Miko by Example

Francisco Martins

fmartins@di.fc.ul.pt

University of Azores

Joint work with:

Liliana Salvador
lsalvador@ncc.up.pt

Luís Lopes
lblopes@ncc.up.pt

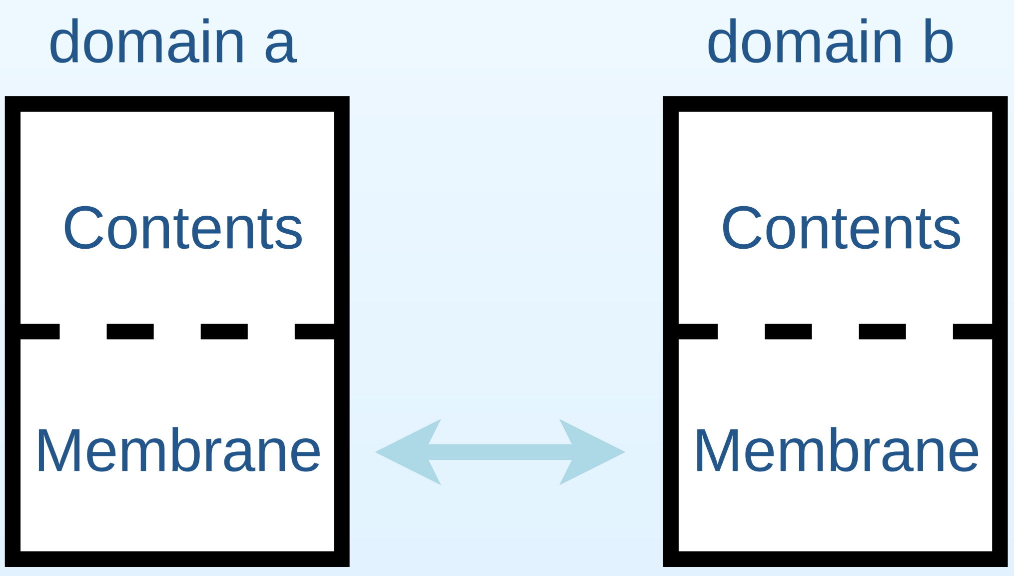
Vasco Vasconcelos
vv@di.fc.ul.pt

Outline

- Miko's programming style
  - Membranes are programmed separately from contents
  - Peer-to-peer communication
- The language
  - Programming membranes and contents
- Examples
  - Establishing a session between a client and a server
  - Membrane local state: counting active sessions
  - A mathematical server

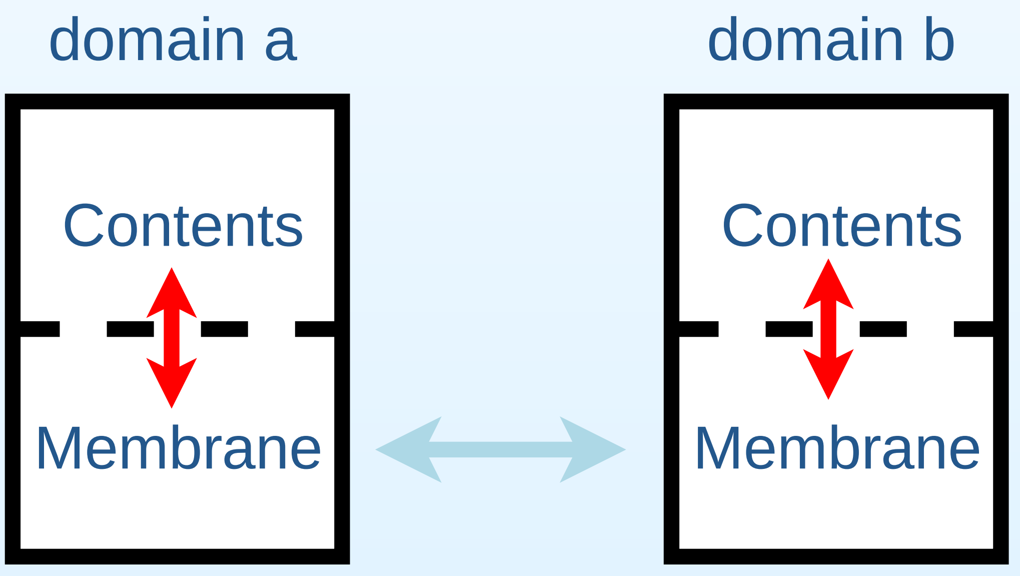
Miko programming style

- Membranes implement the communication protocol between domains
  - (membrane to membrane communication)



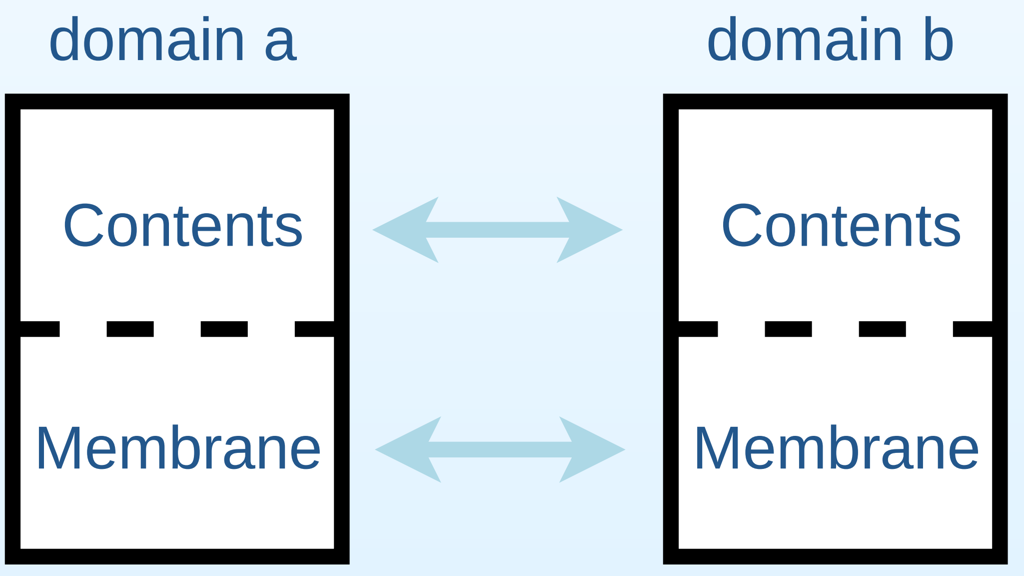
Miko programming style

- Membranes implement the communication protocol between domains
  - (membrane to membrane communication)
- Contents interacts with the domain's membrane
  - (contents to membrane communication)



Miko programming style

- Membranes implement the communication protocol between domains
  - (membrane to membrane communication)
- Contents interacts with the domain’s membrane
  - (contents to membrane communication)
- Peer-to-Peer communication



Programming membranes

- Programmed in a separated file (from the contents)

```
Membrane {






}
```

Programming membranes

- Programmed in a separated file (from the contents)
- Import domain interfaces/share resources

```
Membrane {
  Import a_1, ..., a_n
  Share c_1, ..., c_n



}
```

Programming membranes

- Programmed in a separated file (from the contents)
- Import domain interfaces/share resources
- Offer a set o methods as the domain interface

Membrane {
  Import $a_1, \ldots, a_n$
  Share $c_1, \ldots, c_n$
  {
    $\mathsf{method}_1(\tilde{x}_1) = S_1$
      $\ldots$
    $\mathsf{method}_n(\tilde{x}_n) = S_n$
  }

}

Programming membranes

- Programmed in a separated file (from the contents)
- Import domain interfaces/share resources
- Offer a set o methods as the domain interface
- Contain a computation shell to hold a local state

`Membrane {`
  `Import` $a_1, \ldots, a_n$
  `Share` $c_1, \ldots, c_n$
  `{`
    $\text{method}_1(\tilde{x}_1) = S_1$
      $\ldots$
    $\text{method}_n(\tilde{x}_n) = S_n$
  `}`
  S
`}`

Programming contents

- Select methods at the domain’s membrane

```
Contents {


}
```

Programming contents

- Select methods at the domain's membrane
- Import domain interfaces/share resources

Contents {
  Import $a_1, \ldots, a_n$
  Share $c_1, \ldots, c_n$

}

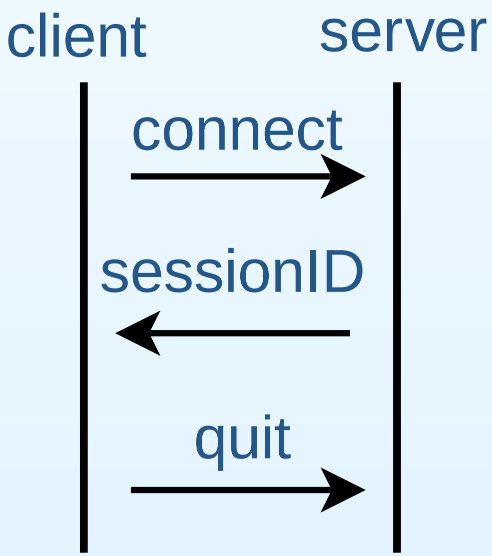
Programming contents

- Select methods at the domain's membrane
- Import domain interfaces/share resources
- Is the computational shell of the domain

`Contents {`
  `Import` $a_1, \ldots, a_n$
  `Share` $c_1, \ldots, c_n$
  P
`}`

A client-server session manager

- Implements the concept of a session.
- Server's membrane
  - provides a *connect* and a *disconnect* method as network interface
- Client's membrane
  - provides a *connect*, an *enter*, and a *disconnect* method.



Server's membrane implementation

```
Membrane {
  {
    connect (client, replyTo) =
      new sessionID
      out [client, enter [() replyTo ! [sessionID] ] ] |
      in [
        sessionID ? {
          quit () = inaction
        }
      ]

    disconnect (sessionID) =
      in [sessionID ! quit [] ]
  }
  inaction
}
```

Client's membrane implementation

```
Membrane {
  {
    connect (server, replyTo) =
      out [ server, connect [myDomain, replyTo] ]

    enter (x) =
      in [ x [] ]

    disconnect (server, sessionID) =
      out [server, quit [sessionID] ]
  }
  inaction
}
```

Client-server communication

```
Contents {
  import S1

  new connection
  myDomain ! connect [ S1, connection ] |
  connection ? (sessionID) myDomain ! disconnect [sessionID]
}
```

Client-server communication

```
Contents {
  import S1

-- new connection
-- myDomain ! connect [ S1, connection ] |
-- connection ? (sessionID) myDomain ! disconnect [sessionID]

  let
    sessionID = myDomain ! connect [ S1 ]
  in
    myDomain ! disconnect [sessionID]
}
```

Controlling the number of clients

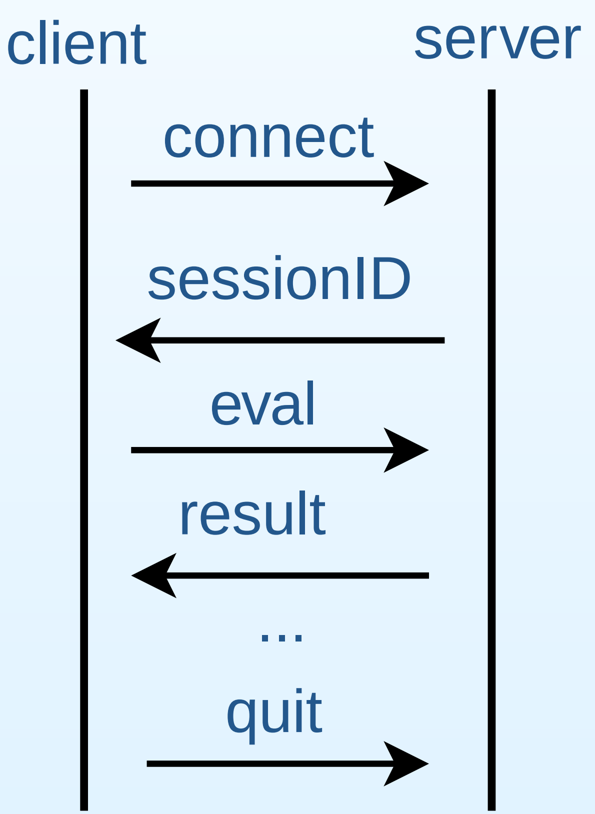
```
Membrane {
  new myController
  {
    connect (client, replyTo) =
      myController ! connect [client, replyTo]
    disconnect (sessionID) =
      myController ! disconnect [sessionID]
  }
```

Controlling the number of clients

```
Membrane {
  new myController
  {...}
  def Controller (counter, max) =
    myController ? {
      connect (client, replyTo) =
        if counter < max
        then ... | Controller [counter + 1, max]
        else Controller [counter, max]
      disconnect (targetDomain, process) =
        ... | Controller [counter-1, max]
    }
  in
    Controller [0, 5]
}
```

A Math server

- Server's membrane
  - provides: *connect*, *disconnect*, *eval*, and *replyResult*
- Client's membrane
  - provides: *connect*, *enter*, *disconnect*, and *eval*



A Math server membrane

```
Membrane{
 {
   connect (client, replyTo) =
     myController ! connect [client, replyTo]

   disconnect (sessionID) =
     myController ! disconnect [sessionID]

   eval (x) =
     in [x [] ]

   replyResult (client, x) =
     out [client, enter [x]]
 }
```

A Math server membrane

```
connect (server, replyTo) = . . .
in [
  def
      Session (self, client) =
        self ? {
          add (n, m, replyTo) =
            myDomain ! replyResult [client, () replyTo ! [n + m]] |
            Session [self, client]
          neg (n, replyTo) =
            myDomain ! replyResult [client, () replyTo ! [0 - n]] |
            Session [self, client]
          disconnect () =
            inaction
      }
  in Session [sessionID, client]
]
```

A Math client membrane

```
Membrane{
 {
   connect (server, replyTo) =
     out [ server, connect [myDomain, replyTo] ]

   enter (x) =
     in [ x [] ]

   disconnect (server, sessionID) =
     out [server, quit [sessionID] ]

   eval (server, x) =
     out [server, eval [x] ]
 }
 inaction
}
```

Interaction with a math server

```
Contents {
 import S1

 let
   sessionID = myDomain ! connect [ S1 ]
 in
   new result
   myDomain ! eval [() sessionID ! add [3, 4, result] ] |
   result ? {
      val (x) = myDomain ! eval [() sessionID ! neg [x, result]] |
         result ? {
            val (x) = io ! printi [x] |
            myDomain ! disconnect [sessionID]
         }
   }
}
```

Future work

- Add a notion of private and public interface
- Finish the compiler
- Change the virtual machine to use the IMC framework