IMC and its components: Design and Use

Lorenzo Bettini

Dipartimento di Sistemi e Informatica, Università di Firenze
bettini@dsi.unifi.it

MIKADO Meeting

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

- A middleware/framework for implementing distributed and mobile code run-time systems

IMC - Implementing Mobile Calculi

- A middleware/framework for implementing distributed and mobile code run-time systems
- Help the implementer of a distributed mobile code application:
  - he relies on the framework for recurrent standard mechanisms and patterns
  - he concentrates on the features that are specific of that particular language
  - he can extend/customize the framework for these features

IMC - Implementing Mobile Calculi

- A middleware/framework for implementing distributed and mobile code run-time systems
- Help the implementer of a distributed mobile code application:
  - he relies on the framework for recurrent standard mechanisms and patterns
  - he concentrates on the features that are specific of that particular language
  - he can extend/customize the framework for these features
- Provides components for
  - Network topology
  - Communication protocols
  - Code mobility

IMC - Implementing Mobile Calculi

- A middleware/framework for implementing distributed and mobile code run-time systems
- Help the implementer of a distributed mobile code application:
  - he relies on the framework for recurrent standard mechanisms and patterns
  - he concentrates on the features that are specific of that particular language
  - he can extend/customize the framework for these features
- Provides components for
  - Network topology
  - Communication protocols
  - Code mobility
- Favor object composition to class inheritance, in order to achieve a dynamic code reuse

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

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.

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

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.

The automatic approach (default)

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

Main base classes

- The package defines the empty interface `MigratingCode` that must be implemented by classes representing code that migrates.
- Migrating code is transmitted inside a `MigratingPacket`:

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

Marshalling and Unmarshalling

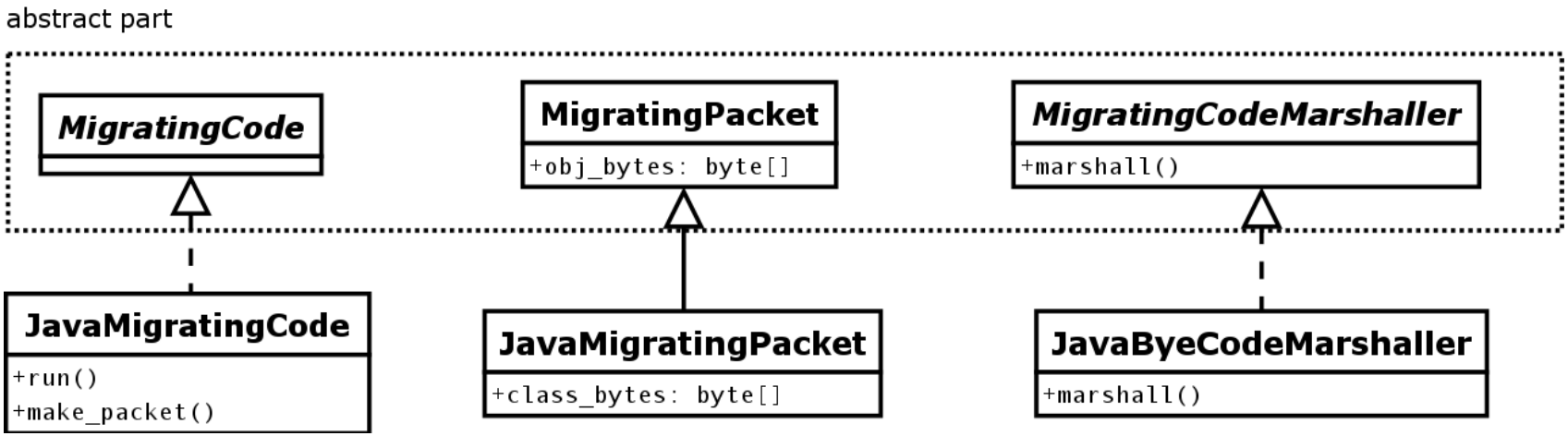
```
public interface MigratingCodeMarshaller {
    MigratingPacket marshal(MigratingCode code) throws IOException;
}

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

Used by the `protocols` package.
The real marshaller and unmarshaller are created by the "abstract factory" of the framework.

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.



Migrating Java code

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

The method `make_packet` builds a packet with:

- 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*)

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

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

Marshalling and Unmarshalling Java code

public class JavaByteCodeMarshaller
implements MigratingCodeMarshaller {...}

Rely on the `make_packet` method of `JavaMigratingCode`

public class JavaByteCodeUnMarshaller
implements MigratingCodeUnMarshaller {...}

Rely on `NodeClassLoader` for loading and deserializing

A Tutorial Example

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

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

where `MyVar`, `MyRetType`, `MyPar` are classes defined by the programmer of this code (i.e., not present at remote sites)

The sender

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

Upon creation of the packet, the byte code for `MyCode` and `MyVar`, `MyRetType` and `MyPar` is stored into the packet.

The receiver

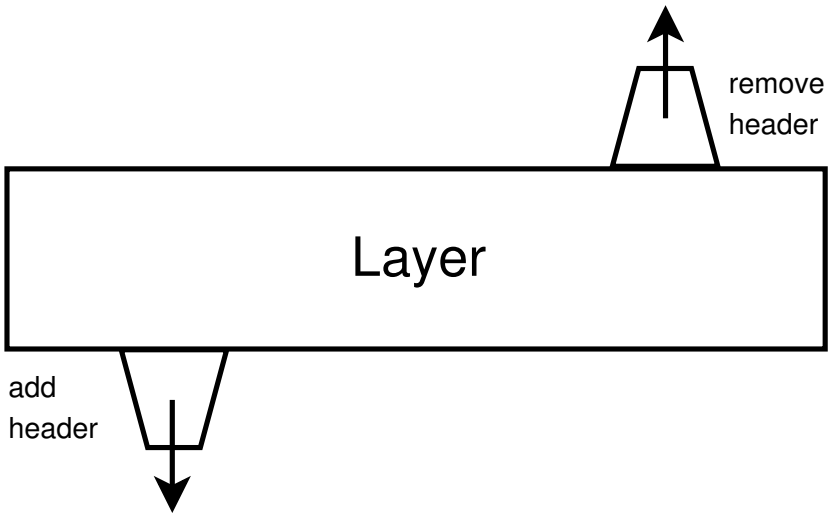
```
public class Receiver {
  ...
  JavaMigratingCode receiveCode(InputStream is) throws Exception {
    MigratingCodeUnMarshaller unmarshaller =
      new JavaByteCodeUnMarshaller();
    ObjectInputStream obj_is = new ObjectInputStream(ss);
    MigratingPacket pack = (MigratingPacket) obj_is.readObject();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

After `unmarshal` returns, the original object is restored in the remote site, and its classes are loaded.

Protocol Layers

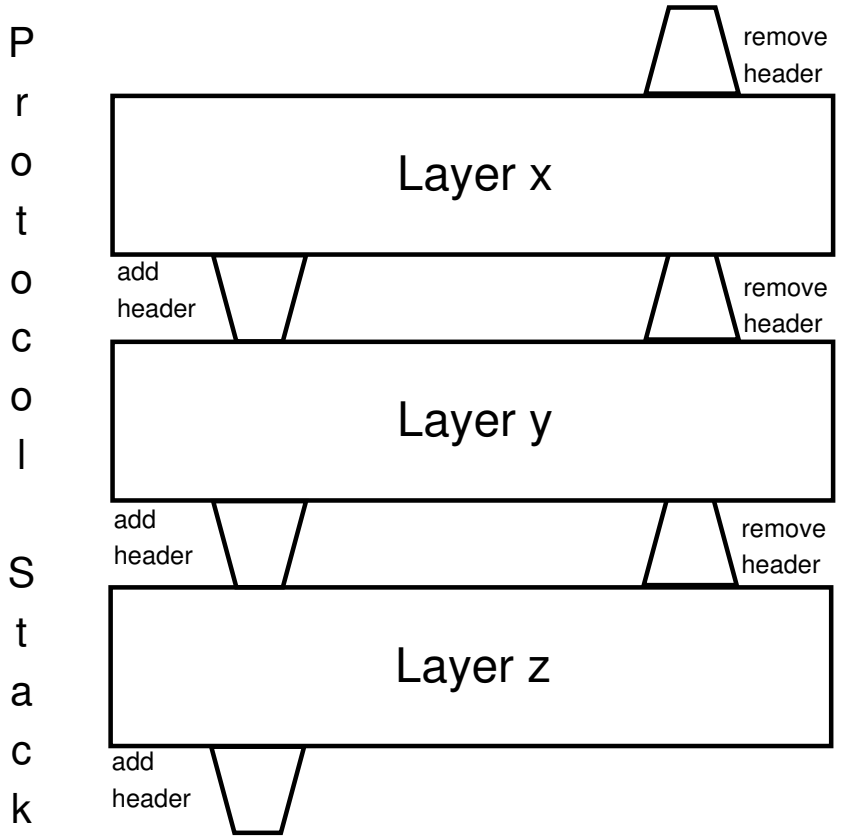
Extend the TCP/IP protocol stack

- Remove header information from the input
- Add header information in the output



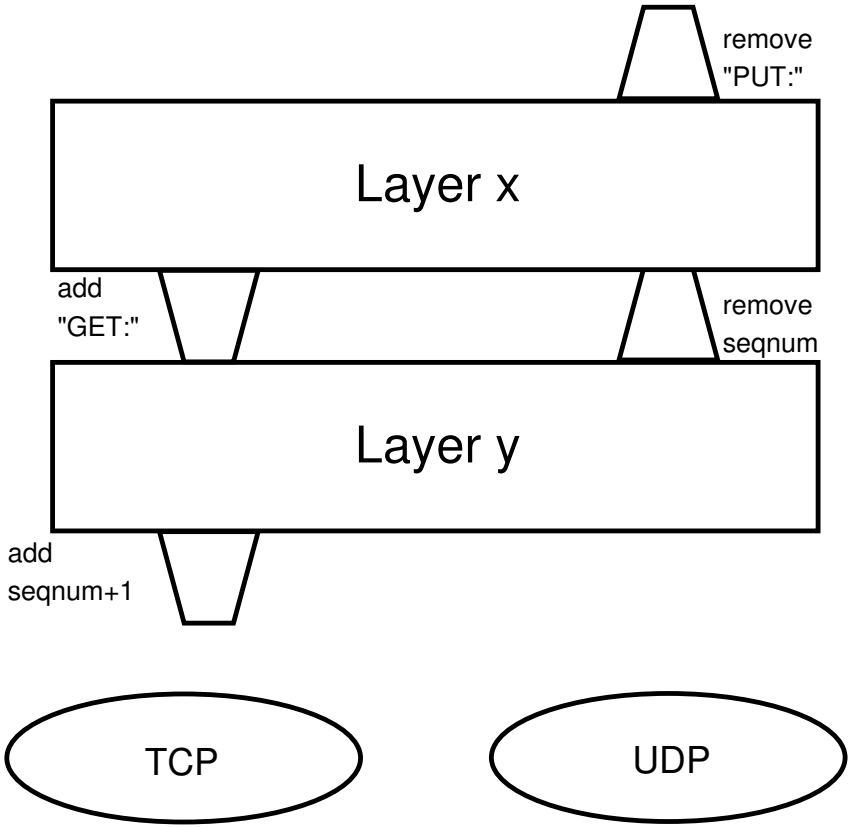
Compositional Layers

- Protocol layers are composed in a *Protocol Stack*.
- Each layer abstract from the lower and higher layer.



Example

The lowest level can be a TCP Socket or UDP Datagrams or whatever you want (even a stream attached to a file, e.g., for simulation)



The `ProtocolLayer` class

```
public class ProtocolLayer {
  protected ProtocolLayer next;

  public void up() throws ProtocolException
  protected void doUp() throws ProtocolException
  public void prepare() throws ProtocolException
  protected void doPrepare() throws ProtocolException
}
```

- The users of a layer can only call `up` and `prepare`:
  - `up` and `prepare` call all the implementation methods `doUp` and `doPrepare` in the stack of layers starting from the lowest layer

Marshalers and UnMarshalers

A protocol layer uses a `Marshaler` and an `UnMarshaler` to write/read from the actual communication layer.
They provide means to write/read any primitive data type (inherited by `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;
}
```

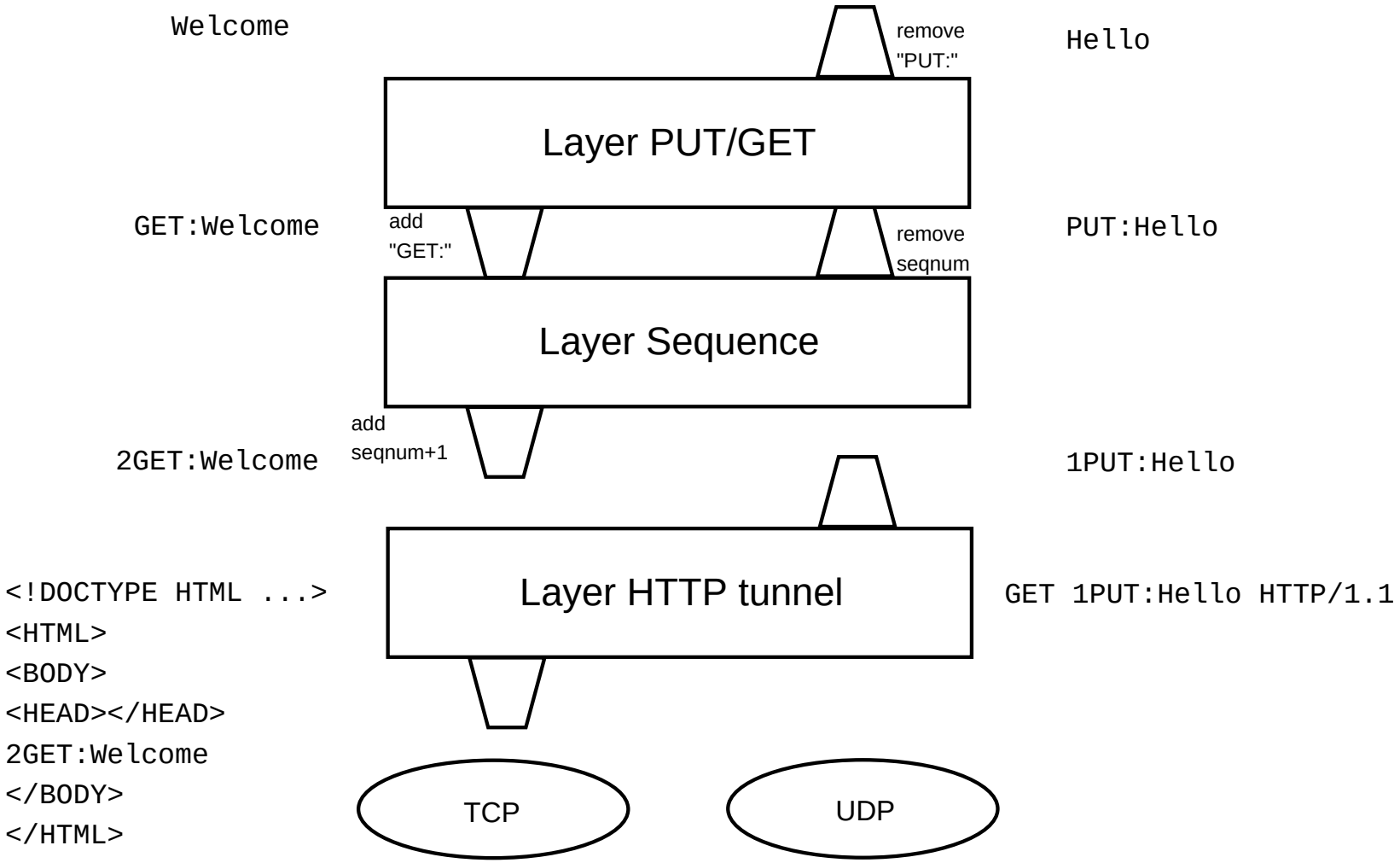
A protocol layer example: PUT/GET

```
public class PutGetProtocolLayer extends ProtocolLayer {
    // Removes a string "PUT:" from the input.
    protected void doUp() throws ProtocolException {
        String req = getCode(getUnmarshaller());
        if (!req.equals("PUT:")) {
            getMarshaller().writeStringLine("UNKNOWN REQUEST");
            getMarshaller().flush();
        }
    }

    // Inserts a "GET:" before the actual payload.
    protected void doPrepare() throws ProtocolException {
        getMarshaller().writeBytes("GET:");
    }
}
```

Tunneling

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

Tunneling


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

Router layer

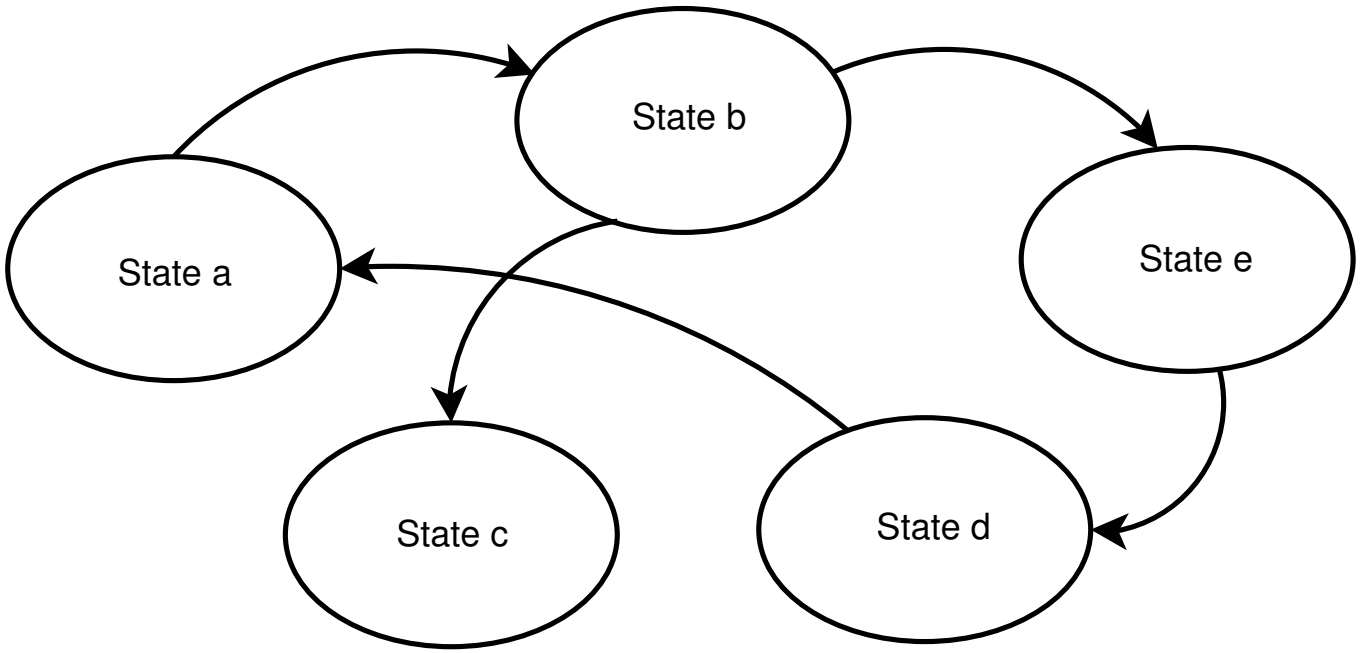
- Reads the destination address
  - if the destination is the same of this node let it go through the stack
  - otherwise forward it to the actual destination node
- This can act on a specific layer of the stack

Router layer

- Reads the destination address
  - if the destination is the same of this node let it go through the stack
  - otherwise forward it to the actual destination node
- This can act on a specific layer of the stack
- The same way you can implement a firewall that acts on a specific layer

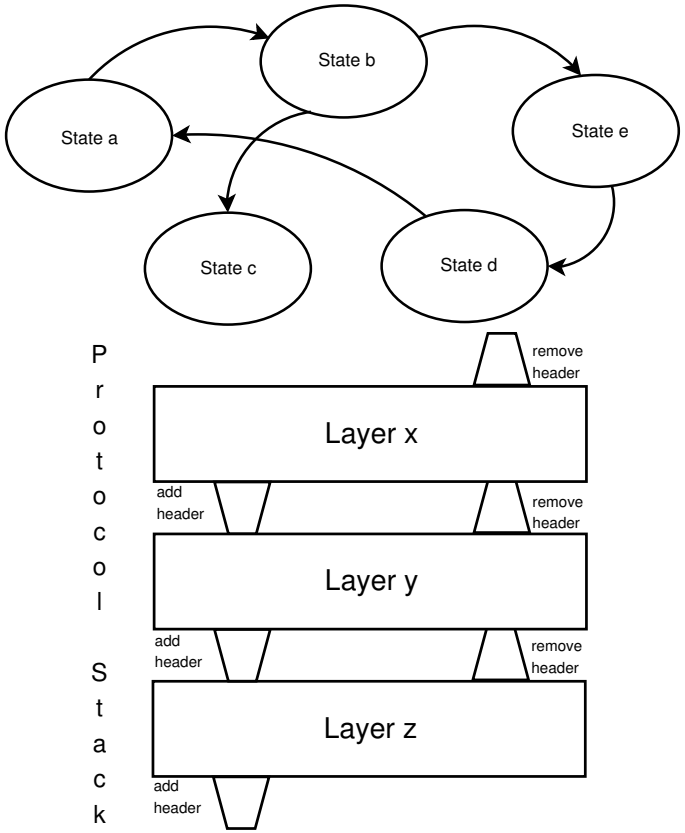
Protocol State

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:



The class Protocol

The method `start` executes one protocol state at time:

```
public final void start() throws ProtocolException {
    String next_state_id = START;
    ProtocolState next_state;

    do {
      next_state = getState(next_state_id);
      next_state_id = next_state.enter();
    } while (next_state_id != END);

    // always try to execute the end state of the protocol
    getState(END).enter();
}
```

The class `ProtocolState`

- Derive from the class `ProtocolState`
- Implement the method `enter`:
  - Use the protocol stack to read and write
  - Return the next state

```
public class MyState extends ProtocolState {
  public String enter() throws ProtocolException {
    // read something from the stack
    if (ok) {
      // write something in response
      return "state_a"
    } else {
      // write error response
      return "state_b"
    }
  }
}
```

Example: building a protocol

```
Protocol protocol = new Protocol(new MyProtocolStack());
protocol.setState(Protocol.START, new MyStartState());
protocol.setState("state_a", new MyStateA());
protocol.setState("state_b", new MyStateB());
protocol.setState("state_c", new MyStateC());
protocol.setState(Protocol.END, new MyEndState());
protocol.start();
```

Connectivity features

Provide basic connectivity classes:

- `ConnectionManagementState`: a protocol state that deals with incoming connection/disconnection requests
- `ConnectState`: a protocol state that deals with establishing a new connection
- `ConnectionManager`: keeps trace of all the established connections
- `Node`: container of executing processes

Connectivity features

Connections are logical and independent from the low level network layer, e.g.:

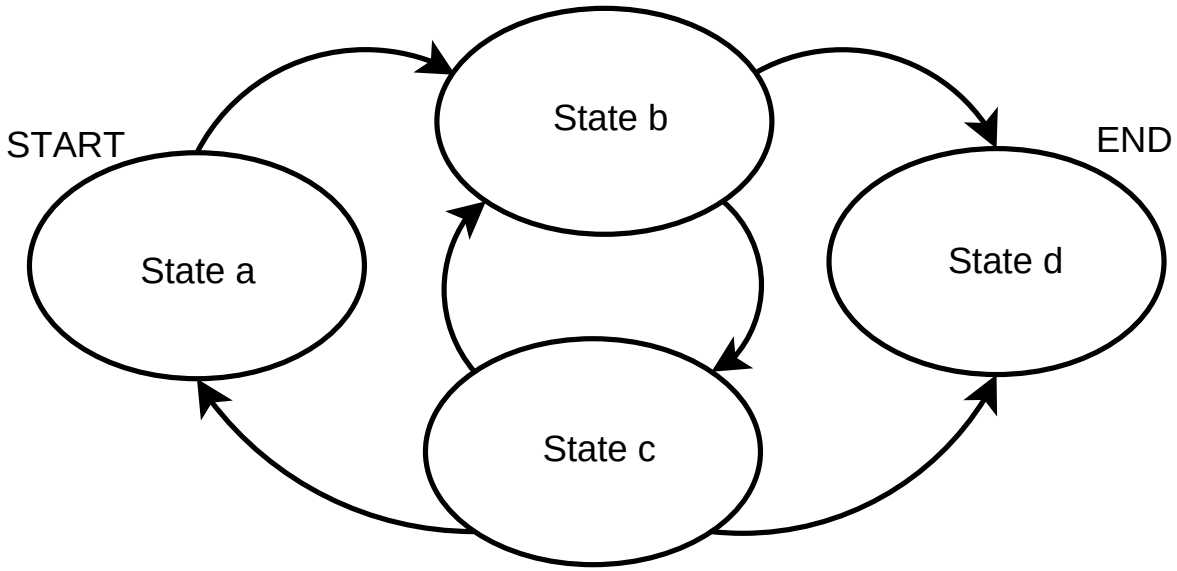
- Real physical connections for TCP sockets
- Logical connections if UDP packets are used
  - possibly with "keep-alive" special packets

The class ConnectionManagementState

```
public String enter() throws ProtocolException {
    while (true) {
      up(); // notifies the layer it wants to read
      String received = getUnmarshaller().readStringLine();
      prepare(); // prepares for writing
      if (received.equals(connection_string)) {
        addConnection(protocolStack);
        getMarshaller().writeStringLine(ok_string);
        return next_state;
      } else if (received.equals(disconnection_string)) {
        removeConnection(getSessionId());
        getMarshaller().writeStringLine(ok_string);
        return next_state;
      } else {
        getMarshaller().writeStringLine(fail_string);
      }
      down(); // flushes writing
    }
}
```

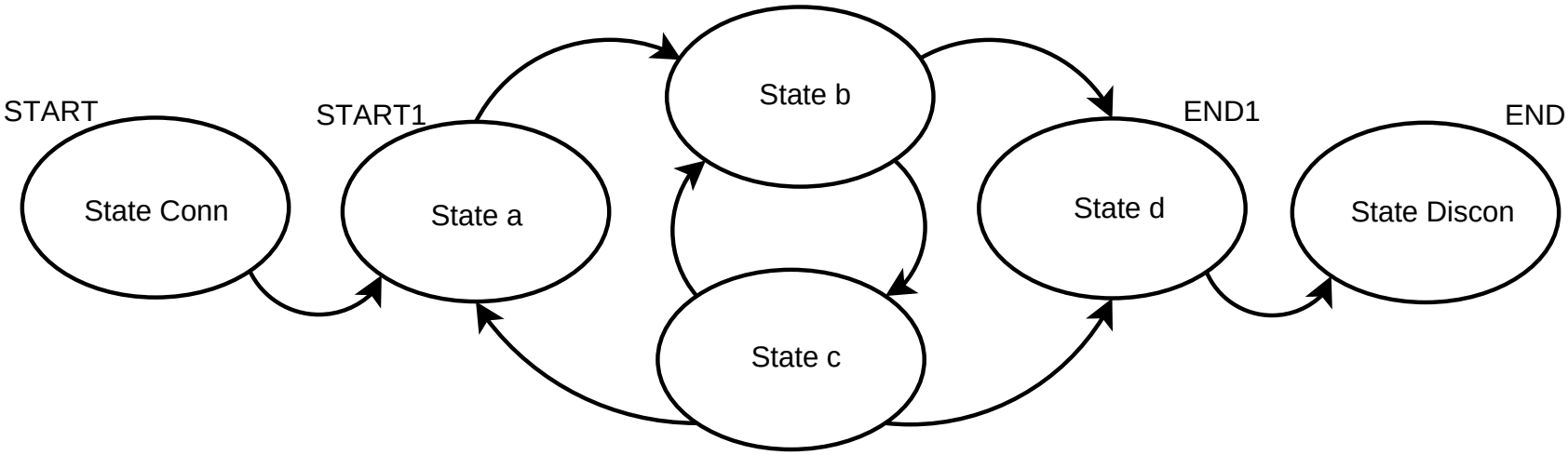
Manipulating protocols

Features to "push" dynamically a new START and END state into an existing protocol.



Manipulating protocols

Existing START and END state are renamed, so that the previous protocol states keep its original behavior



Example: building a protocol (2)

```
Protocol protocol = new Protocol(new MyProtocolStack());
protocol.setState(Protocol.START, new MyStartState());
protocol.setState("state_a", new MyStateA());
protocol.setState("state_b", new MyStateB());
protocol.setState("state_c", new MyStateC());
protocol.setState(Protocol.END, new MyEndState());
...
protocol.insertStart(new ConnectState());
protocol.insertEnd(new DisconnectState());
protocol.start();
```

Node connectivity

The class `Node` already provides features to receive and establish a connection, e.g.:

- **public void** accept(**int** port, Protocol protocol)

  waits for a (physical) connection on the specified port and "decorates" the passed protocol with a start connection state and an end disconnection state

- **public void** connect(String host, **int** port, Protocol protocol)

  establishes a (physical) connection on the specified port and "decorates" the passed protocol with a start connection state and an end disconnection state

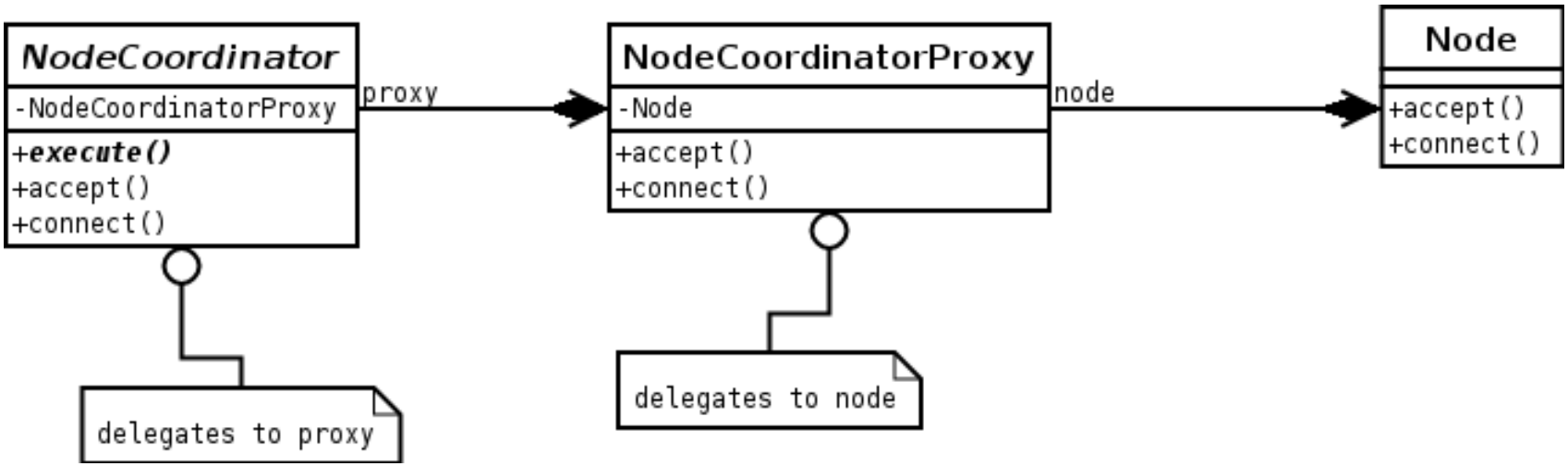
- They also set the low level protocol layer (e.g., a socket) in the passed protocol.

Node coordinators

Node coordinators are super user processes:

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

Standard processes cannot execute privileged actions.



Example

```
public class MyCoordinator extends NodeCoordinator {
  public void execute() throws IMCException {
    Protocol protocol = new MyProtocol();
    accept(9999, protocol);
    protocol.start();
  }
}

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

Example (2)

```
public class AcceptNodeCoordinator extends NodeCoordinator {
    private ProtocolFactory protocolFactory;
    private int port;

    public AcceptNodeCoordinator(ProtocolFactory protocolFactory, int port) {
        this.protocolFactory = protocolFactory;
        this.port = port;
    }

    public void execute() throws IMCException {
        while (true) {
            Protocol protocol = protocolFactory.createProtocol();
            accept(port, protocol);
            new ProtocolThread(protocol).start();
        }
    }
}
```

Events

- The classes of the framework generate events
  - connection event
  - disconnection event
  - etc.
- You can intercept an event with a "listener" (subclass of `EventListener`) by registering it for a specific event

Logical names

Associate a logical name to a physical address:

- `NameManager`: keeps trace of the associations
- `SubscribeState`: protocol state that deals with a name registration
- `UnSubscribeState`: protocol state that deals with a name cancellation

The subscription protocol state should be executed after a successful connection state.
The unsubscription protocol state should be executed before a disconnection state.

Re-engineering of existing systems

- The run-time system for the KLAIM language:
  - Tuple space based communication
  - Multiple tuple spaces distributed over nodes of a net with a possible hierarchical structure
  - Processes are first entity data that can be exchanged among nodes
- The mobility code management was completely removed from KLAVA that now completely relies on the `mobility` package for that.
- The integration of `protocols` and `topology` is ongoing work

Implementing $D\pi$

The implementation of the run-time system for $D\pi$ was started, using from scratch all the packages of the IMC framework.

$D\pi$

A locality based extension of the π-calculus with processes located at sites.
The go primitive allows processes to migrate to remote sites.

References

`http://music.dsi.unifi.it`

- papers
- documentation
- GPL software