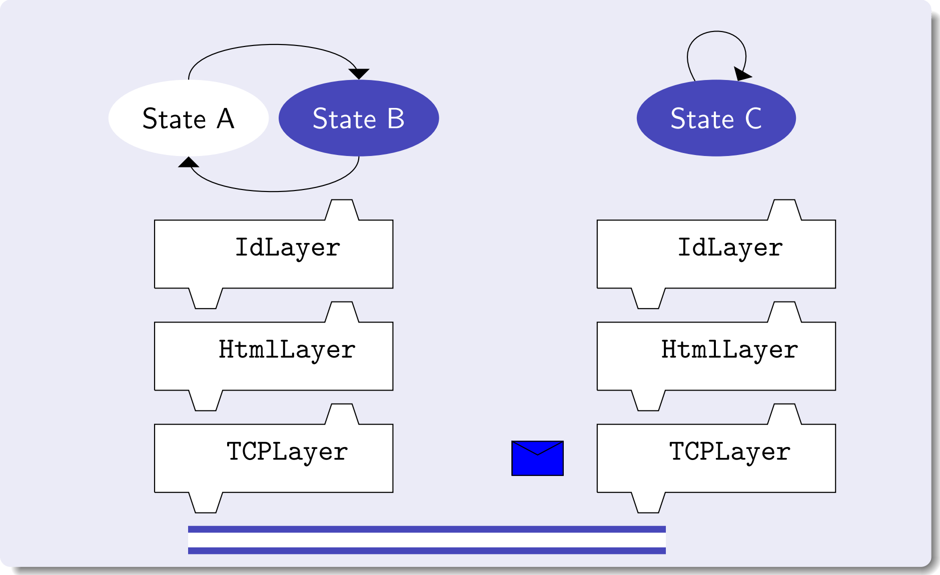
Communication Protocols: an example



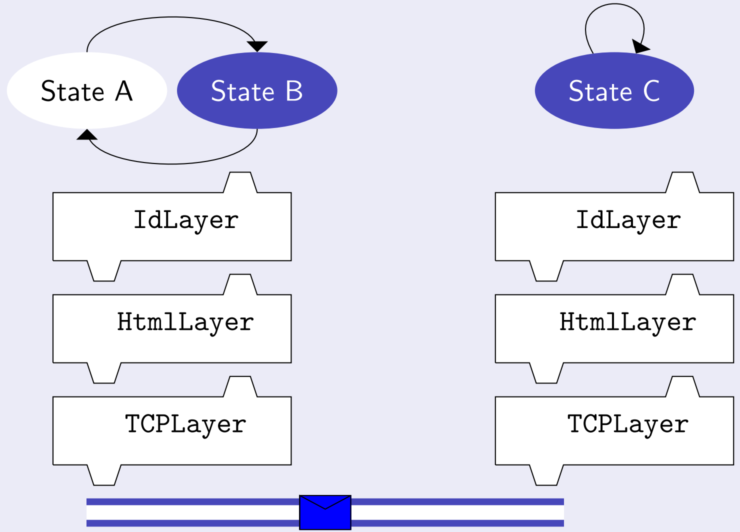
Communication Protocols: an example



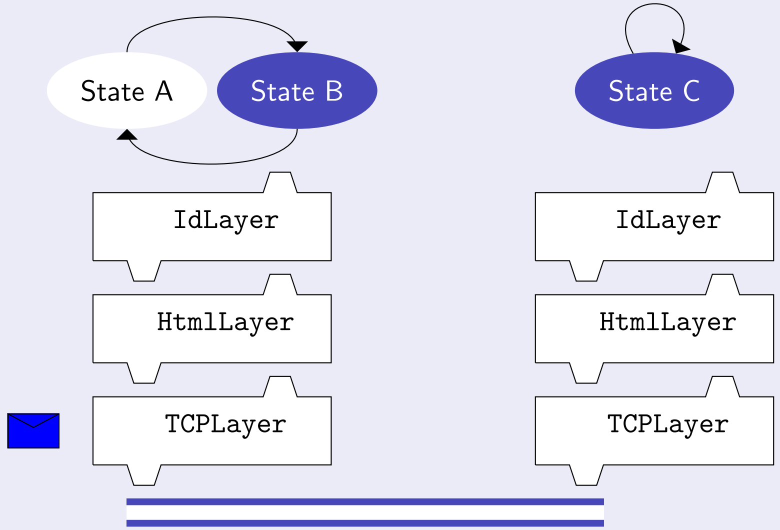
Communication Protocols: an example



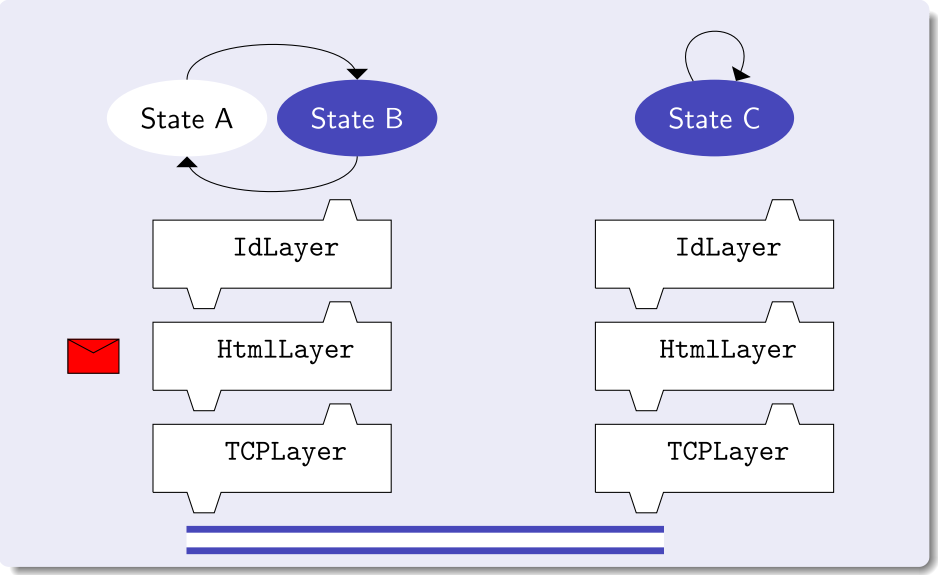
Communication Protocols: an example



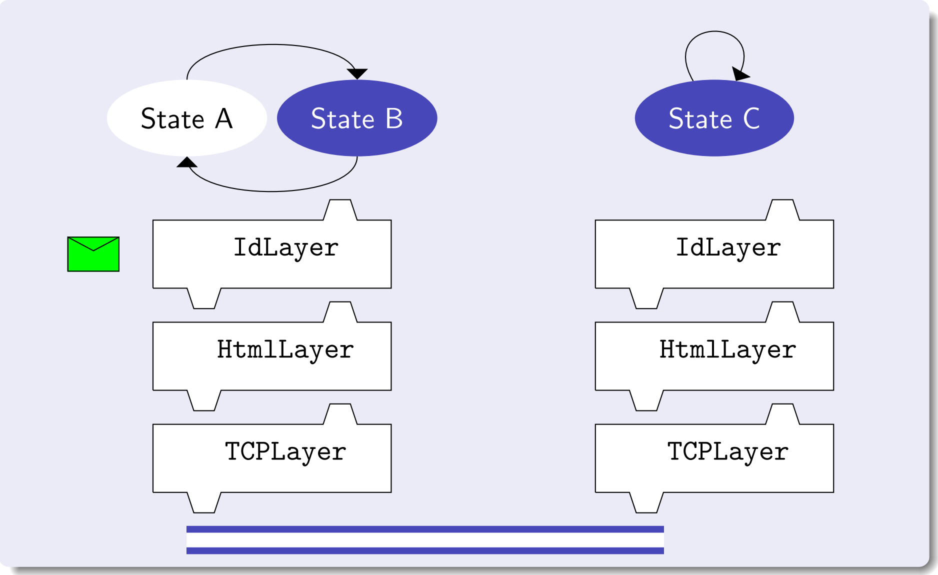
Communication Protocols: an example



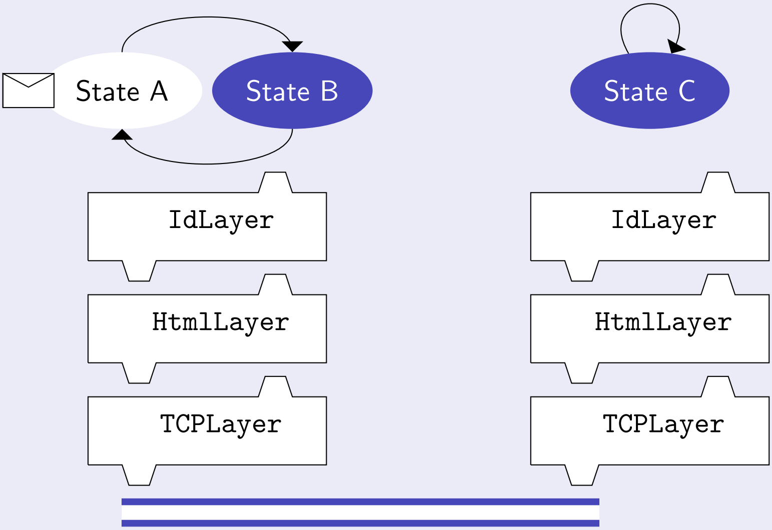
Communication Protocols: an example



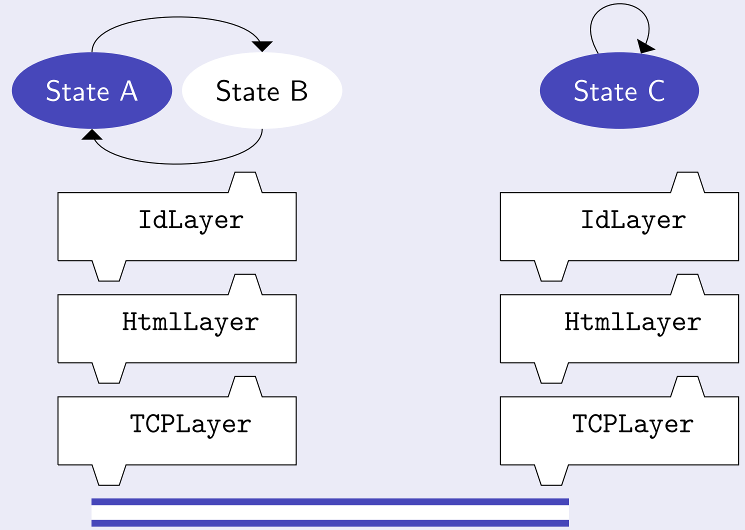
Communication Protocols: an example



Communication Protocols: an example



Communication Protocols: an example



The class ProtocolState

Derive from the class `ProtocolState`; implement `enter()`

```
public class EchoProtocolState extends ProtocolStateSimple {
    public void enter(Object param, TransmissionChannel transmissionChannel)
            throws ProtocolException {
        UnMarshaler unMarshaler = createUnMarshaler();
        String line = unMarshaler.readStringLine();
        releaseUnMarshaler(unMarshaler);
        Marshaler marshaler = createMarshaler();
        marshaler.writeStringLine(line);
        releaseMarshaler(marshaler);
    }
}
```

The class ProtocolState

Derive from the class ProtocolState; implement enter()

```
public class EchoProtocolState extends ProtocolStateSimple {
    public void enter(Object param, TransmissionChannel transmissionChannel)
            throws ProtocolException {
        UnMarshaler unMarshaler = createUnMarshaler();
        String line = unMarshaler.readStringLine();
        releaseUnMarshaler(unMarshaler);
        Marshaler marshaler = createMarshaler();
        marshaler.writeStringLine(line);
        releaseMarshaler(marshaler);
    }
}
```

getUnMarshaler

```
if (transmissionChannel == null || transmissionChannel.unMarshaler == null) {
    return createUnMarshaler();
else
    return transmissionChannel.unMarshaler;
```

Switch state

Avoid writing long switch statements

ProtocolSwitchState

```
protocolSwitchState.addRequestState("WRITE", new WriteState(Protocol.START));
protocolSwitchState.addRequestState("READ", new ReadState(Protocol.START));
protocolSwitchState.addRequestState("REMOVE", new ReadState(Protocol.START));
protocolSwitchState.addRequestState("QUIT", Protocol.END);
Protocol protocol = new Protocol();
protocol.setState(Protocol.START, protocolSwitchState);
```

- Each protocol state is parameterized with the next state in the automaton
- `START` and `END` are the identifiers of the special start and end states

Connectivity features

Provide basic connectivity classes:

- `Node`:
  - container of executing processes (`NodeProcess`)
  - provides features to receive and establish a session, e.g.: `accept`, `connect`
- `SessionManager`: keeps trace of all the established sessions

Connectivity features

Provide basic connectivity classes:

- `Node`:
  - container of executing processes (`NodeProcess`)
  - provides features to receive and establish a session, e.g.: `accept`, `connect`
- `SessionManager`: keeps trace of all the established sessions

Session identifiers

Sessions are logical connections and independent from the low level communication layer. Automatically chosen according to `SessionId`

- Stream connections for TCP sockets (`tcp-mysite.com:9999`)
- Logical connections if UDP packets are used (`udp-mysite.com:9999`)
- Local pipes (`pipe-foobar`)

Session Starter

SessionStarter is an abstract class for establishing a session (both client and server side):

```
public abstract class SessionStarter {
    /** Accepts an incoming session. */
    public abstract Session accept() throws ProtocolException;

    /** Establishes a session. */
    public abstract Session connect() throws ProtocolException;

    /** Closes this starter, but not sessions created through this starter. */
    protected abstract void doClose() throws ProtocolException;
}
```

Specific session starters should be provided for specific low-level communication protocols; the framework provides `TcpSessionStarter`, `UdpSessionStarter` and `LocalSessionStarter` (that uses local pipes, useful for testing).

Example: creating a TCP session, server-side

```
int port = 9999;
ProtocolStack protocolStack = new ProtocolStack();
// possibly customize stack with additional layers
System.out.println("accepting connections on port " + port);
SessionStarter sessionStarter = new TcpSessionStarter(new IpSessionId(port));
Session session = protocolStack.accept(sessionStarter);
System.out.println("established session " + session);
sessionStarter.close();
// no more accepting sessions, but the established session is still up.
UnMarshaler unMarshaler = protocolStack.createUnMarshaler();
while (true) {
  System.out.println("read line: " + unMarshaler.readStringLine());
}
```

The client side will be similar (but it will use `connect` instead of `accept`).

Abstracting from session starters

- IMCSessionStarterTable: associates a `SessionStarter` class to a specific `SessionId` identifier
  - e.g., `"tcp"` → `TcpSessionStarter`
- We can then abstract from a specific session type
- method createSessionStarter(SessionId) returns the `SessionStarter` associated to a specific session type.

Abstracting from session starters

- IMCSessionStarterTable: associates a `SessionStarter` class to a specific `SessionId` identifier
  - e.g., `"tcp"` → `TcpSessionStarter`
- We can then abstract from a specific session type
- method createSessionStarter(SessionId) returns the `SessionStarter` associated to a specific session type.

Abstraction

Switching from a session type to another is just a matter of changing the session identifier.

Binding protocols and sessions

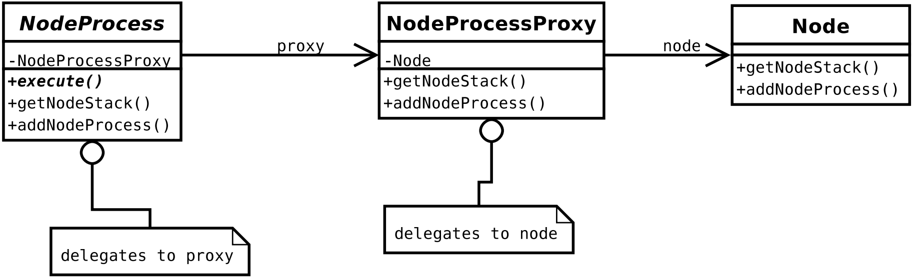
```
public class GenericServer {
    public GenericServer(String host, Protocol protocol)
            throws ProtocolException, IOException {
        SessionStarterTable sessionStarterTable = new IMCSessionStarterTable();
        SessionId sessionId = SessionId.parseSessionId(host);
        System.out.println("accepting session " + sessionId + " ...");
        ProtocolStack protocolStack = new ProtocolStack();
        Session session =
            protocolStack.accept(sessionStarterTable.createSessionStarter(sessionId));
        System.out.println("established session " + session);
        System.out.println("starting protocol... ");
        protocol.setProtocolStack(protocolStack);
        protocol.start();
        protocol.close();
        System.out.println("protocol terminated");
    }
}
```

Nodes & Processes

- A participant to a network is an instance of class Node.
- A node is a container of running processes (class NodeProcess).
  - The programmer must inherit from `NodeProcess`
  - and provide the implementation for method execute
- A process can access the resources contained in a node and migrate to other nodes.
- A node keeps track of all the processes currently in execution.

Processes & proxies

- Processes delegate most of their methods to the node they are running in.
- However, they do not have access to the node itself (to avoid security problems).
- This is achieved by using a proxy: the processes delegate to the proxy and not directly to the node.



A process example

A process running a protocol

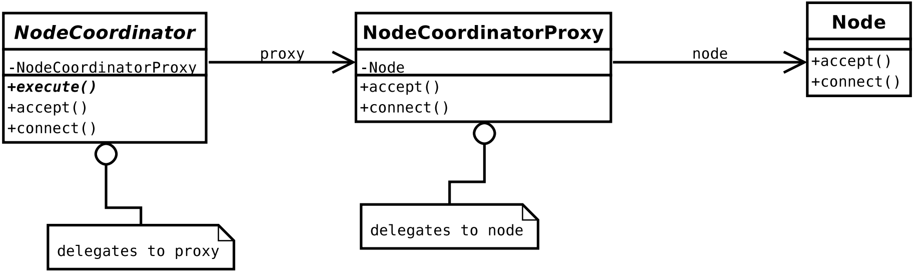
```
public class ProtocolThread extends NodeProcess {
    /** The protocol this thread will execute. */
    protected Protocol protocol;

    public void execute() throws IMCException {
        try {
            protocol.start();
        } finally {
            close();
        }
    }

    public void close() throws IMCException {
        super.close();
        protocol.close();
    }
}
```

Node coordinators

Node coordinators are super user processes:

- Privileged processes
- can execute connection and disconnection actions
- cannot migrate

Standard processes cannot execute privileged actions.



A node coordinator example

Accepting connections

```
public class MyCoordinator extends NodeCoordinator {
    SessionId sessionId;

    public void execute() throws IMCException {
        Protocol protocol = new MyProtocol();
        accept(sessionId, protocol);
        protocol.start();
    }

    public static void main(String args[]) {
        Node node = new Node();
        node.addNodeCoordinator(new MyCoordinator());
    }
}
```

Another node coordinator example

Accepting connections (multi-threaded)

```
public class AcceptNodeCoordinator extends NodeCoordinator {
    private ProtocolFactory protocolFactory;
    private SessionId sessionId;
    private SessionStarter sessionStarter;

    public void execute() throws IMCException {
        if (sessionStarter == null)
            sessionStarter = createSessionStarter(sessionId, null);

        while (true) {
            Protocol protocol = protocolFactory.createProtocol();
            addNodeProcess(
                new ProtocolThread(accept(sessionStarter, protocol)));
        }
    }
}
```

Goals of the package `mobility`

- Package `org.mikado.imc.mobility`
- Make code mobility transparent to the programmer
- All issues are dealt with by the package:
  - code collecting, marshalling
  - code dispatch
  - dynamic loading of code received from a remote site
- Provide abstract interfaces and implementations for Java byte-code mobility
- Used internally by the IMC `protocols` package to exchange migrating objects, but can also be used as a stand-alone package

Moving code

Two possible approaches:

- Automatic: the classes needed by the migrating code are collected and delivered together with that code;
- On demand: when some classes are required by code migrated to a remote site, it is requested to the server that sent the code.

We choose automatic approach

Advantage

Comply with mobile agent paradigm:

- the agent is autonomous when migrating
- disconnected operations
- the originating computer does not need be connected after migration

We choose automatic approach

Advantage

Comply with mobile agent paradigm:

- the agent is autonomous when migrating
- disconnected operations
- the originating computer does not need be connected after migration

Drawback

Code that may be never used is dispatched

Kinds of mobility

Three kinds of mobility have been identified in the literature:

- *weak mobility*: the dynamic linking of code arriving from a different site;

Kinds of mobility

Three kinds of mobility have been identified in the literature:

- *weak mobility*: the dynamic linking of code arriving from a different site;
- *strong mobility*: the movement of the code and of the execution state of a thread to a different site and the resumption of its execution on arrival;

Kinds of mobility

Three kinds of mobility have been identified in the literature:

- *weak mobility*: the dynamic linking of code arriving from a different site;
- *strong mobility*: the movement of the code and of the execution state of a thread to a different site and the resumption of its execution on arrival;
- *full mobility*: the movement of the whole state of the running program including all threads' stacks, namespaces and other resources. This is a generalization of strong mobility that makes the migration completely transparent.

Code mobility in Java (and in IMC)

Unfortunately Java only provides *weak mobility*, since threads' execution state (stack and program counter) cannot be saved and restored

Code mobility in Java (and in IMC)

Unfortunately Java only provides *weak mobility*, since threads' execution state (stack and program counter) cannot be saved and restored

- On arrival the process is simply executed from the start
- It is up to the programmer to keep track of the execution state of the process
- However, all the process fields' values are restored on arrivale

Java byte-code mobility

Starting from these interfaces, the package `mobility` provides concrete classes that automatically deal with

- migration of Java objects together and their byte-code;
- deserializing transparently such objects by dynamically loading their transmitted byte-code.

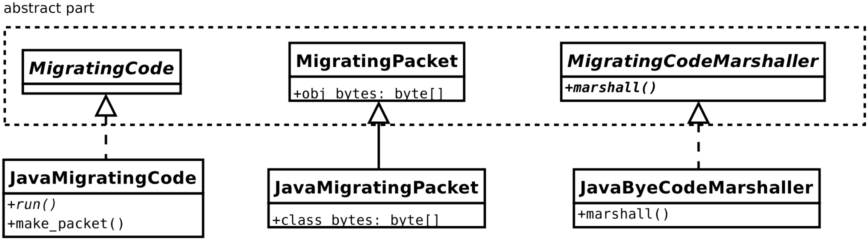


Figure: Main classes of the package

Migrating Java code

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

- the serialized object
- the byte code of all the classes used by the migrating object (these classes are collected by `make_packet` using *Java Reflection API*)

Migrating Java code

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

- the serialized object
- the byte code of all the classes used by the migrating object (these classes are collected by `make_packet` using *Java Reflection API*)

Scalable to inheritance

`make_packet` needs not be redefined in derived classes: it handles subclasses transparently and automatically.

Collecting classes

getUsedClasses

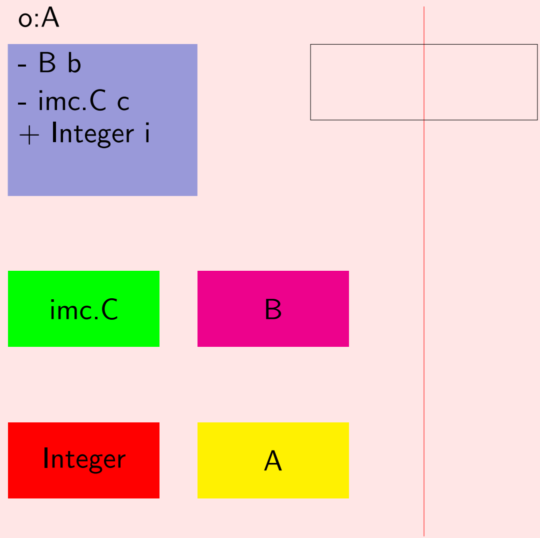
```
protected void getUsedClasses( Class c ) {
  if (c == null || ! addUsedClass( c )) return ;

  Field[] fields = c.getDeclaredFields() ;
  Constructor[] constructors = c.getDeclaredConstructors() ;
  Method[] methods = c.getDeclaredMethods() ;
  int i ;

  for ( i = 0 ; i < fields.length ; i++ )
    getUsedClasses( fields[i].getType() ) ;

  for ( i = 0 ; i < constructors.length ; i++ ) {
    getUsedClasses( constructors[i].getParameterTypes() ) ;
    getUsedClasses( constructors[i].getExceptionTypes() ) ;
  }
  ...
```

Collecting classes

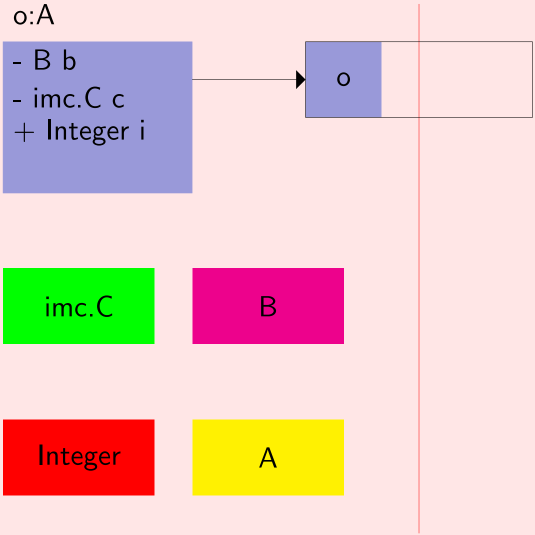
Some classes are excluded by the collection:

- Classes belonging to standard libraries
  - these classes are given additional privileges, thus they must be loaded from the local file system
- classes of specific packages (e.g., the IMC package) that must be present in the remote sites
- classes explicitly excluded by the programmer

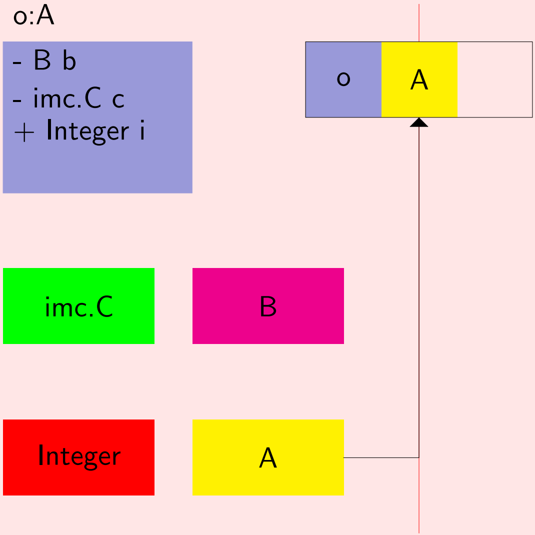
Code Mobility: example



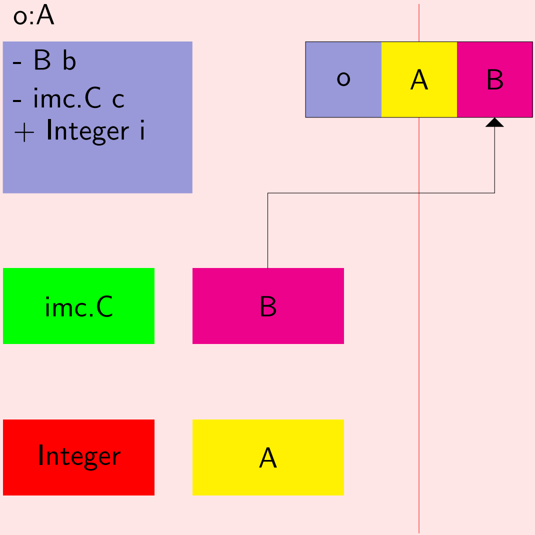
Code Mobility: example



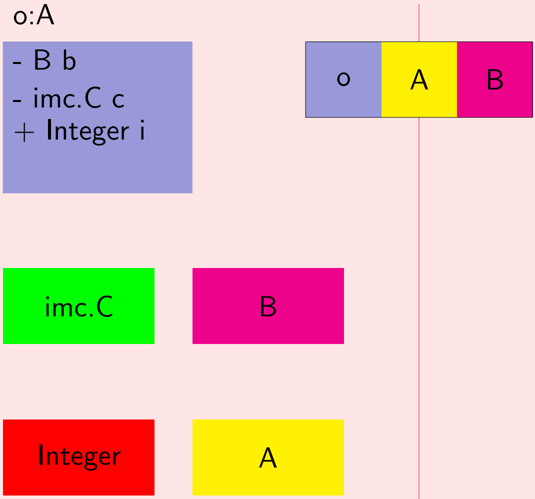
Code Mobility: example



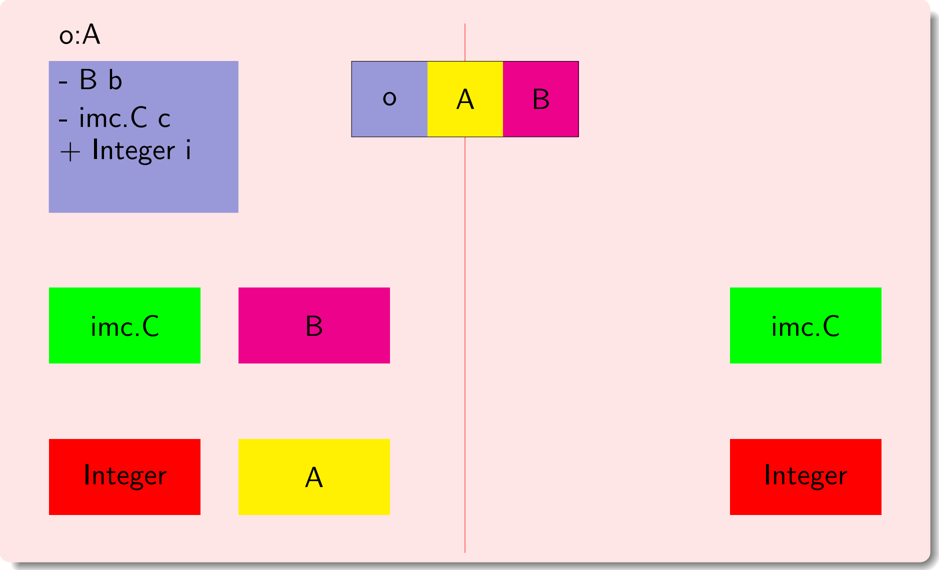
Code Mobility: example



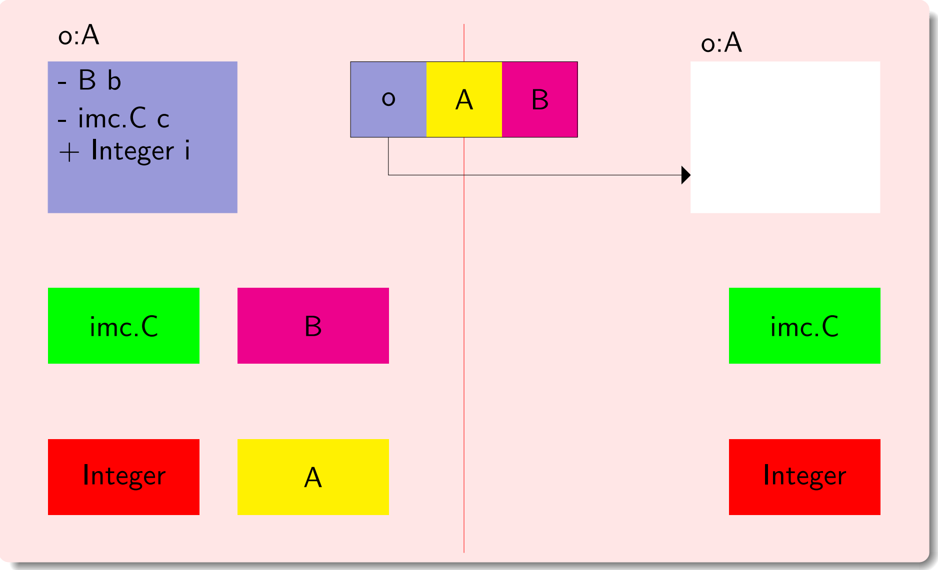
Code Mobility: example



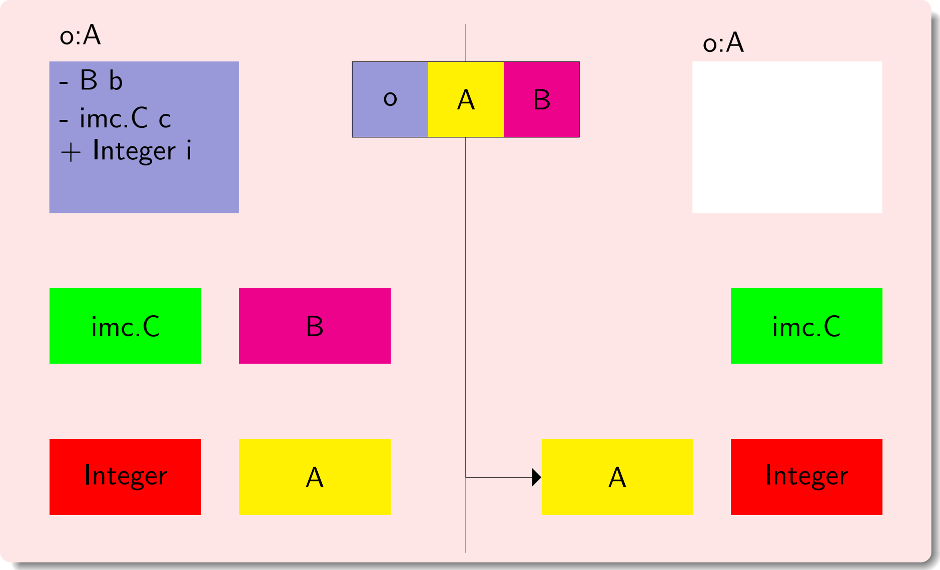
Code Mobility: example



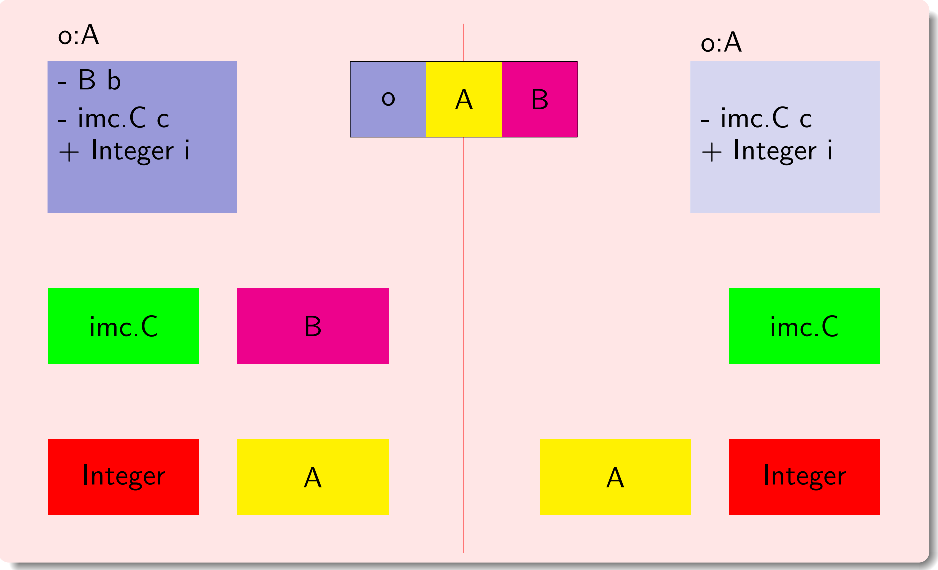
Code Mobility: example



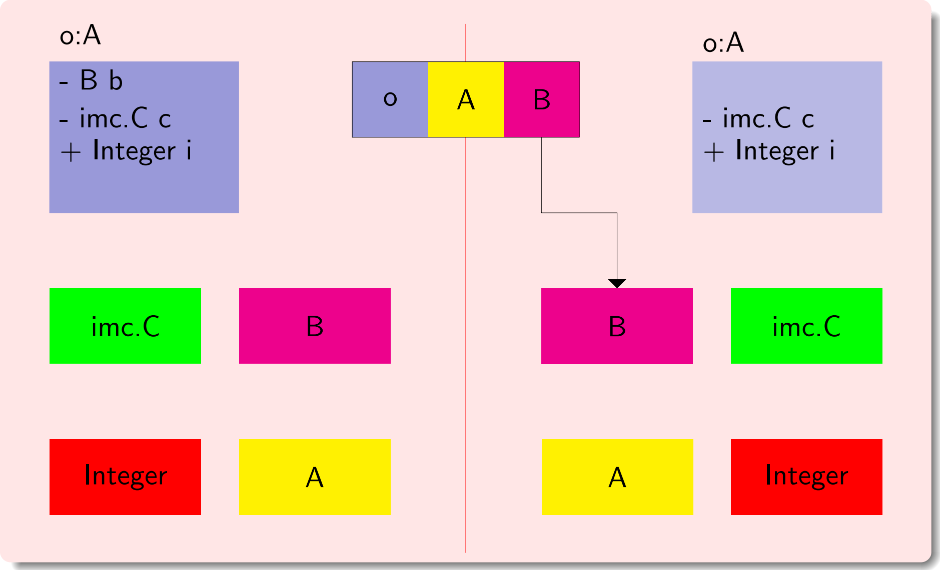
Code Mobility: example



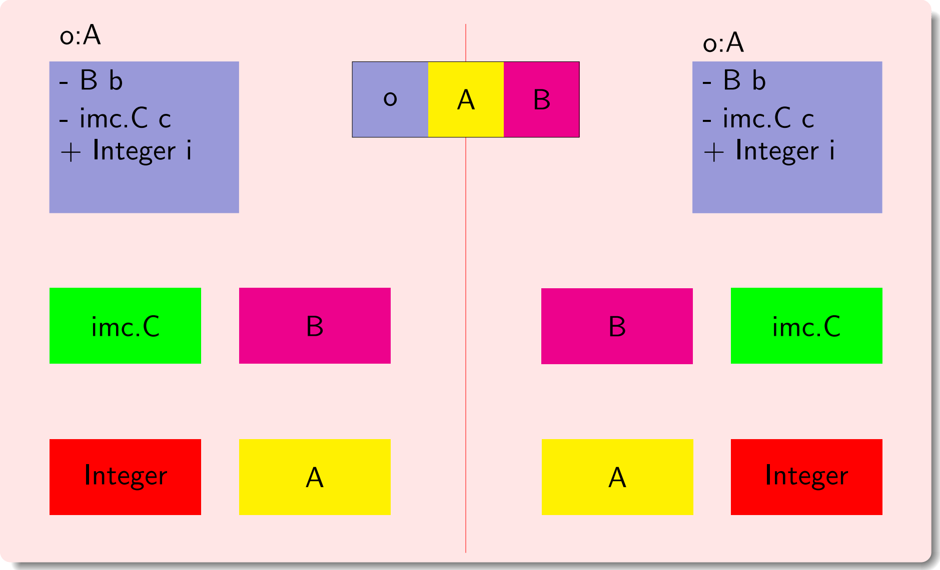
Code Mobility: example



Code Mobility: example



Code Mobility: example



NodeClassLoader

Each site willing to accept migrating code will internally use a `NodeClassLoader` provided by the package

- When a migrating code is received from the network in a `JavaMigratingPacket`
- its classes are stored in the `NodeClassLoader` internal table
- the object is deserialized
- during deserialization, needed classes are loaded by `NodeClassLoader` from the internal class table

Retrieving class byte code

When a `MigratingPacket` is created the byte code of the collected classes is retrieved:

- by the local file system, in the originating site
- by the `NodeClassLoader` table in the other sites

This enables a migrating code to visit many sites in sequence.

DPI programming model

- DPI is a locality-based extension of π:
  - Processes are distributed over a set of nodes (or *locations*) each of which is identified by a *name* or *locality*.
- Like in π calculus, processes interact each other via message passing over *channels*.
- Only local communication is permitted
  - Two processes can interact only if they are located on the same node.
- Processes can change their execution environment performing action go .
- DPI does not assume a specific network topology
  - a variant of DPI, DPIF, considers explicit link

DPI programming model

- DPI is a locality-based extension of π:
  - Processes are distributed over a set of nodes (or *locations*) each of which is identified by a *name* or *locality*.
- Like in π calculus, processes interact each other via message passing over *channels*.
- Only local communication is permitted
  - Two processes can interact only if they are located on the same node.
- Processes can change their execution environment performing action go .
- DPI does not assume a specific network topology
  - a variant of DPI, DPIF, considers explicit link

DPI syntax

Systems

$$
\begin{array}{rcll}
M, N & ::= & \mathbf{0} & \text{Empty} \\
 & | & M|N & \text{Composition} \\
 & | & (\nu e).N & \text{Restriction} \\
 & | & l[\![P]\!] & \text{Agent}
\end{array}
$$

Threads

$$
\begin{array}{rcll}
P, Q, R & ::= & \mathtt{stop} & \text{Termination} \\
 & | & P|Q & \text{Composition} \\
 & | & (\nu e)P & \text{Restriction} \\
 & | & \mathtt{go}\ \ u.P & \text{Movement} \\
 & | & u!\langle v\rangle.P & \text{Output} \\
 & | & u?(X).P & \text{Input} \\
 & | & \mathtt{rec}\ A.P & \text{Recursion} \\
 & | & A & \text{Process Identifier} \\
 & | & \mathtt{if}\ u = v\ \mathtt{then}\ P\ \mathtt{else}\ Q & \text{Matching}
\end{array}
$$

JDPI implementation...

- `JDpiProtocol`:
  - implements the communication protocol between DPI nodes;
- `JDpiProcess`:
  - implements a generic DPI process;
  - is a `JavaMigratingCode`.
- `JDpiNode`:
  - implements a DPI node;
  - provides a computational environment for `JDpiProcesses`;
  - manages local channel interactions;
- `JDpiLocality`:
  - is an abstraction for *node names*
- Two kinds of topology have been implemented:
  - *flat*;
  - a la DPIF.

Implementing DPI

Node

DPI nodes provide:

- a computational environment for processes;
- functionalities for processes interactions

Implementing DPI

Node

DPI nodes provide:

- a computational environment for processes;
- functionalities for processes interactions

Implemented as...

- an extension of `org.mikado.topology.Node`;
- new methods for managing channels:
  - creation;
  - input;
  - output.

Implementing DPI

Processes

- a DPI process (`JDpiProcess`) is implemented as a subclass of the class `NodeProcess`
  - each process has to provide method *body()*;
  - implements the `JavaMigratingCode`;
  - allows to easily migrate a process to a remote site.
- Processes can be added to a node for execution with the method `addProcess`
- Each process interacts with the hosting node using a `JDpiNodeProxy`
  - provides an access to node methods;

Implementing DPI: Example 1

Processes

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

Implemented in Java

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

Implementing DPI: Example 2

Processes

$$\nu a.ex!a$$

Implemented in Java

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

Implementing DPI: Example 3

An agent that migrates over a set of *electronic markets*, in search of a best price. At the end of the search, the agent migrates to locality `home` and provides its result on channel `result`.

Implemented in Java

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

Metrics

The Java implementation JDPI is composed of

- about 1000 lines of code
- only 28 classes:
  - provide 152 methods,
  - the average number of lines per method is 3.5

Ongoing works

- Implementing security policies
  - Calculus of Membranes;
  - Types for access control;
  - Logical specification of security policies;
  - ...
- Consider process interaction based on `XML`
  - query and pattern matching
- Reimplement KLAVA (the run-time system for KLAIM) using IMC
- Integration in Eclipse

Ongoing works

- Implementing security policies
  - Calculus of Membranes;
  - Types for access control;
  - Logical specification of security policies;
  - ...
- Consider process interaction based on `XML`
  - query and pattern matching
- Reimplement KLAVA (the run-time system for KLAIM) using IMC
- Integration in Eclipse

Services (project SENSORIA)

- Add features to deal with services
- Single out recurrent mechanisms of calculi for SOC (e.g., SCC, COWS, ...)

References

`http://imc-fi.sourceforge.net`

- papers
- documentation
- GPL software

References

`http://imc-fi.sourceforge.net`

- papers
- documentation
- GPL software

Thanks!